



PhD-FSTM-2025-080

The Faculty of Science, Technology and Medicine

## DISSERTATION

Defence held on 01/07/2025 in Luxembourg

to obtain the degree of

### DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG EN INFORMATIQUE

by

**XUNZHU TANG**

Born on 15<sup>th</sup> May 1991 in Jiangsu (China)

### AUTOMATING THE ANALYSIS AND GENERATION OF CODE CHANGES WITH FOUNDATION MODELS

#### Dissertation Defence Committee

Dr. Tegawendé F. Bissyandé, Dissertation Supervisor  
*Associate Professor, University of Luxembourg*

Dr. Jacques Klein, Chairman  
*Full Professor, University of Luxembourg*

Dr. Jean-Rémy Falleri  
*Full Professor, Bordeaux INP*

Dr. Bach Le  
*Associate Professor, University of Melbourne*

Dr. Kui Liu  
*Research Scientist, Huawei*



# Abstract

Code changes drive software evolution, yet their growing volume and complexity—with maintenance consuming 70% of development costs—demand sophisticated automated analysis. While existing approaches range from token-based sequences to AST-based structures and pre-trained language models, they fail to capture the multi-faceted nature of code changes: their intent, semantic context, and structural transformations. This dissertation presents a comprehensive framework leveraging Foundation Models to represent, analyze, and generate code changes across the software development lifecycle.

We begin with **Patcherizer**, a novel representation learning framework that unifies sequential and structural aspects of code changes. Unlike prior approaches that treat changes as either token diffs or tree transformations, Patcherizer introduces dual encoders: a SeqIntention module capturing semantic differences in token sequences, and a GraphIntention module representing structural modifications in ASTs. Through pre-training on 90K parsable Java patches, Patcherizer achieves state-of-the-art performance on commit message generation, defect prediction, and our novel task of code change intention detection—moving beyond syntactic operations to uncover semantic intent.

Building on this foundation, we develop specialized systems addressing critical software engineering challenges:

**LLMDA** tackles the security-critical problem of silent security patches—vulnerability fixes released without documentation, enabling n-day attacks. By augmenting patches with LLM-generated explanations, LLMDA bridges the gap between code semantics and missing commit context. Its PT-Former architecture aligns multi-modal inputs (code changes, descriptions, LLM explanations, and task instructions) while Stochastic Batch Contrastive Learning refines classification boundaries. LLMDA achieves 42% improvement over GraphSPD on PatchDB and SPI-DB benchmarks, effectively identifying undocumented vulnerabilities that traditional approaches miss.

**CodeAgent** revolutionizes code review through multi-agent collaboration, recognizing that review is inherently interactive rather than single-pass. Our framework simulates team dynamics with specialized agents for vulnerability detection, format consistency, and code revision. Critically, we introduce QA-Checker to combat prompt drifting—a fundamental challenge in multi-agent systems where conversations stray from objectives. CodeAgent achieves 41% improvement in vulnerability detection over GPT-4, 14% better recall in format inconsistency detection over ReAct, and 30% improvement in Edit Progress metrics for code revision.

**SynFix** addresses repository-level program repair through CallGraph-driven analysis, overcoming the limitations of agent-based approaches that suffer from cascading errors and exploration inefficiencies. By modeling structural dependencies across files, classes, and functions, SynFix ensures fixes propagate correctly through interconnected components. Its four-phase pipeline—CallGraph construction, hierarchical localization, cross-component synchronization, and paired patch validation—achieves state-of-the-art results on SWE-bench benchmarks while maintaining deterministic, cost-effective operation.

Finally, **Repairity** democratizes advanced repair capabilities by transferring reasoning from closed-source to open-source LLMs. Through a three-stage pipeline—extracting reasoning traces from teacher models (Claude-Sonnet3.7), supervised fine-tuning on filtered examples, and our novel Reinforcement Learning with LLM Feedback (RL-LF)—Repairity boosts Qwen2.5-Coder-32B performance by 24.5%, closing 98% of the capability gap with commercial models. This eliminates the accessibility, customizability, and privacy constraints of closed-source solutions.

Together, these contributions form a complete arc from representation to reasoning: learning unified representations (Patcherizer), detecting critical patterns (LLMDA), automating collaborative processes (CodeAgent), handling repository-scale complexity (SynFix), and democratizing capabilities (Repairity). Our extensive evaluations across multiple benchmarks demonstrate that foundation models, when properly architected with domain-specific innovations, can transform code change analysis from reactive maintenance to proactive, intelligent evolution—fundamentally advancing how we build and maintain software systems.



# Acknowledgements

First and foremost, I give thanks to God, who has guided and sustained me and my family through every season of this doctoral journey. His grace has been my strength in moments of doubt and my compass when the path seemed unclear. "Forgetting what is behind and straining toward what is ahead, I press on toward the goal to win the prize for which God has called me heavenward in Christ Jesus" (Philippians 3:13-14 NIV).

To my incredible wife and our precious children — words cannot express the depth of my gratitude for your unwavering love, infinite patience, and remarkable strength. You have been my anchor through late nights, missed weekends, and moments of frustration. Your sacrifices have not gone unnoticed, and this achievement belongs as much to you as it does to me. Thank you for believing in this dream even when it meant putting your own dreams on hold. You carried me when I couldn't carry myself.

I extend my deepest gratitude to my advisor, Prof. Tegawendé F. Bissyandé, whose mentorship has been transformative. Thank you for your patience when I stumbled, your wisdom when I was lost, and your constant belief in me even when I doubted myself. Your guidance extended far beyond academic pursuits — you taught me how to think critically, persevere through challenges, and maintain integrity in research. Your door was always open, and your encouragement never wavered.

To Prof. Jacques Klein, thank you for your early guidance and encouragement that set the foundation for my research journey. I still vividly remember learning draw.io from you — a seemingly small moment that exemplified your dedication to nurturing students at every level. Your insights and support during the crucial early stages of my PhD were invaluable.

I am profoundly grateful to the distinguished members of my thesis committee who generously gave their time and expertise to evaluate this work. To Dr. Jean-Rémy Falleri from Bordeaux INP, thank you for your insightful questions and constructive feedback that challenged me to think more deeply about the implications of my research. To Dr. Bach Le from the University of Melbourne, your expertise in program repair and thoughtful suggestions significantly strengthened this dissertation. To Dr. Kui Liu from Huawei, your industry perspective and technical insights bridged the gap between academic research and real-world application. To Dr. Iyiola Emmanuel Olatunji from the University of Luxembourg, your careful review and valuable comments helped refine the clarity and precision of this work. Each of you brought unique perspectives that enriched this dissertation immeasurably.

To all my amazing colleagues and labmates at the Interdisciplinary Centre for Security, Reliability and Trust (SnT) — thank you for creating an environment where collaboration thrived, ideas flourished, and friendships bloomed. The late-night debugging sessions, conference preparations, paper discussions, and coffee breaks shaped not just my research but also my understanding of what it means to be part of a scientific community. Your inspiration, feedback, and camaraderie made this journey not just bearable but enjoyable.

To my mother, whose prayers have covered me since childhood, thank you for instilling in me the values of perseverance and faith. To my siblings and extended family, your

constant support, encouragement, and presence — whether physical or virtual — provided the emotional foundation I needed to persist. To my friends near and far, thank you for your prayers, your check-ins, and for understanding when I disappeared into the depths of research for months at a time.

Finally, to myself — thank you for not giving up.

This PhD is not an ending but a beginning. With gratitude for all who have brought me to this moment, hope for what lies ahead, and determination to use this knowledge for the betterment of my family and society, I step forward into the next chapter.

Xunzhu Tang  
University of Luxembourg  
July 2025

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	2
1.2	Limitations and Challenges . . . . .	2
1.2.1	Limitations of Existing Approaches . . . . .	2
1.2.2	Primary Challenges . . . . .	3
1.3	Contributions . . . . .	4
1.4	Dissertation Roadmap . . . . .	5
<b>2</b>	<b>Background and Related Work</b>	<b>9</b>
2.1	Background . . . . .	10
2.1.1	Code Change Representation and Analysis . . . . .	10
2.1.2	Security Vulnerabilities and Silent Patches . . . . .	10
2.1.3	Code Review Processes and Automation . . . . .	11
2.1.4	Repository-Wide Program Repair . . . . .	11
2.1.5	Large Language Models for Code . . . . .	12
2.1.6	Dynamic Programming and Multi-Agent Systems . . . . .	13
2.2	Related Work . . . . .	13
2.2.1	Code Change Representation Learning . . . . .	13
2.2.2	Security Patch Detection Techniques . . . . .	14
2.2.3	Automated Code Review Approaches . . . . .	15
2.2.4	Repository-Wide Program Repair . . . . .	16
2.2.5	Performance Gap in Code LLMs . . . . .	17
2.2.6	Program Repair with LLMs . . . . .	18
<b>3</b>	<b>Learning to Represent Code Changes</b>	<b>19</b>
3.1	Introduction . . . . .	21
3.2	Patcherizer . . . . .	23
3.2.1	Handling Incomplete Code Snippets . . . . .	23
3.2.2	Code Change Preprocessing . . . . .	23
3.2.3	Sequence Intention Encoder . . . . .	24
3.2.3.1	Input Layer . . . . .	25
3.2.3.2	Transformer Embedding Layer . . . . .	25
3.2.3.3	SeqIntention Embedding Layer . . . . .	26
3.2.4	Graph Intention Encoder . . . . .	27
3.2.4.1	Graph Building . . . . .	28
3.2.5	Graph Intention Encoding Details . . . . .	29
3.2.5.1	Graph Learning . . . . .	29
3.2.5.2	Graph Intention Encoding . . . . .	30
3.2.6	Aggregator: Aggregating Multimodal Input Embeddings . . . . .	31

3.2.7	Pre-training . . . . .	31
3.2.8	Fine-tuning for Different Tasks . . . . .	31
3.2.8.1	Code Change Description Generation . . . . .	31
3.2.8.2	Code Change Correctness Assessment . . . . .	32
3.2.8.3	Code Change Intention Detection . . . . .	32
3.3	Experimental Design . . . . .	34
3.3.1	Implementation . . . . .	34
3.3.2	Research questions . . . . .	34
3.3.3	Baselines . . . . .	34
3.3.4	Datasets . . . . .	35
3.3.5	Metrics . . . . .	35
3.4	Experimental Results . . . . .	37
3.4.1	[RQ-1]: Performance of Patcherizer . . . . .	37
3.4.1.1	RQ-1.1: Code Change Description Generation . . . . .	37
3.4.1.2	RQ-1.2: Code Change Correctness Assessment . . . . .	39
3.4.1.3	RQ-1.3: Code Change Intention Detection . . . . .	40
3.4.1.4	RQ-1.4: Just-in-Time Defect Prediction . . . . .	41
3.4.2	[RQ-2]: Ablation Study . . . . .	42
3.4.2.1	Code Change Description Generation . . . . .	42
3.4.2.2	Code Change Correctness Assessment . . . . .	44
3.4.2.3	Code Change Intention Detection . . . . .	45
3.4.2.4	Just-in-Time Defect Prediction . . . . .	46
3.4.2.5	Ablation Study on Sequence Intention Encoder Components . . . . .	46
3.4.3	[RQ-3]: Generalizability and Robustness . . . . .	49
3.5	Discussion . . . . .	52
3.5.1	Comparison with Slice-Based Code Change Representation . . . . .	52
3.5.1.1	Methodological Differences . . . . .	52
3.5.1.2	Performance Comparison . . . . .	52
3.5.1.3	Novelty and Contributions . . . . .	52
3.5.2	Threats to Validity . . . . .	53
3.5.3	Limitations . . . . .	53
3.5.4	Handling Incomplete Code Information . . . . .	53
3.5.5	Extensibility . . . . .	54
3.5.5.1	Datasets . . . . .	54
3.5.5.2	Evaluation Metrics . . . . .	54
3.5.5.3	Experimental Results . . . . .	54
3.6	Related Work . . . . .	56
3.6.1	Code change Representation . . . . .	56
3.6.2	Applications of Code Change Embeddings . . . . .	56
3.7	Conclusion . . . . .	58
3.8	Conflict of Interest Statement . . . . .	58
3.9	Acknowledgments . . . . .	58
3.10	Ethical approval . . . . .	58
3.11	Informed consent . . . . .	58
3.12	Author Contributions . . . . .	58
3.13	Availability: . . . . .	59

<b>4 Just-in-Time Detection of Silent Security Patches</b>	<b>61</b>
4.1 Overview	63
4.2 Motivation	65
4.2.1 Silent Security Patches: A Critical Threat	66
4.2.2 Limitations of Existing Approaches	66
4.2.3 Opportunities with Large Language Models (LLMs)	67
4.2.4 Research Objective	67
4.3 The LLMDA approach	68
4.3.1 Data augmentation with LLMs	68
4.3.2 Generation of Bimodal Input Embeddings	68
4.3.3 PT-Former: Embeddings alignment and Concatenation	69
4.3.4 Stochastic Batch Contrastive Learning (SBCL)	71
4.3.5 Prediction and Training Layer for Security Patch Detection	73
4.4 Experimental Setup	74
4.4.1 Research Questions	74
4.4.2 Datasets	74
4.4.3 Evaluation Metrics	74
4.4.4 Baseline Methods	74
4.4.5 Implementation	75
4.5 Experiment Results	76
4.5.1 Overall performance of LLMDA	76
4.5.2 Contributions of key design decisions	77
4.5.2.1 <i>Impact of LLM-generated explanations</i>	77
4.5.2.2 <i>Impact of instruction</i>	77
4.5.2.3 <i>Impact of PT-Former</i>	78
4.5.2.4 <i>Impact of SBCL</i>	78
4.5.3 Discriminative power of LLMDA representations	79
4.5.3.1 <i>Visualization of embedding spaces</i>	79
4.5.3.2 Case studies	80
4.5.4 Robustness of LLMDA	82
4.5.5 Effect of Different Models and Different Instructions	83
4.5.6 <i>Extended Evaluation of Instruction</i>	85
4.6 Discussion	87
4.6.1 Threats to Validity	87
4.6.2 Limitation	87
4.6.3 Leveraging Patch Descriptions for Enhanced Cross-Attention	88
4.6.3.1 Experimental Results	89
4.6.4 Limitations of PT-Former: Dependency on Embedding Models	90
4.6.5 Comparison with CoLeFunDa	90
4.6.6 Model Selection for Initial Embeddings	91
4.6.7 Extended Case Studies on the Impact of Explanations and Instructions	91
4.6.7.1 <b>Case Study 1: Complex Code Changes with Minimal Documentation</b>	91
4.6.7.2 <b>Case Study 2: Misleading Commit Message</b>	92
4.6.7.3 <b>Analysis of Findings</b>	92
4.7 Conclusion	93

<b>5 CodeAgent: Autonomous Communicative Agents for Code Review</b>	<b>95</b>
5.1 Overview . . . . .	97
5.2 CodeAgent . . . . .	99
5.2.1 Tasks . . . . .	99
5.2.2 Pipeline . . . . .	100
5.2.3 Role Card Definition . . . . .	101
5.2.4 Self-Improving CoT with QA Checker . . . . .	101
5.3 Experimental Setup . . . . .	103
5.3.1 Datasets . . . . .	103
5.3.2 Metrics . . . . .	103
5.3.3 State-of-the-Art Tools and Models . . . . .	103
5.4 Experimental Result Analysis . . . . .	105
5.4.1 Vulnerability Analysis . . . . .	105
5.4.2 Consistency and Format Detection . . . . .	106
5.4.3 Code Revision . . . . .	107
5.5 Conclusion . . . . .	108
5.6 Details of QA-Checker Algorithm . . . . .	109
5.7 Complete Related Work . . . . .	111
5.8 Experimental Details . . . . .	112
5.8.1 Role Definition . . . . .	112
5.8.2 Execute Time Across Languages . . . . .	113
5.9 Comparative Analysis of QA-Checker AI System and Recursive Self-Improvement Systems . . . . .	114
5.9.1 Comparison Table . . . . .	114
5.9.2 Differences and Implications . . . . .	114
5.9.3 Importance of QA-Checker in Role Conversations . . . . .	115
5.9.4 Conclusion . . . . .	115
5.10 Capabilities Analysis between CodeAgent and Other Methods . . . . .	115
5.11 Dataset . . . . .	115
5.12 Key Factors Leading to Vulnerabilities . . . . .	117
5.13 Data Leakage Statement . . . . .	117
5.14 Algorithmic Description of CodeAgent Pipeline with QA-Checker . . . . .	117
5.15 Detailed Performance of CodeAgent in Various Languages on VA task . . . . .	119
5.16 More detailed experimental results on CA and FA tasks . . . . .	120
5.17 Case Study . . . . .	120
5.17.1 Performance on 9 languages . . . . .	121
5.17.2 Difference of CodeAgent-3.5 and CodeAgent-4.0 . . . . .	122
5.18 Ablation study . . . . .	122
5.19 Cost statement . . . . .	125
5.20 Tool . . . . .	128
<b>6 SynFix: Synchronous Repair for Codebase via CallGraph</b>	<b>135</b>
6.1 Overview . . . . .	137
6.2 Methodology . . . . .	139
6.2.1 CallGraph Building . . . . .	139
6.2.2 Localization . . . . .	140
6.2.3 Synchrony . . . . .	141
6.2.4 Paired Patches Validation . . . . .	142

6.2.5	Datasets . . . . .	144
6.2.6	Implementation . . . . .	144
6.2.7	Baselines . . . . .	144
6.2.8	Research Questions . . . . .	144
6.3	Experimental Results . . . . .	145
6.3.1	RQ1: Overall Performance . . . . .	145
6.3.2	RQ2: Ablation Study . . . . .	146
6.4	Conclusion . . . . .	148
6.5	Efficiency Considerations with GPT-4o . . . . .	149
6.6	Matching Prompt for Using Chatgpt 3.5 . . . . .	149
6.7	Analysis Prompt for Using Chatgpt 4o . . . . .	149
6.8	Results on SWE-bench Full . . . . .	149
6.9	Results on SWE-bench Verified . . . . .	150
6.9.1	External Experiment Across Foundation Models . . . . .	150
6.10	CallGraph Driven Error Detection and Synchronous Repair . . . . .	151
<b>7</b>	<b>Boosting Open-Source LLMs for Program Repair via Reasoning Transfer and LLM-Guided Reinforcement Learning</b>	<b>155</b>
7.1	Overview . . . . .	157
7.2	Knowledge Transfer in LLMs . . . . .	159
7.2.1	Knowledge Distillation . . . . .	159
7.2.2	Learning from Demonstrations . . . . .	159
7.2.3	Reinforcement Learning from AI Feedback . . . . .	160
7.3	The REPAIRITY Approach . . . . .	161
7.3.1	Problem Formulation . . . . .	161
7.3.2	Step 1: Data Collection for SFT . . . . .	161
7.3.2.1	Reasoning Trace Extraction . . . . .	162
7.3.2.2	Verification and Filtering . . . . .	162
7.3.3	Step 2: Reasoning Trace Learning . . . . .	162
7.3.3.1	Training Objective . . . . .	163
7.3.3.2	Fine-tuning Implementation . . . . .	163
7.3.3.3	Ablation: Direct Output Fine-tuning . . . . .	163
7.3.4	Step 3: Reinforcement Learning with LLM Feedback (RL-LF) . . . . .	163
7.3.4.1	RL-LF Framework . . . . .	163
7.3.4.2	Reward Data Collection . . . . .	163
7.3.4.3	Reward Model Training . . . . .	164
7.3.4.4	Policy Optimization with PPO . . . . .	164
7.3.4.5	Value Function and Advantage Estimation . . . . .	165
7.3.5	Novelty of RL-LF . . . . .	165
7.3.6	Implementation Details . . . . .	166
7.4	Experimental Setup . . . . .	167
7.4.1	Benchmarks . . . . .	167
7.4.2	Evaluation Metrics . . . . .	167
7.4.3	Models . . . . .	168
7.4.4	Experimental Procedure . . . . .	168
7.4.5	Computational Resources . . . . .	168
7.5	Experimental Results . . . . .	170
7.5.1	Performance Comparison . . . . .	170

7.5.1.1	BigCodeBench . . . . .	170
7.5.1.2	SWE-bench Verified . . . . .	170
7.5.1.3	MBPP . . . . .	171
7.5.1.4	Defects4J . . . . .	171
7.5.2	Ablation Study . . . . .	171
7.5.2.1	Component Contributions . . . . .	172
7.5.2.2	Benchmark-Specific Patterns . . . . .	172
7.5.3	Case Study . . . . .	173
7.5.3.1	Structured Data Extraction . . . . .	175
7.5.3.2	Type Coercion Bug Fix . . . . .	176
7.5.4	Generalization Capability . . . . .	176
7.5.5	Summary of Results . . . . .	176
7.6	Discussion . . . . .	178
7.6.1	Analysis of Key Findings . . . . .	178
7.6.1.1	Effectiveness of Reasoning Transfer . . . . .	178
7.6.1.2	Synergy Between RT Learning and RL-LF . . . . .	178
7.6.2	Limitations and Future Work . . . . .	178
7.6.3	Implications for Software Engineering . . . . .	179
7.6.4	Ethical Considerations . . . . .	179
7.7	Related Work . . . . .	180
7.7.1	Program Repair with LLMs . . . . .	180
7.7.2	Knowledge Transfer Between LMs . . . . .	180
7.7.3	Reasoning Enhancement in LLMs . . . . .	180
7.7.4	Reinforcement Learning for Model Alignment . . . . .	180
7.8	Conclusion . . . . .	181
<b>8</b>	<b>Conclusion</b>	<b>183</b>
<b>9</b>	<b>Future Work</b>	<b>185</b>
9.1	Efficient Reasoning Strategies . . . . .	186
9.1.1	MoE Optimization for Faster Inference . . . . .	186
9.1.2	Draft-Based Reasoning with RL Selection . . . . .	186
9.2	Automatic Coding Advancements . . . . .	187
9.2.1	Comprehensive Benchmark for Coding from Scratch . . . . .	187
9.2.2	Structural Routing for Coding and Debugging . . . . .	188
9.3	Integration of Future Directions . . . . .	188

# List of Figures

1.1	Dissertation roadmap showing the progression from foundational representation (Patcherizer) through specialized applications (LLMDA, CodeAgent, SynFix) to capability democratization (Repairity). Arrows indicate knowledge flow and dependencies between contributions. . . . .	7
3.1	Overview of Patcherizer. . . . .	24
3.2	Example illustrating how semantically distinct code changes can result in similar AST diffs. Both code changes involve replacing the condition in an “if” statement, and their corresponding AST diffs may appear structurally similar. However, their <i>intentions</i> differ: the left change disables functionality, while the right one introduces a refactor with encapsulated logic. Traditional AST-based models may treat these equivalently, while Patcherizer captures this distinction through its sequence and graph intention encoders. . . . .	25
3.3	Architecture for the Sequence Intention Encoder. . . . .	25
3.4	cross-resnet architecture. . . . .	27
3.5	Architecture for the Graph Intention Encoder. . . . .	27
3.6	An example of merging graphs. . . . .	28
3.7	graph-cross-resnet architecture. . . . .	30
3.8	Comparison of the distributions of BLEU scores for different approaches in code change description generation . . . . .	38
3.9	Illustrative example of code change description generation . . . . .	40
3.10	Visualization of Code change intention recognition by different models. . . . .	41
3.11	Case analysis of the ablation study . . . . .	43
3.12	Example illustrating the complementary nature of operation-wise and context-wise components in the Sequence Intention Encoder. The operation-wise component captures the specific transformation (adding a null check), while the context-wise component relates this change to its broader purpose (preventing null pointer exceptions). . . . .	48
3.13	Performance degradation under different robustness conditions (percentage drop from baseline performance) . . . . .	51
4.1	Overview of LLMDA . . . . .	66
4.2	Architecture of PT-Former. . . . .	70
4.3	Overview of our SBCL layer. . . . .	72
4.4	Illustration of embedding subspaces of security/non-security patches for contrastive learning . . . . .	79
4.5	Embeddings yielded by GraphSPD . . . . .	80
4.6	Embeddings yielded by TwinRNN . . . . .	80
4.7	Embeddings yielded by Vulfixminer . . . . .	80

4.8	Embeddings yielded by CodeT5 . . . . .	80
4.9	Embeddings yielded by LEOPARD . . . . .	80
4.10	PCA visualizations of security and non-security patch embeddings by baselines and LEOPARD. . . . .	80
4.11	Cross-Attention Between Patch and Description version of PT-Former . .	89
5.1	<b>A Schematic diagram of role data cards of simulated code review team and their conversations within CodeAgent.</b> We have six characters in CodeAgent across four phases, including “Basic Info Sync”, “Code Review”, “Code Alignment”, and “Document”. Code review is a kind of collaboration work, where we design conversations between every two roles for every step to complete the task. . . . .	99
5.2	CodeAgent’s pipeline/scenario of a full conversation during the code review process among different roles. “Basic Info Sync” demonstrates the basic information confirmation by the CEO, CTO, and Coder; “Code Review” shows the actual code review process; “Code Alignment” illustrates the potential code revision; and “Document” represents the summarizing and writing conclusion for all the stakeholders. All the conversations are being ensured by the Quality Assurance checker until they reach the maximum dialogue turns or meet all the requirements. . . . .	100
5.3	This diagram shows the architecture of our designed Chain-of-Thought (CoT): Question-Answer Checker (QA-Checker). . . . .	101
5.4	Overlap of vulnerability detection by CodeBERT, GPT-3.5, GPT-4.0, and CodeAgent. . . . .	106
5.5	Specialization of six main characters in CodeAgent. . . . .	112
5.6	Execution time with CodeAgent across different language (count unit: second).114	
5.7	Distribution of positive, negative of both merged and closed data across 9 languages, including ‘python’, ‘java’, ‘go’, ‘c++’, ‘javascript’, ‘c’, ‘c#’, ‘php’, ‘ruby’ . . . . .	116
5.8	Comparative Visualization of Merged and Closed Commit Counts Across Various Programming Languages . . . . .	116
5.9	Comparison of models on the <b>merged</b> data across 9 languages on <b>CA task</b> . 120	
5.10	Comparison of models on the <b>closed</b> data across 9 languages on <b>CA task</b> . . 121	
5.11	Comparison of models on the <b>merged</b> data across 9 languages on <b>FA task</b> . 121	
5.12	Comparison of models on the <b>closed</b> data across 9 languages on <b>FA task</b> . . 122	
5.13	Example in Python project with sha value: “0d3b94bbd7...” . . . . .	123
5.14	Example in Java project with sha value: “6e3c6d17d9...” . . . . .	124
5.15	Example in Go project with sha value: “a6b00b09...” . . . . .	125
5.16	Example in C++ project with sha value: “09e88c7c2...” . . . . .	126
5.17	Example in JavaScript project with sha value: “6e9a80d93...” . . . . .	127
5.18	Example in C project with sha value: “5f93cf43d...” . . . . .	128
5.19	Example in C# project with sha value: “0e231c7a...” . . . . .	129
5.20	Example in PHP project with sha value: “6679f059b...” . . . . .	130
5.21	Example in Ruby project with sha value: “584f72e7f4...” . . . . .	131
5.22	Comparison of code review between CodeAgent-3.5 and CodeAgent-4.0 on one Java project with sha value: “6e3c6d17...” . . . . .	132
5.23	website of CodeAgent . . . . .	133
6.1	Pipeline of CallGraph building. . . . .	140

6.2	Pipeline of PANTHER	140
6.3	Ablation Study: Impact of removing dynamic analysis and CallGraph dependencies on SWE-bench Lite.	147
7.1	Overview of the Repairity	161



# List of Tables

3.1	Performance Results of code change description generation. . . . .	38
3.2	Scoring Criteria Dong et al. [2022] . . . . .	38
3.3	Human Evaluation Results . . . . .	39
3.4	Performance of Code Change Correctness Assessment . . . . .	40
3.5	AUC (%) Results on JIT defect prediction on QT and OPENSTACK datasets. . . . .	42
3.6	Ablation study results based on the code change description generation task. . . . .	43
3.7	Performance of Code Change Correctness Assessment . . . . .	44
3.8	Intention Classification Performance of Different Patcherizer Variants . . . . .	45
3.9	AUC Scores for JIT Defect Prediction with Different Patcherizer Variants . . . . .	46
3.10	Ablation study of Sequence Intention Encoder components on code change description generation. . . . .	47
3.11	Cross-task generalizability performance on independent datasets . . . . .	50
3.12	Comprehensive robustness analysis across tasks (performance in %) . . . . .	50
3.13	AUC (%) Results on JIT defect prediction on QT and OPENSTACK datasets	52
3.14	Performance metrics (%) on security code change detection . . . . .	55
4.1	Performance metrics (%) on security patch detection . . . . .	76
4.2	Performance (%) of LLMDA <sub>EX-</sub> (without the LLM-generated explanations) . . . . .	77
4.3	Performance (%) of LLMDA <sub>IN-</sub> (without the designed instruction) . . . . .	78
4.4	Performance (%) of LLMDA <sub>PT-</sub> (without the <i>PT-Former</i> module) . . . . .	78
4.5	Performance (%) of LLMDA <sub>SBCL-</sub> (without contrastive learning) . . . . .	79
4.6	Attention scores for security and non-security labels by GraphSPD and LLMDA on two sample patches . . . . .	81
4.7	Performance (%) of GraphSPD, LLMDA, and additional baselines against unseen patches from the FFmpeg dataset Zhou et al. [2019]. Numbers between parentheses (↓X) correspond to the drop of performance when compared to the evaluation in cross-validation with PatchDB. . . . .	82
4.8	Performance metrics (%) on security patch detection with different instructions . . . . .	83
4.9	Comparison of Instruction Types for Explanation Generation . . . . .	84
4.10	Performance (%) with Different Instruction Positions . . . . .	86
4.11	Impact of Instruction . . . . .	86
4.12	Performance (%) with Enhanced Cross-Attention . . . . .	89
4.13	Performance (%) and Training Efficiency Comparison: Original LLMDA vs. CodeLLaMA-13B . . . . .	91
5.1	Comparison of Positive and Negative Samples in CA and FA (CA and FA are defined in Section 5.2.1). . . . .	103

5.2	The number of vulnerabilities found by CodeAgent and other approaches. As described in Appendix-Section 5.11, we have 3,545 items to evaluate. Rate <sub>cr</sub> represents the confirmed number divided by the number of findings while Rate <sub>ca</sub> is the confirmed number divided by the total evaluated number. CodeAgent <sub>w/o</sub> indicates the version without QA-Checker. . . . .	104
5.3	Comparison of CodeAgent with other methods on merged and closed commits across 9 languages on <b>CA task</b> . ‘Imp’ represents the improvement. . . . .	107
5.4	Comparison of CodeAgent with other methods on merged and closed commits across the 9 languages on <b>FA task</b> . ‘Imp’ represents the improvement. . . . .	107
5.5	Experimental Results for the Code Revision ( <b>CR task</b> ) of CodeAgent and the state-of-the-art works. Bold indicates the best performers. . . . .	108
5.6	Comparative Overview of QA-Checker AI System and Recursive Self-Improvement Systems . . . . .	114
5.7	Comparison of capabilities for CodeAgent and other approaches. ‘✓’ indicates the presence of a specific feature in the corresponding framework, ‘✗’ is absence. ChatDev and MetaGPT are two representative multi-agent frameworks, GPT is a kind of single-agent framework, and CodeBert is a representative pre-trained model. . . . .	115
5.8	Statistics of Studied Datasets. . . . .	117
5.9	Vulnerable problems (#) found by CodeAgent. Rate <sub>merge</sub> means the value of confirmed divided by the total number in the merged and Rate <sub>close</sub> is the value of confirmed divided by the total number in the closed. Rate <sub>avg</sub> is the value of the confirmed number divided by the total number of the merged and closed. . . . .	120
5.10	Correlation Table between specific programming language and case study. . . . .	121
5.11	Vulnerable problems (#) found by CodeAgent <sub>w/o</sub> . . . . .	126
5.12	Summarizes the average query time and cost per code review for CodeAgent-3.5 and CodeAgent-4. . . . .	128
6.1	Results on SWE-bench Lite. . . . .	145
6.2	Results on SWE-bench Full. . . . .	150
6.3	Results on SWE-bench Verified. . . . .	151
6.4	Comparison of foundation models on SWE-bench Lite. . . . .	151
7.1	Overview of Benchmark Tasks and Dataset Statistics . . . . .	167
7.2	BigCodeBench Pass@1 Results . . . . .	170
7.3	SWE-bench Verified Results . . . . .	170
7.4	MBPP Accuracy Results . . . . .	171
7.5	Defects4J Results . . . . .	171
7.6	Ablation studies on different benchmarks: BigCodeBench (Pass@1), SWE-bench Verified (Resolved %), MBPP (Accuracy), and Defects4J (Compilation Rate, Pass@ 1, and BLEU. ‘base’ represents ‘Qwen2.5-Coder-32B-Instruct’) . . . . .	172

# 1 Introduction

*In this chapter, we introduce the critical role of code changes in software evolution and the challenges of their effective representation and analysis. We then present our comprehensive framework leveraging Foundation Models to transform how we understand, analyze, and generate code changes, followed by the roadmap of this dissertation.*

## Contents

---

<b>1.1</b>	<b>Motivation</b>	<b>2</b>
<b>1.2</b>	<b>Limitations and Challenges</b>	<b>2</b>
1.2.1	Limitations of Existing Approaches	2
1.2.2	Primary Challenges	3
<b>1.3</b>	<b>Contributions</b>	<b>4</b>
<b>1.4</b>	<b>Dissertation Roadmap</b>	<b>5</b>

---

## 1.1 Motivation

Code changes are the lifeblood of software evolution. Every bug fix, feature addition, security patch, and performance optimization manifests through code changes—the fundamental transformations that drive software from one state to another. With software maintenance consuming up to 70% of total development costs and poorly managed changes being a primary source of critical failures, the ability to effectively represent, analyze, and generate code changes has become paramount for modern software engineering.

Consider the cascading impact of code change mismanagement: Microsoft’s 2021 Azure outage, triggered by a seemingly minor code change, affected millions of users globally and resulted in substantial financial losses. Similarly, the 2024 CrowdStrike incident, where a faulty update caused widespread system failures, demonstrates how a single misunderstood change can paralyze critical infrastructure. These incidents underscore a fundamental challenge: as software systems grow exponentially in size and complexity, traditional approaches to code change analysis—treating them as simple textual diffs or unstructured token sequences—are no longer sufficient.

The challenge extends beyond preventing failures. In cybersecurity, approximately 96% of modern applications incorporate open-source components, yet critical security patches often go unnoticed when released "silently" without proper documentation. These silent patches create windows of vulnerability where attackers can exploit known fixes that haven’t been deployed—so-called n-day attacks. In code review, the manual process remains costly and error-prone, with reviewers potentially overlooking semantic errors and contextual issues, especially in multi-stakeholder collaborations. In automated program repair, current approaches focus on isolated functions while missing repository-wide dependencies, leading to patches that fix one issue but introduce regressions elsewhere.

The emergence of Large Language Models (LLMs) presents both unprecedented opportunities and new challenges. While closed-source models like GPT-4 and Claude demonstrate remarkable code understanding capabilities, they remain inaccessible for many organizations due to privacy concerns, customization limitations, and deployment constraints. Meanwhile, open-source alternatives lag significantly in performance, creating a capability gap that limits widespread adoption of AI-powered code analysis tools.

This dissertation addresses these fundamental challenges through a comprehensive framework that leverages Foundation Models to represent, analyze, and generate code changes. We progress from developing unified representations that capture both structural and semantic aspects of changes, through specialized systems for security detection and collaborative review, to repository-scale repair and ultimately to democratizing these capabilities through knowledge transfer from closed to open-source models.

## 1.2 Limitations and Challenges

### 1.2.1 Limitations of Existing Approaches

**Fragmented Representation Paradigms.** Current approaches to code change representation fall into three disconnected categories, each with critical limitations. AST-based approaches like Fira capture fine-grained structural edits but remain sensitive to syntax-level noise and formatting changes. Embedding-based approaches such as CC2Vec and CCBERT learn latent representations but are trained on limited supervision signals and lose structural integrity including control and data flow. LLM-based approaches leveraging models like CodeBERT and CodeT5+ benefit from large-scale pretraining but produce brittle outputs without task-specific guidance. No existing method successfully captures

the intent, structure, and semantics of code changes in a unified and generalizable manner.

**Inadequate Security Patch Detection.** The detection of silent security patches—fixes released without CVE assignments or security advisories—remains critically limited. Static analysis tools rely on predetermined patterns that fail to interpret the conceptual intentions behind changes, generating excessive false positives. Graph-based models like GraphSPD focus on localized code segments without understanding broader system interactions. Most critically, these approaches cannot compensate for missing or misleading commit messages, which are common in security-sensitive changes where developers may intentionally obscure vulnerability details.

**Single-Pass Review Automation.** Current code review automation treats this inherently collaborative process as a single-pass transformation problem. Systems focus primarily on rewriting submitted code while ignoring critical aspects like vulnerability detection, format consistency checking, and multi-stakeholder perspectives. This simplification fails to capture the iterative refinement and diverse expertise required in professional code reviews, where security experts, domain specialists, and coding standard enforcers must coordinate their feedback.

**Local-Scope Program Repair.** Despite advances in LLM-based program repair, existing approaches remain confined to fixing isolated functions or classes. They lack understanding of complex interdependencies across files, classes, and modules that characterize real-world codebases. Agent-based systems attempt to address this through iterative exploration but introduce new problems: cascading errors from early incorrect decisions, inefficient exploration strategies, and inability to filter irrelevant feedback. These limitations result in patches that may pass local tests but introduce subtle regressions in dependent components.

**Inaccessible Advanced Capabilities.** A significant performance chasm exists between closed-source LLMs (GPT-4, Claude) and open-source alternatives (Llama, Qwen, Mistral) in code-related tasks. While closed models excel through superior reasoning capabilities—systematically analyzing code, identifying issues, and formulating solutions—they present insurmountable barriers for organizations with strict privacy requirements, customization needs, or resource constraints. Previous attempts to bridge this gap through instruction tuning or RLHF require extensive human annotation, limiting scalability and failing to transfer the core reasoning processes that distinguish advanced models.

## 1.2.2 Primary Challenges

**Dual-Nature Representation Challenge.** Code changes possess a fundamental duality: they are both sequential transformations (how tokens change) and structural modifications (how AST nodes transform). Capturing both aspects simultaneously while preserving their interdependencies requires novel architectural designs that can model sequence intentions and graph intentions in a unified framework.

**Semantic Gap Challenge.** The gap between what code changes do syntactically and what they intend semantically remains vast. A boundary check addition might represent a critical security fix, a defensive programming practice, or a performance optimization. Bridging this gap requires not just analyzing the change itself but understanding its context, purpose, and potential impact across the system.

**Scale and Dependency Challenge.** Modern software repositories contain millions of lines of code with complex webs of dependencies. Changes rarely exist in isolation—they ripple through call graphs, inheritance hierarchies, and module boundaries. Modeling these repository-wide implications requires representations that can efficiently capture both local modifications and global ramifications without becoming computationally intractable.

**Multi-Agent Coordination Challenge.** Collaborative software engineering tasks involve multiple perspectives and expertise areas that must be coordinated effectively. In multi-agent systems, maintaining focus and preventing "prompt drifting"—where conversations diverge from objectives—while preserving the benefits of diverse viewpoints presents a fundamental challenge in system design.

**Knowledge Transfer Challenge.** The reasoning capabilities that make closed-source LLMs effective—their ability to systematically decompose problems, identify patterns, and synthesize solutions—are encoded implicitly in their parameters. Extracting and transferring these capabilities to open-source models without access to training data or architectural details requires novel approaches that can capture not just outputs but underlying reasoning processes.

## 1.3 Contributions

This dissertation makes five interconnected contributions that collectively advance the state-of-the-art in code change representation and analysis:

**① Patcherizer: Unified Foundation for Code Change Representation.** We introduce Patcherizer, a novel dual-encoding framework that captures both sequential and structural aspects of code changes through SeqIntention and GraphIntention modules. Unlike previous approaches that treat changes as either token sequences or tree transformations, Patcherizer explicitly models the intention behind changes at both levels. Through pre-training on 90K parsable Java patches, Patcherizer achieves state-of-the-art performance across multiple tasks: improving BLEU by 19.39% in commit message generation, achieving 0.89 F1 in defect prediction, and introducing the novel task of code change intention detection that moves beyond syntactic operations to semantic understanding.

*This work is under minor review at Empirical Software Engineering (EMSE 2025).*

**② LLMDA: LLM-Augmented Detection for Silent Security Patches.** We present LLMDA, which addresses the critical challenge of identifying undocumented security fixes through multi-modal learning. LLMDA augments patches with LLM-generated semantic explanations, employs PT-Former to align code and text modalities, and introduces Stochastic Batch Contrastive Learning to refine classification boundaries. This approach achieves 42% improvement over GraphSPD on PatchDB and SPI-DB benchmarks, effectively identifying silent security patches that traditional approaches miss, thereby preventing n-day attacks and reducing vulnerability windows.

*Published in ACM Transactions on Software Engineering and Methodology (TOSEM 2025).*

**③ CodeAgent: Multi-Agent Collaborative Code Review.** We introduce CodeAgent, a multi-agent framework that transforms code review from single-pass transformation to collaborative refinement. Specialized agents handle vulnerability detection, format consistency, and code revision while our novel QA-Checker component prevents prompt drifting—maintaining conversation focus despite LLM stochasticity. CodeAgent achieves 41% improvement in vulnerability detection over GPT-4, 14% better recall in format inconsistency detection, and 30% improvement in Edit Progress metrics, demonstrating the power of coordinated multi-perspective analysis.

*Published at the Conference on Empirical Methods in Natural Language Processing (EMNLP 2024).*

④ **SynFix: Repository-Scale Program Repair via Dependency Modeling.** We present SynFix, a CallGraph-driven framework that explicitly models cross-file, cross-class, and cross-function dependencies for repository-wide program repair. Through its four-phase pipeline—CallGraph construction, hierarchical localization, cross-component synchronization, and paired patch validation—SynFix ensures fixes maintain consistency across interconnected components. On SWE-bench benchmarks, SynFix achieves state-of-the-art results: resolving 52.33% of SWE-bench-lite, 55.8% of SWE-bench-verified, and 29.86% of SWE-bench-full issues, significantly outperforming agent-based approaches while maintaining deterministic, cost-effective operation.

*Published in Findings of the Association for Computational Linguistics (ACL 2025).*

⑤ **Repairity: Democratizing Advanced Repair Capabilities.** We introduce Repairity, a three-stage methodology that transfers reasoning capabilities from closed to open-source LLMs. Through systematic extraction of reasoning traces, supervised fine-tuning with filtered examples, and our novel Reinforcement Learning with LLM Feedback (RL-LF), Repairity boosts Qwen2.5-Coder-32B performance by 24.5%, closing 98% of the capability gap with Claude-Sonnet3.7. This breakthrough eliminates barriers to advanced code analysis tools for organizations with privacy or customization requirements.

*Under review at ACM Transactions on Software Engineering and Methodology (TOSEM 2025).*

## 1.4 Dissertation Roadmap

This dissertation presents a comprehensive journey from foundational representation to advanced reasoning in code change analysis. Figure 1.1 illustrates the interconnected nature of our contributions and their progression through increasing levels of sophistication.

**Chapter 2: Background and Related Work** establishes the theoretical and practical foundations, surveying existing approaches in code change representation, security patch detection, code review automation, and program repair. We identify key limitations that motivate our research direction.

**Chapter 3: Patcherizer - Learning to Represent** introduces our foundational contribution: a unified representation framework that captures both sequential and structural aspects of code changes. Patcherizer establishes the representational foundation upon which our subsequent specialized systems build, demonstrating that effective dual-encoding can significantly improve performance across diverse downstream tasks.

The dissertation then branches into three parallel tracks, each addressing critical real-world challenges:

**Chapter 4: LLMDA - Learning to Detect** specializes representation for security-critical applications. Building on Patcherizer’s foundation, LLMDA augments representations with LLM-generated explanations and multi-modal alignment to identify silent security patches that threaten system integrity.

**Chapter 5: CodeAgent - Learning to Collaborate** extends beyond individual analysis to multi-agent collaboration. CodeAgent demonstrates how specialized agents can work together to provide comprehensive code reviews, introducing mechanisms to maintain focus and prevent conversation drift in complex multi-stakeholder scenarios.

**Chapter 6: SynFix - Learning to Repair** addresses repository-scale challenges through dependency-aware program repair. SynFix’s CallGraph-driven approach ensures that fixes respect complex interdependencies, moving beyond local repairs to repository-wide con-

sistency.

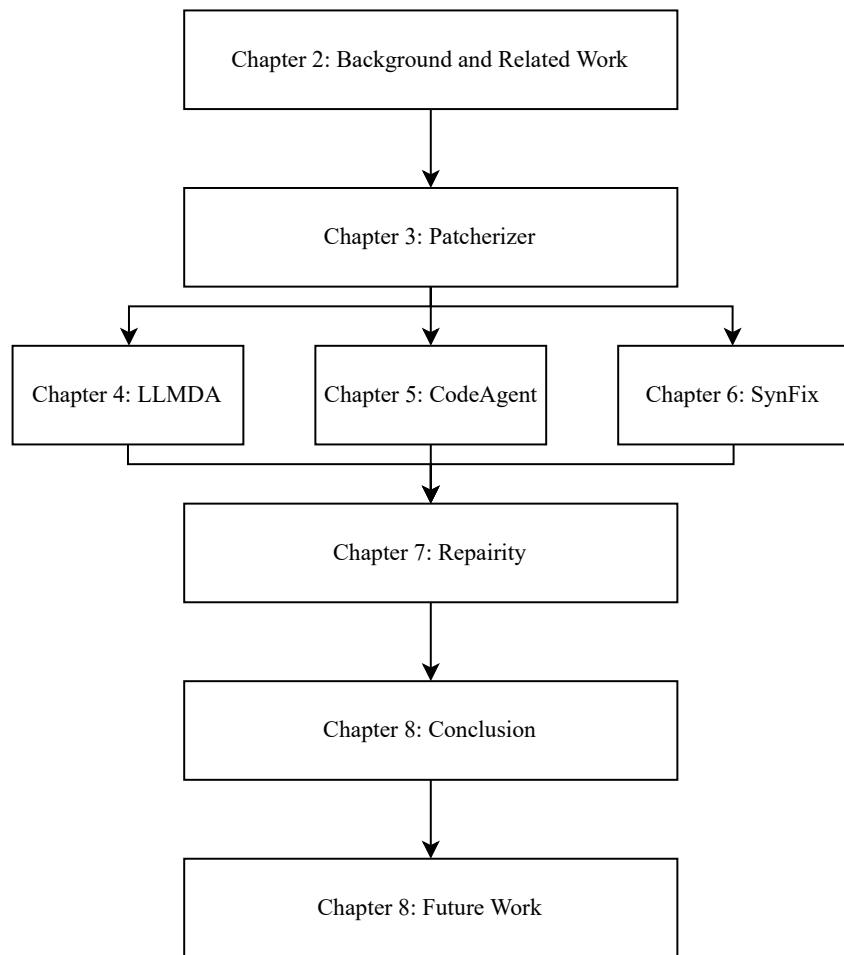
These three tracks converge in:

**Chapter 7: Repairity - Learning to Teach** synthesizes insights from our specialized systems to address the fundamental challenge of capability transfer. Repairity demonstrates how advanced reasoning from closed-source models can be distilled into open-source alternatives, democratizing access to sophisticated code analysis capabilities.

**Chapter 8: Conclusions** reflects on our contributions' collective impact, discussing how our progression from representation to reasoning establishes new paradigms for automated software evolution.

**Chapter 9: Future Work** outlines promising research directions, including cross-language generalization, real-time adaptation mechanisms, and integration with continuous integration/deployment pipelines.

This roadmap reflects our core thesis: that effective code change analysis requires not just better representations but a complete ecosystem of specialized yet interconnected systems. Each contribution addresses specific limitations while contributing to a unified vision where code changes are not merely processed but truly understood—their intent decoded, their impact assessed, and their implications managed across entire software systems.



**Figure 1.1:** Dissertation roadmap showing the progression from foundational representation (Patcherizer) through specialized applications (LLMDA, CodeAgent, SynFix) to capability democratization (Repairity). Arrows indicate knowledge flow and dependencies between contributions.



## 2 Background and Related Work

*This chapter introduces the necessary concepts and related works for understanding the objective, techniques, contributions and key concerns of the research studies that we have conducted in this dissertation. We begin by providing background on code change representation, security patch detection, code review automation, repository-wide program repair, and large language models for code understanding. We then review related work in these areas, examining existing approaches, their strengths, and limitations. This comprehensive overview establishes the foundation for our research contributions in the following chapters.*

### Contents

---

<b>2.1</b>	<b>Background</b>	<b>10</b>
2.1.1	Code Change Representation and Analysis	10
2.1.2	Security Vulnerabilities and Silent Patches	10
2.1.3	Code Review Processes and Automation	11
2.1.4	Repository-Wide Program Repair	11
2.1.5	Large Language Models for Code	12
2.1.6	Dynamic Programming and Multi-Agent Systems	13
<b>2.2</b>	<b>Related Work</b>	<b>13</b>
2.2.1	Code Change Representation Learning	13
2.2.2	Security Patch Detection Techniques	14
2.2.3	Automated Code Review Approaches	15
2.2.4	Repository-Wide Program Repair	16
2.2.5	Performance Gap in Code LLMs	17
2.2.6	Program Repair with LLMs	18

---

## 2.1 Background

This section introduces the fundamental concepts that form the basis of our research, providing the necessary context for understanding our contributions.

### 2.1.1 Code Change Representation and Analysis

Code changes represent the fundamental units of software evolution, serving as the mechanism through which software systems grow and adapt over time. These changes typically manifest as commits in version control systems, patches for specific issues, or pull requests in collaborative development environments. Formally, a code change can be defined as a transformation from one program state  $P$  to another program state  $P'$ , where the transformation includes additions, deletions, and modifications to the codebase.

The representation of code changes is critical for a variety of software engineering tasks. Unlike static code analysis, which examines a single snapshot of code, change analysis must capture both the structural and semantic aspects of the transformation between states. This representation challenge is compounded by the diverse nature of changes, which can range from simple bug fixes to complex feature additions or architectural refactorings.

Several formal models have emerged for representing code changes. At the most basic level, a change can be represented as a textual diff, which captures line-by-line additions and deletions. More sophisticated representations model changes at the abstract syntax tree (AST) level, capturing structural modifications to the code. Graph-based representations extend this further by modeling not only the structural changes but also the dependencies and relationships between different code elements.

The analysis of code changes serves multiple purposes in software engineering. It enables the detection of bugs and security vulnerabilities introduced by changes, facilitates code review by highlighting the most important aspects of a change, supports the generation of documentation and commit messages, and underpins automated program repair by identifying and fixing problematic changes. The effectiveness of these tasks depends heavily on the quality and comprehensiveness of the underlying change representation.

### 2.1.2 Security Vulnerabilities and Silent Patches

Security vulnerabilities represent a significant threat to software systems, potentially exposing sensitive data, allowing unauthorized access, or enabling system compromise. According to industry research, approximately 96% of software applications incorporate at least one open-source component, making the security of these components a critical concern for the entire software ecosystem.

Silent security patches present a particular challenge in this context. These are security fixes implemented without explicit notification or documentation of their security implications. Unlike vulnerabilities that receive Common Vulnerabilities and Exposures (CVE) identifiers and public advisories, silent patches may be described simply as “bug fixes” or “performance improvements” in commit messages, obscuring their true security significance.

The prevalence of silent patches is concerning for several reasons. First, they prevent downstream users from properly assessing and prioritizing updates, potentially leaving systems vulnerable to known exploits. Second, they complicate vulnerability management processes, making it difficult to track which vulnerabilities have been addressed in a particular version of software. Third, they contribute to the “n-day” vulnerability problem, where exploits are developed for publicly available patches before users have had time to update their systems.

The identification of silent security patches requires sophisticated analysis techniques that can distinguish security-relevant changes from regular maintenance updates. This analysis must consider not only the syntactic structure of the code changes but also their semantic implications for system security. Furthermore, the analysis must be able to operate effectively even in the absence of detailed commit messages or documentation, relying instead on patterns and characteristics of the code changes themselves.

### **2.1.3 Code Review Processes and Automation**

Code review is a fundamental quality assurance practice in software development, involving the systematic examination of code changes to identify bugs, ensure adherence to coding standards, verify security properties, and maintain overall code quality. Traditional code review processes involve human reviewers manually inspecting code changes and providing feedback to the author, a process that can be time-consuming and subject to human limitations such as fatigue and bias.

The core components of the code review process include: (1) Change Submission, where developers submit their code changes for review; (2) Review Assignment, where appropriate reviewers are selected based on expertise and availability; (3) Change Inspection, where reviewers examine the code for issues; (4) Feedback Provision, where reviewers communicate their findings and suggestions; and (5) Revision and Approval, where the original author addresses feedback and the change is ultimately accepted or rejected.

Code review is inherently collaborative and multidimensional. It involves multiple stakeholders with different perspectives and expertise, each focusing on different aspects of the code. Some reviewers may specialize in security analysis, others in performance optimization, and still others in maintainability and code style. This diversity of focus is essential for comprehensive review but presents challenges for automation.

The dynamics of code review extend beyond simple issue identification. Effective code review involves a conversation between authors and reviewers, negotiation of changes, explanation of design decisions, and collective problem-solving. This collaborative aspect distinguishes code review from many other software engineering tasks and necessitates specialized approaches for automation.

Automation of code review has become increasingly important as software systems grow in complexity and development velocity increases. Automated tools can help address the scalability challenges of manual review, provide consistent evaluation across changes, and reduce the cognitive load on human reviewers. However, capturing the nuanced, collaborative nature of code review in automated systems remains a significant challenge.

### **2.1.4 Repository-Wide Program Repair**

Automated program repair (APR) aims to automatically detect and fix bugs in software without human intervention. Traditional APR approaches typically focus on isolated bugs in individual functions or files, using techniques such as search-based repair, semantic repair, or example-based repair to generate patches that pass a given test suite.

Repository-wide program repair extends this concept to address bugs that span multiple files, classes, or modules within a software repository. This broader scope introduces several unique challenges. First, changes in one part of the codebase may have ripple effects across interdependent components, requiring careful consideration of these dependencies during repair. Second, the search space for potential fixes grows exponentially with the size and complexity of the repository, necessitating efficient search strategies. Third, the evaluation of proposed fixes must consider not only the immediate bug but also potential regressions

in related functionality.

The core components of repository-wide repair include: (1) Bug Localization, which identifies the specific locations in the codebase that need to be modified; (2) Dependency Analysis, which maps the relationships and interactions between different code components; (3) Repair Generation, which produces candidate fixes for the identified bugs; (4) Consistency Checking, which ensures that fixes maintain the integrity of cross-component dependencies; and (5) Validation, which verifies that the proposed repairs fix the target bug without introducing new issues.

Several benchmarks have been developed to evaluate repository-wide program repair techniques. SWE-bench and its refined subset SWE-bench-lite provide realistic scenarios based on actual bugs from open-source projects, challenging repair tools to address complex issues involving multiple files and components. These benchmarks not only test the effectiveness of repair techniques but also their scalability and practicality in real-world development environments.

### **2.1.5 Large Language Models for Code**

Large Language Models (LLMs) have emerged as powerful tools for code understanding and generation, leveraging their ability to learn patterns and semantics from vast corpora of code. These models, based on transformer architectures, have demonstrated impressive capabilities in tasks such as code completion, bug detection, documentation generation, and program repair.

The application of LLMs to code-related tasks builds on several key capabilities. First, these models can capture both the syntactic structure and semantic meaning of code, understanding how different components interact and what functionality they implement. Second, they can generalize across programming languages and coding patterns, applying knowledge learned from one context to another. Third, they can generate coherent and contextually appropriate code based on natural language descriptions or partial implementations.

Current LLMs for code fall into two broad categories: closed-source models developed by commercial entities (e.g., GPT-4, Claude) and open-source models available for public use and modification (e.g., Llama, Mistral, Qwen). While closed-source models often demonstrate superior performance, open-source alternatives offer advantages in terms of accessibility, customizability, and deployment flexibility, particularly in privacy-sensitive or bandwidth-constrained environments.

The capabilities of LLMs for code tasks are enhanced through several specialized techniques. Chain-of-thought (CoT) prompting encourages models to generate intermediate reasoning steps before producing final answers, improving performance on complex tasks that require step-by-step problem solving. Few-shot learning enables models to adapt to new tasks based on a small number of examples, reducing the need for extensive fine-tuning. Reinforcement learning from feedback (human or AI-generated) helps align model outputs with desired criteria, such as correctness, efficiency, or adherence to coding standards.

Despite their impressive capabilities, LLMs for code face several challenges. They may generate syntactically correct but semantically incorrect code, struggle with complex logical reasoning, or fail to consider edge cases and potential vulnerabilities. Addressing these limitations requires specialized techniques that enhance the models' reasoning capabilities and ensure the reliability of their outputs.

### 2.1.6 Dynamic Programming and Multi-Agent Systems

Dynamic programming (DP) is a mathematical optimization method that breaks down complex problems into simpler subproblems and solves each subproblem only once, storing the solutions to avoid redundant computation. In the context of software engineering tasks like code review and program repair, DP provides a formal framework for decomposing complex analyses into manageable components.

The formal elements of DP in this context include: (1) State Space, which represents the set of subproblems or analysis states; (2) Transition Function, which defines how to move from one state to another; (3) Value Function, which quantifies the quality or correctness of a solution at each state; and (4) Optimal Substructure, which ensures that optimal solutions to subproblems can be combined to form optimal solutions to the overall problem.

Multi-agent systems extend this approach by distributing the problem-solving process across multiple autonomous agents, each with specific capabilities and responsibilities. In software engineering contexts, these agents might include specialized analyzers for different aspects of code quality, collaborative reviewers with different expertise, or coordinated repair generators working on different parts of a codebase.

The integration of DP principles with multi-agent systems and LLMs creates powerful frameworks for addressing complex software engineering tasks. LLMs provide the reasoning capabilities needed to analyze code and generate solutions, DP offers a structured approach to breaking down complex problems, and multi-agent architectures enable specialized processing and collaborative problem-solving. This combination is particularly well-suited to tasks like code review and repository-wide program repair, which require both deep understanding of individual components and comprehensive consideration of system-wide interactions.

## 2.2 Related Work

This section reviews existing research related to our work, analyzing current approaches, their strengths, and limitations.

### 2.2.1 Code Change Representation Learning

Research on code change representation has evolved from manual feature engineering to sophisticated learning-based approaches. Early work by Kamei et al. Kamei et al. [2016, 2012] focused on manually designed features such as change size, file history, and developer experience to characterize changes for defect prediction. While effective for specific tasks, these approaches lacked generalizability across different projects and required significant domain expertise.

The emergence of deep learning led to more automated representation approaches. Hoang et al. Hoang et al. [2020] proposed CC2Vec, which learns representations of code changes for commit message generation and defect prediction using a hierarchical attention network. Xu et al. Xu et al. [2019] developed CommitGen, a sequence-to-sequence model for generating commit messages from code diffs. Nie et al. Nie et al. [2021] introduced CoreGen, which combines abstract syntax tree (AST) information with token sequences to improve change representation. These sequence-based approaches captured local patterns effectively but struggled with global structural properties.

Graph-based representations emerged to address these limitations. Liu et al. Liu et al. [2020c] proposed ATOM, which models code changes as graph transformations of the AST. Lin et al. Lin et al. [2022] developed a context-aware graph neural network to capture both the changed code and its surrounding context. Wang et al. Wang et al. [2023b] introduced GraphSPD specifically for security patch detection, using graph neural networks to model

both the structure and semantics of code changes. These approaches better captured structural relationships but sometimes missed sequential patterns important for understanding code evolution.

Hybrid approaches aim to combine the strengths of both sequential and structural representations. Dong et al. Dong et al. [2022] proposed FIRA, which integrates both token sequences and AST structures in a unified representation. Tian et al. Tian et al. [2022a] developed a multi-modal approach that combines code structure, natural language descriptions, and metadata for patch correctness assessment. These integrated approaches show promise but often focus on specific applications rather than general-purpose representations.

Despite these advances, several limitations remain. Most existing approaches focus on the structural or syntactic aspects of changes rather than capturing the developer's intention or the semantic impact of changes. They typically treat code changes as isolated entities without considering their role in the broader codebase evolution. Additionally, they often lack the ability to adapt to different downstream tasks without significant retraining.

### 2.2.2 Security Patch Detection Techniques

Research on security patch detection has evolved from rule-based methods to sophisticated machine learning approaches. Traditional techniques relied on manually defined patterns and heuristics. Tian et al. Tian et al. [2012] proposed a keyword-based approach that identified security patches based on specific terms in commit messages. Wang et al. Wang et al. [2020b, 2019b] extended this with additional structural features extracted from code changes. While these approaches provided a foundation for security patch detection, they suffered from high false positive rates and limited generalizability across different projects and vulnerability types.

Sequential learning models improved upon these limitations by leveraging the natural sequence structure of code. Zhou et al. Zhou et al. [2021b] introduced SPI-DB, a benchmark dataset for security patch identification, along with a sequence-based model that outperformed traditional approaches. Wang et al. Wang et al. [2021c] proposed PatchRNN, a recurrent neural network model that processes code changes as sequences of tokens to identify security-relevant modifications. These approaches captured sequential patterns effectively but struggled with complex structural relationships in code.

Graph-based approaches emerged to address these structural limitations. Wang et al. Wang et al. [2023b] introduced GraphSPD, which constructs semantic representations of code changes using graph neural networks. This approach significantly advanced the state-of-the-art by incorporating both local code context and structural information. However, GraphSPD primarily focused on local code structures and immediate dependencies, limiting its ability to capture cross-module interactions and higher-level security implications.

LLM-based approaches have recently shown promise for security patch detection. Chakraborty et al. Chakraborty et al. [2021] leveraged pre-trained language models to identify security vulnerabilities in code. Yang et al. Yang et al. [2024a] extended this to security patch detection, using LLMs to understand both the code changes and their security implications. These approaches benefit from the rich semantic understanding of LLMs but often lack the specialized representations needed for security-specific tasks.

Despite these advances, several limitations persist in security patch detection. Most approaches struggle with "silent" security patches that lack explicit security-related documentation. They often focus on syntactic patterns rather than semantic understanding of security implications. Additionally, they typically operate on isolated changes without considering system-wide security properties. These limitations highlight the need for ap-

proaches that combine structural code analysis with semantic understanding and broader contextual awareness.

### 2.2.3 Automated Code Review Approaches

Automated code review research spans a spectrum from focused static analysis tools to holistic review systems. Early work primarily focused on static analysis for specific defect types. Beller et al. Beller et al. [2014] surveyed the use of static analysis tools in code review, finding that while useful for detecting certain bug classes, these tools lacked the contextual understanding needed for comprehensive review. Thongtanunam et al. Thongtanunam et al. [2015] investigated factors affecting code review effectiveness, highlighting the importance of reviewer expertise and review scope.

Learning-based approaches have extended static analysis capabilities. Tufano et al. Tu-fano et al. [2021, 2022] proposed using neural machine translation models to learn from historical code reviews, generating automated review comments for new changes. Thongtanunam et al. Thongtanunam et al. [2022] developed AutoTransform, which automatically fixes identified issues during review. Watson et al. Watson et al. [2022] conducted a systematic review of machine learning approaches for code review automation, identifying significant progress but noting limitations in handling the collaborative nature of review.

Recent work has begun addressing the collaborative aspects of code review. Davila et al. Davila and Nunes [2021] analyzed communication patterns in code review, identifying the importance of multi-party interactions for effective review outcomes. Oliveira et al. Oliveira et al. [2023] investigated how reviewers collaborate to assess different quality aspects, finding that diverse expertise leads to more comprehensive reviews. These studies highlight the collaborative nature of code review but do not directly translate these insights into automated approaches.

Agent-based systems represent a promising direction for capturing the collaborative nature of code review. Li et al. Li et al. [2023a] proposed CAMEL, a framework for collaborative AI agents that could be adapted to code review scenarios. Qian et al. Qian et al. [2023] developed a communicative agent framework specifically for software engineering tasks. Hong et al. Hong et al. [2023] introduced MetaGPT, a multi-agent framework that simulates software development teams, including code review activities. These frameworks demonstrate the potential of multi-agent approaches but have not been specifically optimized for the unique challenges of code review.

Challenges in multi-agent code review include maintaining focus and coherence across multiple agents. Zheng et al. Zheng et al. [2024] identified "prompt drifting" as a key issue in multi-agent systems, where conversations diverge from their original purpose. Yang et al. Yang et al. [2024d] proposed techniques for adapting prompts to maintain coherence in multi-turn interactions. These approaches address important coordination challenges but have not been extensively applied to code review scenarios.

Despite progress in automated code review, significant limitations remain. Most approaches focus on specific aspects of review (e.g., bug detection, style enforcement) rather than holistic evaluation. They typically operate in a single-agent paradigm, failing to capture the collaborative nature of human code review. Additionally, they often lack the ability to provide contextually appropriate feedback that balances immediate issue correction with longer-term code quality concerns.

## 2.2.4 Repository-Wide Program Repair

Repository-wide program repair extends beyond fixing isolated bugs to address issues spanning multiple files and components. Traditional APR approaches focused on single-file or single-function repairs. Le Goues et al. Le Goues et al. [2012] introduced GenProg, a genetic programming approach for automated bug fixing. Nguyen et al. Nguyen et al. [2013] developed SemFix, using symbolic execution and constraint solving for semantic repair. Kim et al. Kim et al. [2013] proposed PAR, a pattern-based approach inspired by human-written patches. While these approaches demonstrated success on focused repairs, they struggled with bugs that required changes across multiple files or components.

Repository-level benchmarks have emerged to evaluate more comprehensive repair capabilities. Su et al. Jimenez et al. [2024a] introduced SWE-bench, a benchmark featuring real-world software engineering tasks that require understanding and modifying large codebases. The SWE-bench-lite subset focuses specifically on bug-fixing tasks across multiple files. These benchmarks have highlighted the limitations of traditional APR approaches when applied to repository-wide contexts, where dependencies and interactions between components must be considered.

LLM-based approaches have shown promise for repository-wide repair. Chen et al. Chen et al. [2021a] evaluated large language models for code-related tasks, including bug fixing, finding that they could generate plausible fixes for various bug types. Nijkamp et al. Nijkamp et al. [2023] developed CodeGen, a family of LLMs specifically trained for code generation and repair tasks. Roziere et al. Rozière et al. [2023] introduced Code Llama, demonstrating strong capabilities in understanding and modifying existing codebases. These models provide a foundation for repository-wide repair but often lack the specialized mechanisms needed to ensure consistency across components.

Agent-based approaches have been proposed to address the challenges of repository-wide repair. Zhang et al. Zhang et al. [2024a] developed AutoCodeRover, an agent-based system that iteratively explores and modifies codebases to fix bugs. The SWE-Agent system Yang et al. Yang et al. [2024c] similarly uses an agent-based approach to navigate and repair complex codebases. While these approaches enable more flexible exploration of solution spaces, they face challenges related to tool usage complexity, planning efficiency, and the reliability of multi-turn interactions.

Liu et al. Liu et al. [2024b] conducted a comprehensive survey of agent-based approaches for software engineering, identifying common challenges including error-prone tool usage and inefficient exploration strategies. Olausson et al. Olausson et al. [2023] and Shi et al. Shi et al. [2023] both highlighted issues with cascading errors in multi-turn agent interactions, where mistakes in early decisions compound through subsequent steps. These limitations suggest the need for more structured approaches to repository-wide repair.

CallGraph-driven approaches offer a structured alternative to agent-based exploration. Zhou et al. Zhou et al. [2012] demonstrated the importance of call graphs for understanding bug propagation across components. Austin et al. Austin et al. [2021] used call graphs to identify components affected by specific program changes. These works highlight the potential of structural code representations for guiding repository-wide repairs, but they have not been fully integrated with the generative capabilities of modern LLMs.

Despite advances in repository-wide program repair, significant challenges remain. Most approaches struggle to ensure consistency across interdependent components when changes are made. They often lack efficient mechanisms for identifying all relevant components affected by a bug. Additionally, they typically focus on functional correctness without considering broader quality aspects such as maintainability and performance implications.

### 2.2.5 Performance Gap in Code LLMs

A significant performance gap exists between closed-source and open-source large language models for code tasks. Bommasani et al. Bommasani et al. [2021] highlighted the challenges and opportunities presented by foundation models, noting the tension between the superior capabilities of proprietary models and the accessibility benefits of open alternatives. Evaluations by Chen et al. Chen et al. [2021a], Nijkamp et al. Nijkamp et al. [2023], and Roziere et al. Rozière et al. [2023] have consistently shown that closed-source models like GPT-4 and Claude outperform their open-source counterparts on code understanding and generation tasks, including program repair.

Several techniques have been proposed to enhance the performance of open-source LLMs for code tasks. Instruction tuning approaches, as explored by Chen et al. Chen et al. [2021a], adapt pre-trained models to follow specific instructions, improving their ability to perform targeted code manipulation tasks. Li et al. Li et al. [2022a] investigated reinforcement learning techniques for improving code model performance, particularly on challenging competition-level problems. These approaches show promise but typically require extensive human annotation or feedback, limiting their scalability as noted by Ouyang et al. Ouyang et al. [2022].

Knowledge distillation represents a promising approach for transferring capabilities from closed-source to open-source models. Hinton et al. Hinton et al. [2015] introduced the core concept of knowledge distillation, where a smaller "student" model learns from a larger "teacher" model. Sanh et al. Sanh et al. [2019] applied this approach to language models with DistilBERT, creating a more efficient model that retained most of the capabilities of its teacher. In the code domain, Wang et al. Wang et al. [2023e] developed CodeDistill, demonstrating effective transfer of code understanding capabilities from larger to smaller models. These approaches focus primarily on general language capabilities rather than the specific reasoning processes needed for complex tasks like program repair.

Chain-of-thought (CoT) prompting has emerged as a powerful technique for enhancing reasoning capabilities in LLMs. Wei et al. Wei et al. [2022] showed that prompting models to generate intermediate reasoning steps significantly improves performance on complex reasoning tasks. Kojima et al. Kojima et al. [2022] extended this to zero-shot settings, demonstrating that simply asking models to reason step-by-step improves outcomes even without examples. Chen et al. Chen et al. [2022] applied similar ideas to program synthesis, showing that explicit reasoning about problem decomposition and solution strategies leads to better code generation. These approaches enhance model reasoning but have not been systematically applied to transferring reasoning capabilities between models.

Reinforcement learning from feedback represents another promising direction for improving model performance. Christiano et al. Christiano et al. [2017] and Ouyang et al. Ouyang et al. [2022] developed reinforcement learning from human feedback (RLHF) techniques, which use human preferences to guide model improvement. Recent work by Lee et al. Lee et al. [2023] and Bai et al. Bai et al. [2022] has explored using AI systems rather than humans to provide feedback, enabling more scalable training. In the code domain, Chen et al. Chen et al. [2023b] used reinforcement learning with automated feedback from test case results to improve code generation. Yuan et al. Yuan et al. [2023a] similarly applied reinforcement learning to enhance reasoning about code. These approaches offer scalable ways to improve model performance but have not been extensively applied to bridging the gap between open and closed-source models for program repair.

Despite these advances, significant challenges remain in closing the performance gap between open and closed-source code models. Most approaches focus on general capabilities rather than the specific reasoning processes needed for complex tasks like program repair.

They typically require extensive training data or human feedback, limiting their practical applicability. Additionally, they often fail to capture the structured, logical reasoning that distinguishes top-performing models in code-related tasks.

### **2.2.6 Program Repair with LLMs**

Large language models have recently demonstrated impressive capabilities in automated program repair. Zhao et al. Zhao et al. [2023] introduced ChatRepair, leveraging ChatGPT’s capabilities for automated bug fixing through structured prompting. Wei et al. Wei et al. [2023c] developed Repilot, a framework that guides LLMs through the repair process with specialized prompts and context management. Dakhel et al. Dakhel et al. [2023] created RING, which integrates LLM-based repair directly into development workflows, providing just-in-time fix suggestions. These systems demonstrate the potential of LLMs for program repair but primarily rely on proprietary models, limiting their accessibility and customizability.

Reasoning enhancement techniques have been proposed to improve LLM performance on program repair tasks. Wei et al. Wei et al. [2022] introduced chain-of-thought prompting, which encourages models to generate intermediate reasoning steps before producing final answers. Jin et al. Jin et al. [2023] developed ARCADE, which applies structured reasoning frameworks specifically to program repair, decomposing the repair process into explicit steps. Zhang et al. Zhang et al. [2023c] proposed PACE, a planning-based approach that guides LLMs through systematic exploration of repair strategies. These approaches improve repair quality but have not been systematically incorporated into open-source models.

Reinforcement learning methods have shown promise for aligning LLMs with desired repair behaviors. Christiano et al. Christiano et al. [2017] and Ouyang et al. Ouyang et al. [2022] developed foundational techniques for reinforcement learning from human feedback (RLHF), which have been widely adopted for model alignment. Recent work by Lee et al. Lee et al. [2023] and Bai et al. Bai et al. [2022] has explored using AI feedback instead of human annotations, enabling more scalable training. These approaches provide frameworks for improving model behavior but have not been extensively specialized for the unique requirements of program repair.

Despite these advances, significant challenges remain in LLM-based program repair. Most approaches focus on generating plausible fixes without systematically ensuring correctness across all scenarios. They often struggle with complex bugs that require deep understanding of program semantics and intercomponent dependencies. Additionally, they typically rely on proprietary models, limiting their accessibility and integration into development workflows where privacy or deployment constraints exist.

# 3 Learning to Represent Code Changes

*In this chapter, we propose to investigate a novel representation learning approach for code changes that effectively captures both sequential and structural semantics. Our approach, Patcherizer, addresses the limitations of existing methods by combining three key components: (1) the context surrounding code changes, (2) a novel SeqIntention representation that captures semantic intent within token-based changes, and (3) a GraphIntention representation that models structural modifications at the AST level. By integrating these complementary perspectives, Patcherizer creates more comprehensive embeddings that preserve both contextual and transformation semantics. Our extensive evaluation across three downstream tasks demonstrates that Patcherizer significantly outperforms the state-of-the-art approaches, achieving improvements of 19.39% in BLEU, 8.71% in ROUGE-L, and 34.03% in METEOR for commit message generation, while also excelling in defect prediction and patch correctness assessment. Finally, our approach enables more effective representation learning for various software engineering tasks by explicitly modeling how modifications affect program execution flow and structural relationships.*

The content presented herein derives from scholarly investigation previously documented in the academic publication referenced below:

- **Xunzhu Tang**, Haoye Tian, Weiguo Pian, Saad Ezzini, Abdoul Kader Kaboré, Andrew Habib, Kisub Kim, Jacques Klein, and Tegawendé F. Bissyandé. Learning to Represent Code Changes. *Empirical Software Engineering (EMSE)* (2025): minor revision (under review).

## Contents

---

<b>3.1</b>	<b>Introduction</b>	<b>21</b>
<b>3.2</b>	<b>Patcherizer</b>	<b>23</b>
3.2.1	Handling Incomplete Code Snippets	23
3.2.2	Code Change Preprocessing	23
3.2.3	Sequence Intention Encoder	24
3.2.4	Graph Intention Encoder	27
3.2.5	Graph Intention Encoding Details	29
3.2.6	Aggregator: Aggregating Multimodal Input Embeddings	31
3.2.7	Pre-training	31
3.2.8	Fine-tuning for Different Tasks	31

<b>3.3 Experimental Design . . . . .</b>	<b>34</b>
3.3.1 Implementation . . . . .	34
3.3.2 Research questions . . . . .	34
3.3.3 Baselines . . . . .	34
3.3.4 Datasets . . . . .	35
3.3.5 Metrics . . . . .	35
<b>3.4 Experimental Results . . . . .</b>	<b>37</b>
3.4.1 [RQ-1]: Performance of Patcherizer . . . . .	37
3.4.2 [RQ-2]: Ablation Study . . . . .	42
3.4.3 [RQ-3]: Generalizability and Robustness . . . . .	49
<b>3.5 Discussion . . . . .</b>	<b>52</b>
3.5.1 Comparison with Slice-Based Code Change Representation . . . . .	52
3.5.2 Threats to Validity . . . . .	53
3.5.3 Limitations . . . . .	53
3.5.4 Handling Incomplete Code Information . . . . .	53
3.5.5 Extensibility . . . . .	54
<b>3.6 Related Work . . . . .</b>	<b>56</b>
3.6.1 Code change Representation . . . . .	56
3.6.2 Applications of Code Change Embeddings . . . . .	56
<b>3.7 Conclusion . . . . .</b>	<b>58</b>
<b>3.8 Conflict of Interest Statement . . . . .</b>	<b>58</b>
<b>3.9 Acknowledgments . . . . .</b>	<b>58</b>
<b>3.10 Ethical approval . . . . .</b>	<b>58</b>
<b>3.11 Informed consent . . . . .</b>	<b>58</b>
<b>3.12 Author Contributions . . . . .</b>	<b>58</b>
<b>3.13 Availability: . . . . .</b>	<b>59</b>

---

## 3.1 Introduction

A software code change represents the source code differences between two software versions. It has a dual role: on the one hand, it serves as a formal summary of the code changes that a developer intends to make on a code base; on the other hand, it is used as the main input specification for automating software evolution. Code changes are thus a key artifact that is pervasive across the software development life cycle. In recent years, building on empirical findings on the repetitiveness of code changes Barr et al. [2014], several approaches have built machine learning models based on code change datasets to automate various software engineering tasks such as code change description generation Linares-Vásquez et al. [2015], Buse and Weimer [2010], Cortés-Coy et al. [2014], Jiang et al. [2017], Xu et al. [2019], Liu et al. [2019, 2018, 2020c], code completion Svyatkovskiy et al. [2019], Liu et al. [2020b,a], Ciniselli et al. [2021], Pian et al. [2022], code change correctness assessment Tian et al. [2022c], and just-in-time defect prediction Hoang et al. [2019], Kamei et al. [2016].

Early work manually crafted a set of hand-picked features to represent a code change Kamei et al. [2016, 2012]. Recently, the successful application of deep learning to learn powerful representations of text, signal, and images Devlin et al. [2018], Niemeyer and Geiger [2021], Qin et al. [2021], Li et al. [2021], Tang et al. [2021], Pian et al. [2023b], Wang et al. [2022b] led to adopting similar techniques in software engineering where researchers develop deep ML models to represent code and code changes Yin et al. [2019], Hoang et al. [2020], Feng et al. [2020b], Nie et al. [2021], Jiang et al. [2021], Tian et al. [2022c], Pian et al. [2022], Liu et al. [2023]. Initially, these approaches treated code Feng et al. [2020b], Elnaggar et al. [2021], Wang et al. [2021f], Guo et al. [2021a,a] and other code-like artifacts, such as code changes Xu et al. [2019], Nie et al. [2021], Dong et al. [2022], Liu et al. [2020c], as a sequence of tokens and thus employ natural language processing methods to extract code in text format. Researchers have recognized the limitations of using code token sequences alone (often represented by + and – lines in the textual diff format) to capture the full semantics of code changes, as these symbols lack inherent meaning that a DL model can learn. To address this, they began incorporating the code structure, such as Abstract Syntax Trees (ASTs), to better capture the underlying structural information in source code Zhang et al. [2019a], Alon et al. [2019, 2020], Guo et al. [2021a]. Therefore, recent work such as commit2vec Cabrera Lozoya et al. [2021], C<sup>3</sup> Brody et al. [2020], and CC2Vec Hoang et al. [2020] attempted to represent code changes more structurally by leveraging ASTs as well. To get the best of both worlds, more recent work tried to combine token information with structure information to obtain a better code change representation Dong et al. [2022]. Finally, several such approaches of code change representation learning have been evaluated on specific tasks, e.g., BATS Tian et al. [2022a] for code change correctness assessment and FIRA Dong et al. [2022] for code change description generation.

On the one hand, token-based approaches for code change representation Hoang et al. [2020], Xu et al. [2019], Nie et al. [2021] lack the rich structural information of source code and intention features inside the sequence is still unexplored. On the other hand, graph-based representation of code changes Liu et al. [2020c], Lin et al. [2022] lacks the context which is better represented by the sequence of tokens Hoang et al. [2020], Xu et al. [2019], Nie et al. [2021] of the code change itself and also the surrounding unchanged code and intention features inside graph changes is also still unexplored. In conclusion, approaches that try to combine context and AST information to represent code changes (e.g., FIRA Dong et al. [2022]) do not use the intention features of either sequence or graph from the code change but rather rely on representing the code before and after the change while adding some

*ad-hoc* annotations to highlight the changes for the model.

**This paper.** We propose a novel code change representation that tackles the aforementioned problems and provides an extensive evaluation of our approach on three practical and widely used downstream software engineering tasks. Our approach, Patcherizer, learns to represent code changes through a combination of (1) the context around the code change, (2) a novel SeqIntention representation of the sequential code change, and (3) a novel representation of the GraphIntention from the code change. Our approach enables us to leverage powerful DL models for the sequence intention such as Transformers and similarly powerful graph-based models such as GCN for the graph intention. Additionally, our model is pre-trained and hence task agnostic where it can be fine-tuned for many downstream tasks. We provide an extensive evaluation of our model on three popular code change representation tasks: (1) Generating code change description in NL, (2) Predicting code change correctness in program repair, and (3) Code change intention detection.

Overall, this paper makes the following contributions:

- ▶ *A novel representation learning approach for code changes:* we combine the context surrounding the code change with a novel sequence intention encoder and a new graph intention encoder to represent the intention of code changes in the code change while enabling the underlying neural models to focus on the code change by representing it explicitly. To that end, we developed: ① *an adapted Transformer architecture for code sequence intention* to capture sequence intention in code changes taking into account not only the changed lines (added and removed) but also the full context (i.e., the code chunk before the code change application); ② *an embedding approach for graph intention* to compute embeddings of graph intention capturing the semantics of code changes.
- ▶ *A dataset of parsable code changes:* given that existing datasets only provide code changes with incomplete details for readily collecting the code before and after the code change, extracting AST diffs was challenging. We therefore developed tool support to enable such collection and produced a dataset of 90k code changes, which can be parsed using the Java compiler.
- ▶ *Extensive evaluation:* we evaluate our approach by assessing its performance on several downstream tasks. For each task, we show how Patcherizer outperforms carefully-selected baselines. We further show that Patcherizer outperforms the state of the art in code change representation learning.

**Motivation for Code Change Intention Detection.** In addition to description generation and correctness prediction, we introduce and explore a novel downstream task: *code change intention detection*. Unlike traditional diff tools that identify syntactic operations like additions or deletions, our goal is to uncover the semantic *intent* behind a change. This capability is crucial for automating commit message refinement, supporting intelligent software analytics, and enabling advanced tooling for software reviewers. For example, being able to detect whether a patch intends to disable a feature versus merely refactor a method allows tools to generate more meaningful messages or guide human reviewers' attention to impactful modifications. We argue that intention detection should go beyond surface diffs and rely on learned semantics from both structure and context—a gap that Patcherizer fills effectively.

## 3.2 Patcherizer

Figure 7.1 presents the overview of Patcherizer. Code changes are first preprocessed to split the available information about added (+) and removed (−) lines, identifying the code context (i.e., the code chunk before applying the code change) and computing the ASTs of the code before and after applying the code changes (cf. Section 3.2.2). Then, Patcherizer deploys two encoders, which capture sequence intention semantics (cf. Section 3.2.3) and graph intention semantics (cf. Section 3.2.4). Those encoded information are aggregated (cf. Section 3.2.6) to produce code change embeddings that can be applied to various downstream tasks. In the rest of this section, we will detail the different components of Patcherizer before discussing the pre-training phase (cf. Section 3.2.7).

### 3.2.1 Handling Incomplete Code Snippets

In real-world software repositories, patches often appear as partial code fragments extracted from diffs, lacking full method or class definitions required for reliable parsing and structural analysis. To ensure that our model can robustly represent such incomplete code, **Patcherizer** employs a *context construction mechanism* that reconstructs semantically valid and parsable code by leveraging surrounding, unmodified code as context.

During the preprocessing phase, we identify the surrounding code chunk prior to patch application (referred to as `cbp`) and combine it with the lines to be added (`ccp`) and removed (`ccm`) to recover a coherent code segment. This reconstruction not only aids in forming a complete representation for the sequential input but also enables successful parsing of the Abstract Syntax Tree (AST) required for graph-based representation.

Specifically, when certain structural elements are missing in the diff (e.g., a method body without its signature or class declaration), our system uses a sliding window strategy to extract surrounding lines from the same file to syntactically complete the snippet. We then apply the `javalang` parser to the reconstructed code segment to generate the corresponding ASTs before and after the change. If the full reconstruction still fails due to excessive incompleteness, we fall back to using only the available sequence-level representation without graph-based embedding.

This fallback mechanism ensures that **Patcherizer** can generalize to a wide range of real-world patch inputs—whether they are complete or incomplete—by balancing robustness with semantic fidelity. It is particularly valuable in scenarios such as just-in-time commit analysis, where full project context may not be readily available.

### 3.2.2 Code Change Preprocessing

The preprocessing aims to focus on three main information within a code change for learning its representation. The code before applying the code change (which provides contextual information of the code change), plus and minus lines (which provide information about the code change operations), and the difference between *AST* graphs before and after code change (which provides information about graph intention in the code). Through the following steps we collect the necessary multi-modal inputs (code text, sequence intention, and graph intention) for the learning:

1. **Collect +/− lines in the code change.** We scan each code change line. Those starting with a + are added to a *pluslist*, while those starting with a − are added to a *minuslist*. Both lists record the line numbers in the code change.
2. **Reconstruct before/after code.** Besides +/− lines, a code change includes unchanged code that are part of the context. We consider that the full context is the code before

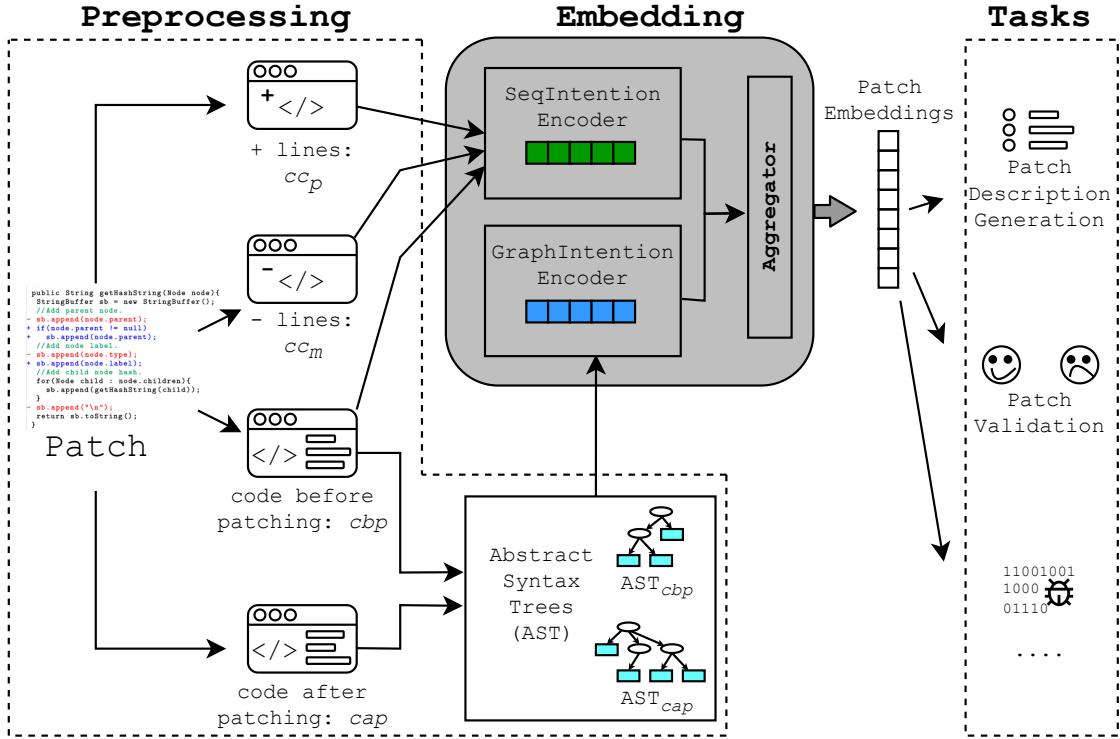


Figure 3.1: Overview of Patcherizer.

applying the code change (i.e., unchanged & minuslist lines). We also construct the code after applying the code change (i.e., unchanged & pluslist lines). The reconstruction leverages the recorded line numbers for inserting each added/removed line to the proper place and ensure accuracy.

3. **Generate code ASTs before and after code change.** We apply the Javalang Thunes [2013] tool to generate the ASTs for the reconstructed code chunks before and after applying the code change.
4. **Construct vocabulary.** Based on the code changes of the code changes in the training data, we build a vocabulary using the Byte-Pair-Encoding (*BPE*) algorithm.

At the end of this preprocessing phase, for each given code change, we have a set of inputs:  $\langle cc_p, cc_m, cbp, cap, G_{cbp}, G_{cap} \rangle$ , where  $cc_p$  is the sequence of added (+) lines of code,  $cc_m$  is the sequence of removed (-) lines of code,  $cbp$  is the code chunk before the patch is applied,  $cap$  is the code chunk after the patch is applied,  $G_{cbp}$  is the AST graph of  $cbp$  and  $G_{cap}$  is the graph of  $cap$ .

### 3.2.3 Sequence Intention Encoder

Intention features refer to the semantic signals that capture *why* a change was made (e.g., to fix a bug, refactor code, disable functionality), rather than just *what* was changed. Unlike prior approaches that focus on syntax or structural patterns, our intention encoders aim to learn such semantics through contextualized token changes (in the sequence encoder) and structural shifts (in the graph encoder).

Despite the success of AST-based methods, they often struggle to distinguish code changes that are structurally similar but semantically distinct. Our intention encoders are designed to capture such nuances. Figure 3.2 provides an illustrative example: although both code changes involve editing an `if` condition, one disables functionality while the other refactors logic. Traditional AST diffs treat both similarly, but their purposes are fundamentally different. Patcherizer captures these differences through intention-aware encoding.

```

1 if (x) {
2     doSomething();
3 }

```

Listing 3.1: Disabling a Feature

```

1 if (checkCondition()) {
2     doSomething();
3 }

```

Listing 3.2: Refactoring Logic

**Figure 3.2:** Example illustrating how semantically distinct code changes can result in similar AST diffs. Both code changes involve replacing the condition in an “if” statement, and their corresponding AST diffs may appear structurally similar. However, their *intentions* differ: the left change disables functionality, while the right one introduces a refactor with encapsulated logic. Traditional AST-based models may treat these equivalently, while Patcherizer captures this distinction through its sequence and graph intention encoders.

A first objective of Patcherizer is to build an encoder that is capable of capturing the semantics of the sequence intention in a code change. Although prior work focuses mainly on  $+/ -$  lines or simply does some calculation between changed codes and their contextual contents, we postulate that code context is a relevant additional input for better encoding such differences. Figure 3.3 depicts the architecture of the Sequence intention encoder. We leverage the relevant subset of the preprocessed inputs (cf. Section 3.2.2) to pass to a Transformer embedding layer and further develop a specialized layer, named the *SeqIntention embedding layer*, which captures the intention features from the sequence.

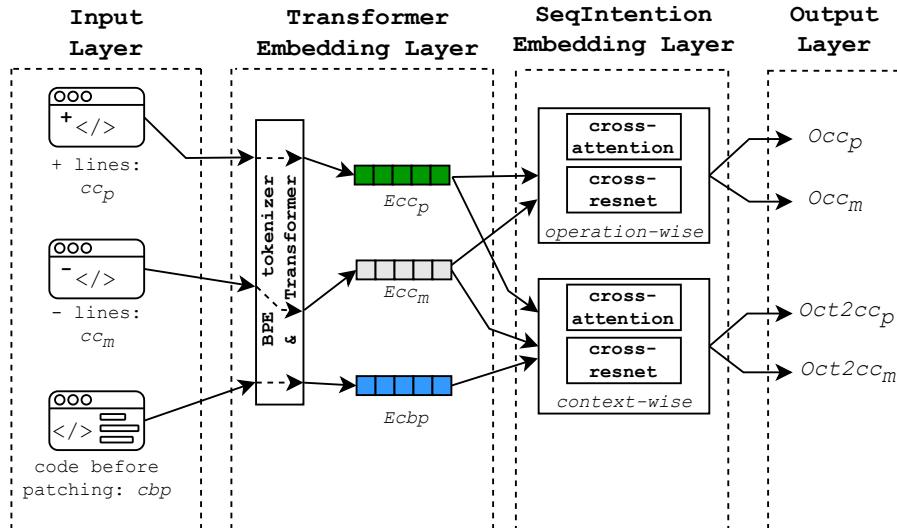


Figure 3.3: Architecture for the Sequence Intention Encoder.

### 3.2.3.1 Input Layer

The input, for each code change, consists of the triplet  $\langle cc_m, cc_p, cbp \rangle$ , where  $cc_m$  is the set of removed ( $-$ ) lines,  $cc_p$  the set of added ( $+$ ) lines and  $cbp$  is the code before code changeing, which represents the context.

### 3.2.3.2 Transformer Embedding Layer

To embed the sequence of code changes, we use a Transformer as the initial embedder. Indeed, Transformers have been designed to capture semantics in long texts and have been demonstrated to be effective for inference tasks Devlin et al. [2018], Wang et al. [2022a].

We note that  $cc_m \in cbp$ . Assuming that  $cc_p = \{token_{p,1}, \dots, token_{p,j}\}$ ,  $cc_m = \{token_{m,1}, \dots, token_{m,k}\}$ ,  $cbp = \{token_{cbp,1}, \dots, token_{cbp,l}\}$ , where  $j, k, l$  represent the maximum length of  $cc_p$ ,  $cc_m$ , and  $cbp$  respectively, we use the initial embedding layer in the Pytorch’s nn.module implemen-

tation to produce first vector representations for each input information as:

$$E_X = \text{Transformer}(\text{Init}(X; \Theta_1); \Theta_2) \quad (3.1)$$

where  $X$  represents an input (either  $cc_m$ ,  $cc_p$  or  $cbp$ );  $\text{Init}$  is the initial embedding function;  $\text{Transformer}$  is the model based on a transformer architecture;  $\Theta_1$  and  $\Theta_2$  are the parameters of  $\text{Init}(\cdot)$  and  $\text{Transformer}(\cdot)$ , respectively.

The Transformer embedding layer outputs  $E_{cc_p} = [e_{p,1}, e_{p,2}, \dots, e_{p,j}] \in \mathbb{R}^{j \times d_e}$ ,  $E_{cc_m} = [e_{m,1}, e_{m,2}, \dots, e_{m,k}] \in \mathbb{R}^{k \times d_e}$ ,  $E_{cbp} = [e_{cbp,1}, e_{cbp,2}, \dots, e_{cbp,l}] \in \mathbb{R}^{l \times d_e}$ , where  $d_e$  is the size of the embedding vector.

### 3.2.3.3 SeqIntention Embedding Layer

Once the Transformer embedding layer has produced the initial embeddings for the inputs  $cc_p$ ,  $cc_m$  and  $cbp$ , our approach seeks to capture how they relate to each other. Prior works Shaw et al. [2018], Devlin et al. [2018], Xu et al. [2019], Dong et al. [2022] have proven that self-attention is effective in capturing relationships among embeddings. We thus propose to capture relationships between the added and removed sequences, with the objective of capturing the intention of the code change through the change operations. We also propose to pay attention to context information when capturing the semantics of the sequence intention.

**Operation-wise:** To obtain the intention of modifications in code changes, we apply a cross-attention mechanism between  $cc_p$  and  $cc_m$ . To that end, we design a resnet architecture where the model performs residual learning of the importance of inputs (*i.e.*,  $E_{cc_p}$  and its evolved  $\sqsubseteq_p$ , which will be introduced below).

To enhance  $E_{cc_p}$  into  $E_{cc_m}$ , we apply a **cross-attention** mechanism. For the  $i$ -th token in  $cc_m$ , we compute the matrix-vector product,  $E_{cc_p} e_{m,i}$ , where  $e_{m,i} \in \mathbb{R}^{d_e}$  is a vector parameter for  $i$ -th token  $\rangle$  in  $cc_m$ . We then pass the resulting vector through a softmax operator, obtaining a distribution over locations in the  $E_{cc_p}$ ,

$$\alpha_{\rangle} = \text{SoftMax}(E_{cc_p} e_{m,i}) \in \mathbb{R}^k, \quad (3.2)$$

where  $\text{SoftMax}(x) = \frac{\exp(x)}{\sum_j \exp(x_j)}$ .  $\exp(x)$  is the element-wise exponentiation of the vector  $x$ .  $k$  is the length of  $cc_m$ . The attention  $\alpha$  is then used to compute vectors for each token in  $cc_m$ ,

$$\sqsubseteq_{\rangle} = \sum_{n=1}^j \alpha_{\rangle, n} h_n. \quad (3.3)$$

where  $h_n \in E_{cc_p}$ ,  $j$  is the length of  $cc_p$ . In addition,  $\sqsubseteq_{\rangle}$  is the new embedding of  $i$ -th token in  $cc_m$  enhanced by semantic of  $E_{cc_p}$ .

Then, we get new  $cc_m$  embedding  $v_m = [\sqsubseteq_1, \dots, \sqsubseteq_k] \in \mathbb{R}^{k \times d_e}$ .

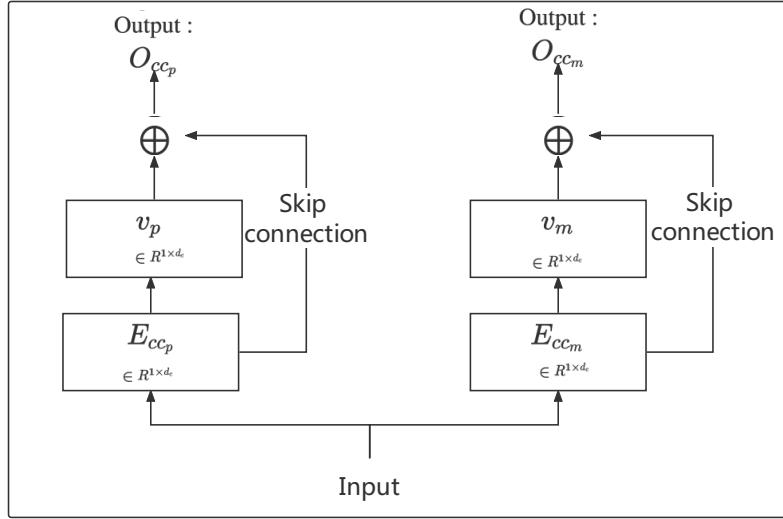
Similarly, following steps above, we can obtain new embedding of  $cc_p$ ,  $v_p \in \mathbb{R}^{j \times d_e}$ , enhanced by the semantic of  $cc_m$ .

For the combination of  $v_p$ ,  $v_m$ ,  $E_{cc_p}$ ,  $E_{cc_m}$ , inspired by Shi et al. [2021], He et al. [2016], we design a **cross-resnet** for combining  $v_p$ ,  $v_m$ ,  $v_{cc_p}$ , and  $v_{cc_m}$ . The pipeline of **cross-resnet** is shown in Figure 3.4. The process is as follows:

$$\begin{aligned} O_{cc_p} &= f(n(E_{cc_p}) + \mathcal{A}[\lceil (E_{cc_p}, v_p)]) \\ O_{cc_m} &= f(n(E_{cc_m}) + \mathcal{A}[\lceil (E_{cc_m}, v_m)]) \end{aligned} \quad (3.4)$$

where  $n(\cdot)$  is a normalization function in Devlin et al. [2018];  $\mathcal{A}[\lceil (\cdot)]$  is the adding function,  $f(\cdot)$  represents *RELU* Glorot et al. [2011] activation function.

Finally, we obtain output  $O_{cc_m}$  and  $O_{cc_p}$ .



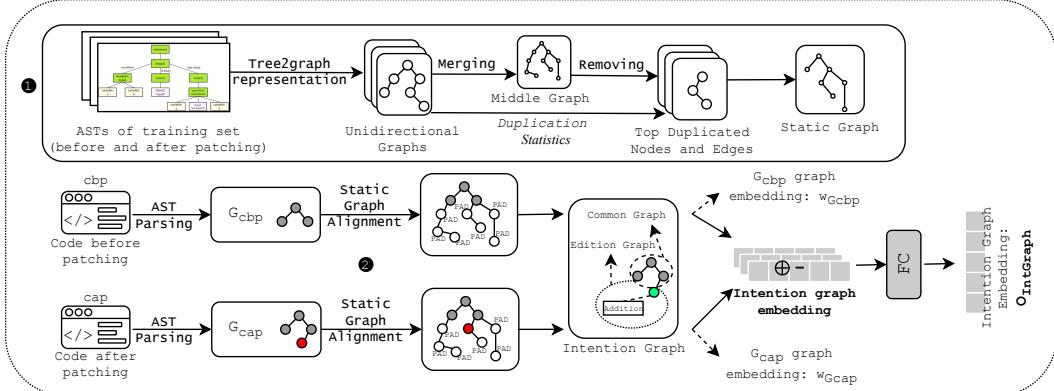
**Figure 3.4:** cross-resnet architecture.

**Context-Wise:** Similar to operation-wise block, we enhance the contextual information into modified lines by **cross-attention** and **cross-resnet** blocks. The computation process is as follows:

$$\begin{aligned} O_{ct2cc_p} &= f(n(E_{cc_p}) + \mathcal{A} \lceil \lceil (E_{cc_p}, E_{cbp})) \\ O_{ct2cc_m} &= f(n(E_{cc_m}) + \mathcal{A} \lceil \lceil (E_{cc_m}, E_{cbp})) \end{aligned} \quad (3.5)$$

where  $E_{cbp}$  is the embedding of  $cbp$  calculated by Equation 3.1;  $O_{ct2cc_p}$  represents the vector of context-enhanced plus embedding and  $O_{ct2cc_m}$  is the vector of context-enhanced minus embedding.

### 3.2.4 Graph Intention Encoder



**Figure 3.5:** Architecture for the Graph Intention Encoder.

Concurrently to encoding sequence information from code changes, we propose to also capture the graph intention features of the structural changes in the code when the code change is applied. To that end, we rely on a Graph Convolutional Network (GCN) architecture, which is widely used to capture dynamics in social networks, and is effective for typical graph-related tasks such as classification or knowledge injection Kipf and Welling [2016], Zhang et al. [2019b,d]. Once the GCN encodes the graph nodes, the produced embeddings can be used to assess their relationship via computing their cosine similarity scores Luo et al. [2018]. Concretely, in Patcherizer, we use a GCN-based model to capture the graph intention features. The embedder model was trained by inputting a static graph, a graph resulting from the merge of all sub-graphs from the training set. Overall, we implement

this encoding phase in two steps: building the static graph, performing graph learning and encoding the graph intention (cf. Figure 3.5).

### 3.2.4.1 Graph Building

To start, we consider the  $G_{cbp}$  and  $G_{cap}$  trees, which represent in graph forms.

**① Static Graph building:** Each code change in the dataset can be associated to two graphs:  $G_{cbp}$  and  $G_{cap}$ , which are obtained by parsing the *cbp* and *cap* code snippets. After collecting all graphs (which are unidirectional graphs) for the whole training set, we merge them into a “big” graph by iteratively linking the common nodes. In this big graph, each distinct code snippet AST-inferred graph is placed as a distinct sub-graph. Then, we will merge the graphs shown in Figure 3.6 which illustrates the merging progress of two graphs: if a node  $\mathcal{N}$  has the same value, position, and neighbors in both ASTs, it will be merged into one (e.g., red nodes 1 and 2). However, when a common node has different neighbors between the ASTs (e.g., red nodes 3 and 4), the merge keeps one instance of the common node but includes all neighbors connected to the merged red nodes (*i.e.*, all green and grey nodes are now connected to red nodes 3 and 4, respectively). After iterating over all graphs, we eventually build the static graph.

However, on the one hand, some nodes in most subgraphs such as ‘prefix\_operators’, ‘returnStatement’, and ‘StatementExpression’ are not related to the semantics of the code change. On the other hand, as data statistics in our study, 97.2% nodes in the initial graphs (ASTs) extracted from code are from parser tools instead of code changes, which means these nodes are rarely related with the semantic of the code change. Thus, as shown in step 1 in Figure 3.5, we remove nodes whose children do not contain words in the code change to reduce the size of the graph because these nodes will be considered noise in our research.

In the remainder of this paper, we refer to the final graph as the *static graph*  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ , where  $\mathcal{V}$  is the set of nodes and  $\mathcal{E}$  is the set of edges.

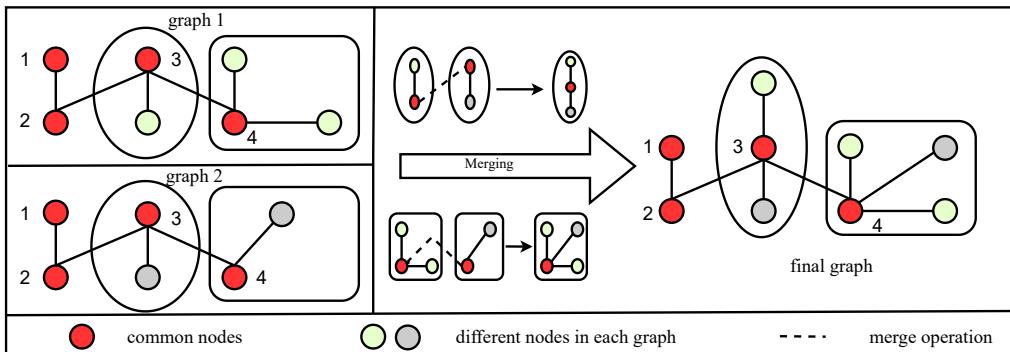


Figure 3.6: An example of merging graphs.

**② Graph Alignment to the Static Graph:** GCN requires that the input graphs are all of the same size Kipf and Welling [2016]. Yet, the graphs built using the graphs of *cbp* and *cap* do not have as many nodes and edges as the static graph used for training the GCN network. Consequently, we propose to use the global static graph to initialize all specific ast-diff graphs for unique code change. Given the global static graph (huge graph mentioned before)  $\mathcal{G}_{global} = (\mathcal{V}_g, \mathcal{E}_g)$  and an AST graph  $\mathcal{G}_{local} = (\mathcal{V}_l, \mathcal{E}_l)$ , we leverage the *VF* graph matching algorithm Cordella et al. [1999] to find the most similar sub-graph with  $\mathcal{G}_{local}$  in  $\mathcal{G}_{global}$ :

$$subGraph = VFG(\mathcal{G}_{local}, \mathcal{G}_{global}). \quad (3.6)$$

where  $VFG(\cdot)$  is the function representing the *VF* matching algorithm net [2018]. The matched sub-graph is a subset of both  $\mathcal{G}_{local}$  and  $\mathcal{G}_{global}$ : some nodes of  $subGraph$  will be

in  $\mathcal{G}_{local}$  but not in  $\mathcal{G}_{global}$ . We then align  $\mathcal{G}_{local}$  to the same size of  $\mathcal{G}_{global}$  as follows: we use the [PAD] element to pad the node of subGraph to the same size of  $\mathcal{G}_{global}$  and then we obtain  $\mathcal{G}_{l_{PAD}}$ . Therefore,  $\mathcal{G}_{l_{PAD}}$  keeps the same size and structure of the static graph  $\mathcal{G}_{global}$ . Eventually, all graphs are aligned to the same size of  $\mathcal{G}_{global}$  and the approach can meet the requirements for GCN computation for graph learning.

### 3.2.5 Graph Intention Encoding Details

The graph intention encoding process involves comparing and merging Abstract Syntax Trees (ASTs) from the original and modified code. Our approach uses a tree-based differencing algorithm inspired by GumTree [11] but optimized for Java ASTs. The AST comparison and merging process follows three key phases:

1. **Node Mapping:** We identify mappings between nodes in the original and modified ASTs using a combination of structural similarity (based on node types and parent-child relationships) and token-level similarity metrics. This hybrid approach achieves approximately 92% accuracy when evaluated against manually labeled AST diffs from our dataset.
2. **Change Detection:** Based on these mappings, we detect node operations (addition, deletion, update, move) by analyzing both mapped and unmapped nodes. For unmapped nodes in the modified AST, we classify them as additions; for unmapped nodes in the original AST, we classify them as deletions; for mapped nodes with different values, we classify them as updates.
3. **Graph Construction:** We construct a unified graph where matched nodes from both ASTs are merged into single nodes with special attributes indicating whether they were preserved, added, or deleted. This creates a comprehensive representation that explicitly encodes the transformation between the original and modified code.

The remaining 8% of cases where the AST differencing is imperfect typically involve complex refactorings with significant structural changes. To mitigate the impact of these inaccuracies on downstream tasks, our dual-encoding approach complements the graph intention encoding with sequence intention encoding, providing robustness when AST-based differencing is imperfect. As demonstrated in our ablation studies (RQ-2), when graph intention encoding contains inaccuracies, the sequence intention encoding can effectively compensate, maintaining strong performance across all downstream tasks.

The graph merging process preserves the original hierarchical relationships while adding special edges to represent the transformation operations. This allows our model to learn patterns of how code structures evolve rather than just focusing on token-level changes, contributing significantly to the improved performance across all downstream tasks as demonstrated in our experiments.

#### 3.2.5.1 Graph Learning

Inspired by Zhang et al. [2019b], we build a deep graph convolutional network based on the undirected graph formed following the above construction steps to further encode the contextual dependencies in the graph. Specifically, for a given undirected graph  $\mathcal{G}_{l_{PAD}} = (\mathcal{V}_{l_{PAD}}, \mathcal{E}_{l_{PAD}})$ , let  $\mathcal{P}$  be the renormalized graph laplacian matrix Kipf and Welling [2016] of  $\mathcal{G}_{l_{PAD}}$ :

$$\begin{aligned} \mathcal{P} &= \hat{\mathcal{D}}^{-1/2} \hat{\mathcal{A}} \hat{\mathcal{D}}^{-1/2} \\ &= (\mathcal{D} + \mathcal{L})^{-1/2} (\mathcal{A} + \mathcal{L}) (\mathcal{D} + \mathcal{L})^{-1/2} \end{aligned} \tag{3.7}$$

where  $\mathcal{A}$  denotes the adjacency matrix,  $\mathcal{D}$  denotes the diagonal degree matrix of the graph  $\mathcal{G}_{l_{PAD}}$ , and  $\mathcal{L}$  denotes the identity matrix. The iteration of GCN through its different layers is formulated as:

$$\mathcal{H}^{(l+1)} = \sigma(((1-\alpha)\mathcal{P}\mathcal{H}^{(l)} + \alpha\mathcal{H}^{(0)}))((1-\beta^{(l)})\mathcal{L} + \beta^{(l)}\mathcal{W}^{(l)})) \quad (3.8)$$

where  $\alpha$  and  $\beta^{(l)}$  are two hyper parameters,  $\sigma$  denotes the activation function and  $\mathcal{W}^{(\dagger)}$  is a learnable weight matrix. Following GCN learning, we use the average embedding of the graph to represent the semantic of structural information in code snippet:

$$w_G = \frac{1}{L} \sum_{i=1}^L (\mathcal{H}_i). \quad (3.9)$$

Thus, at the end of the graph embedding, we obtain representations for  $G_{cbp}$  and  $G_{cap}$ , i.e.,  $w_{G_{cbp}}$  and  $w_{G_{cap}} \in \mathbf{R}^{1 \times d_e}$ .

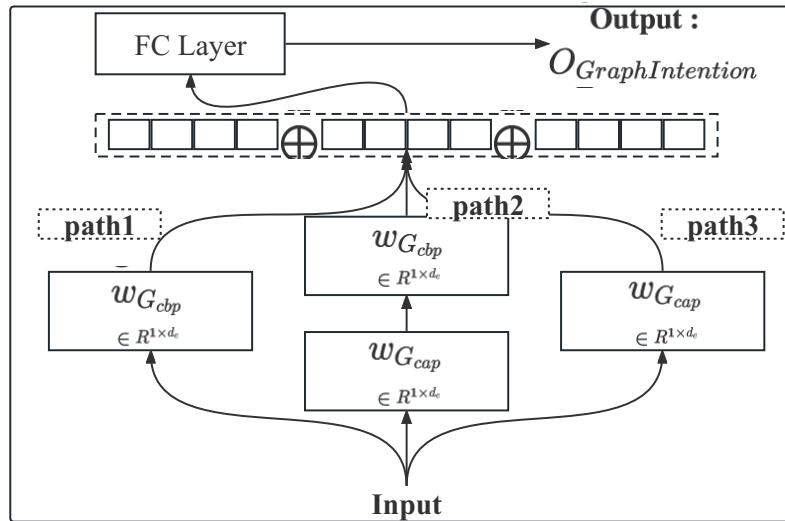


Figure 3.7: graph-cross-resnet architecture.

### 3.2.5.2 Graph Intention Encoding

Once we have computed the embeddings of the code snippets before and after code changeing, (i.e., the embeddings of  $cbp$  and  $cap$ ), we must get the representation of their differences to encode the intention inside the graph changes. To that end, similarly to the previous cross-resnet for sequence intention, we design a **graph-cross-resnet** operator which ensembles the semantic of  $w_{G_{cbp}}$  and  $w_{G_{cap}}$ . Figure 3.7 illustrates this crossing. In this graph-cross-resnet, the model can choose and highlight a path automatically by the backpropagation mechanism. The *GraphIntention* is therefore calculated as follows:

$$\begin{aligned} path1 &= w_{G_{cbp}} \\ path2 &= \mathcal{A} \lceil \lceil (w_{G_{cbp}}, w_{G_{cap}}) \\ path3 &= w_{G_{cap}} \\ O_{GraphIntention} &= \mathcal{FC}(f(\mathcal{A} \lceil \lceil (path1, path2, path3)), \Theta_3) \end{aligned} \quad (3.10)$$

where  $\mathcal{FC}$  is a fully-connected layer and  $\Theta_3$  is the parameter of  $\mathcal{FC}$ .

At the end, the graph-cross-resnet component outputs the sought graph intention embedding: *GraphIntention*.

### 3.2.6 Aggregator: Aggregating Multimodal Input Embeddings

With the sequence intention encoder and the graph intention encoder, we can produce for each code change several embeddings of different input modalities (code sequences and graphs) that must be aggregated into a single representation.

Concretely, we use the  $\mathcal{A}[\cdot]$  aggregation function to merge the *SeqIntention* embeddings (combination of  $O_{cc_p}$ ,  $O_{cc_m}$ ,  $O_{ct2cc_p}$  and  $O_{ct2cc_m}$  - cf. Equations 3.4 and 3.5) and *GraphIntention* embedding  $O_{GraphIntention}$  before outputting the final representation  $E_{Patcherizer}$ . Actually, we use  $E_{Patcherizer}$  as the representation of code change out of the model.

### 3.2.7 Pre-training

Patcherizer is an approach that is agnostic to downstream tasks. We propose to build a pre-trained model using a large corpus of code changes. The objective is to enable the model to learn contextual and structural semantics of code edits, thereby enhancing the quality and robustness of code change representations.

The pre-training task is formulated as masked token prediction. Following the popular bidirectional objective from masked language modeling (MLM; Devlin et al. [2018]), we randomly mask a subset of tokens in the input sequence and train the model to predict these masked tokens using their surrounding context. Formally, this can be expressed as computing the conditional probability  $P(x_i|x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$ .

Inspired by previous pre-training works on code representation learning Feng et al. [2020b], Elnaggar et al. [2021], Zhang et al. [2019c], we adopt the MLM objective for the encoder. Unlike traditional MLM models that rely solely on BERT-style encoders, we employ an encoder-decoder architecture where the decoder is a left-to-right Transformer Vaswani et al. [2017], similar to GPT-style models, which are more suitable for autoregressive generation tasks.

Specifically, we use our proposed Patcherizer encoder to produce latent embeddings of code changes. These embeddings are then passed to a standard Transformer-based decoder that shares the same vocabulary. The decoder autoregressively generates tokens, starting from an initial  $\langle s \rangle$  token, using the following formulation:

$$index = \text{argmax}(p(y_t|y_{t-1}, \dots, y_1, E_{Patcherizer})) \quad (3.11)$$

where  $y_t$  is the token to be predicted at position  $t$ , and  $E_{Patcherizer}$  is the encoded representation from the Patcherizer encoder. The decoder outputs a distribution over the vocabulary, from which we select the token with the highest probability.

### 3.2.8 Fine-tuning for Different Tasks

Patcherizer serves as a task-agnostic encoder that generates semantically rich embeddings of code changes. After pre-training on a large corpus of code edits using the masked token prediction task (cf. Section 3.2.7), we fine-tune Patcherizer on several downstream tasks relevant to software maintenance. These tasks include code change description generation, code change correctness assessment, and code change intention detection.

#### 3.2.8.1 Code Change Description Generation

This task involves generating a natural-language description (e.g., commit message) given a code change. The fine-tuning setup mirrors that of pre-training: we employ an encoder-decoder architecture, where the encoder is the pre-trained Patcherizer and the decoder is a Transformer-based autoregressive generator.

The dataset consists of paired samples of code changes and corresponding natural-language descriptions. During training, we input the code change to the encoder and generate the description token-by-token with the decoder. The learning objective is to maximize the likelihood of generating the correct sequence:

$$\text{index} = \text{argmax}(p(y_t | y_{t-1}, \dots, y_1, E_{\text{Patcherizer}})) \quad (3.12)$$

where  $y_t$  is the target token at time  $t$  and  $E_{\text{Patcherizer}}$  is the encoded representation from the Patcherizer encoder.

### 3.2.8.2 Code Change Correctness Assessment

This task aims to predict whether a given code change is a correct fix for a reported bug. It is formulated as a binary classification problem.

We use the Patcherizer encoder to generate an embedding for the code change and use a separate pre-trained BERT Devlin et al. [2018] model to embed the associated bug report. The two embeddings are concatenated and passed to a fully connected classification layer:

$$\hat{y}_i = \text{sigmoid}(E_{\text{patch}_i} \oplus E_{\text{bugReport}_i}) \quad (3.13)$$

where  $E_{\text{patch}_i}$  is the embedding of the code change,  $E_{\text{bugReport}_i}$  is the embedding of the bug report, and  $\oplus$  denotes concatenation.

We train the classifier using binary cross-entropy loss:

$$\mathcal{L}_a = - \sum_{i=1}^n (y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)) \quad (3.14)$$

where  $y_i$  is the ground truth label and  $\hat{y}_i$  is the predicted probability.

### 3.2.8.3 Code Change Intention Detection

This task focuses on identifying the primary semantic intention behind a given code change, classifying it into categories such as `add`, `remove`, or `update`. While traditional diff tools can highlight superficial syntactic operations, they fail to capture the deeper semantics or purpose of a change. For instance, a change may involve both an addition and removal, but its true intention may be to *disable a feature*, *refactor a loop*, or *resolve a bug*—information that is not directly evident from raw diffs.

Code change intention detection serves several practical use cases. It enables:

- **Automated commit message refinement**, where semantically meaningful summaries can be synthesized.
- **Fine-grained maintenance analytics**, aiding in the analysis of developer behavior and software evolution patterns.
- **Automated patch review assistance**, where intention-aware prioritization of patches can improve reviewer efficiency.

To perform this task, we train a classifier over Patcherizer-generated embeddings, which capture both context-aware sequence semantics and structural graph-based information. Because the embeddings encode deep semantics, no external input beyond the code change is needed for classification.

To visualize the high-dimensional patch embeddings and better understand the distribution of different patch intentions, we employ t-SNE (t-distributed Stochastic Neighbor Embedding) for dimension reduction. The t-SNE algorithm minimizes the Kullback-Leibler divergence between the joint probabilities of the high-dimensional data and the

low-dimensional embedding, effectively preserving the local structure of the data. The t-SNE algorithm can be formulated as:

$$KL(P \parallel Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}} \quad (3.15)$$

where  $P$  represents the pairwise similarities in the high-dimensional space, and  $Q$  represents the pairwise similarities in the low-dimensional space. The probabilities  $p_{ij}$  and  $q_{ij}$  are defined as:

$$p_{ij} = \frac{\exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)}{\sum_{k \neq i} \exp\left(-\frac{\|x_k - x_i\|^2}{2\sigma^2}\right)} \quad (3.16)$$

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l} (1 + \|y_k - y_l\|^2)^{-1}} \quad (3.17)$$

where  $x_i$  and  $x_j$  are high-dimensional data points,  $y_i$  and  $y_j$  are their low-dimensional representations, and  $\sigma$  is the perplexity parameter.

The visualizations and downstream performance demonstrate that Patcherizer effectively distinguishes between different code change intentions. By uncovering the *purpose* behind edits, Patcherizer brings semantic reasoning to automated code analysis pipelines.

## 3.3 Experimental Design

We provide the implementation details (cf. Sec. 3.3.1), discuss the research questions (cf. Sec. 3.3.2), and present the baselines (cf. Sec. 3.3.3), the datasets (cf. Sec. 3.3.4), and the metrics (cf. Sec. 3.3.5).

### 3.3.1 Implementation

In the pre-training phase used for the Sequence Intention Embedding step, we apply a beam search Vijayakumar et al. [2016] for the best performance in predicting the masked words. The beam size was set to 3. The dimension of the hidden layer output in models is set to 512, and the default value of dropout rate is set to 0.1. For the Transformer, we apply 6 heads for the multi-header attention module and 4 layers for the attention.

For the Graph Intention Embedding step, we use javalang Thunes [2013] to parse code fragments and collect ASTs. We build on graph manipulation packages (*i.e.*, networkx net [2018], and dgl Wang et al. [2019a]) to represent these ASTs into graphs.

Patcherizer’s training involves the Adam optimizer Kingma and Ba [2014] with learning rate 0.001. All model parameters are initialized using Xavier algorithm Glorot and Bengio [2010]. All experiments are performed on a server with an Intel(R) Xeon(R) E5-2698 v4 CPU 2.20GHz, 256GB physical memory and one NVIDIA Tesla V100 GPU with 32GB memory.

### 3.3.2 Research questions

**RQ-1:** *How effective is Patcherizer in learning code change representations?*

**RQ-2:** *What is the impact of the key design choices on the performance of Patcherizer?*

**RQ-3:** *To what extent is Patcherizer effective on independent datasets?*

### 3.3.3 Baselines

We consider several SOTA models as baselines. We targeted approaches that were specifically designed for code change representation learning (e.g., CC2Vec) as well as generic techniques (e.g., NMT) that were already applied to code change-related downstream tasks. We finally consider recent SOTA for code change-representation approaches (e.g., FIRA) for specific downstream tasks.

#### For code change description generation

- **NMT** technique has been leveraged by Jiang *et al.* Jiang et al. [2017] for translating code commits into commit messages.
- **CoDiSum** Xu et al. [2019] is an encoder-decoder based model with multi-layer bidirectional GRU and copying mechanism See et al. [2017].
- **ATOM** Liu et al. [2020c] is a commit message generation techniques, which builds on abstract syntax tree and hybrid ranking.
- **FIRA** Dong et al. [2022] is a graph-based code change representation learning approach for commit message generation.
- **Coregen** Nie et al. [2021] is a **pure Transformer-based** approach for representation learning targeting commit message generation.
- **CCRep** Liu et al. [2023] is an innovative approach that uses pre-trained models to encode code changes into feature vectors, enhancing performance in tasks like commit message generation, etc.
- **CC2Vec** Hoang et al. [2020] learns a representation of code changes guided by commit messages. It is the incumbent state of the art that we aim to outperform on all tasks.
- **NNGen** Liu et al. [2018] is an IR-based commit message prediction technique.

- **CoRec** Wang et al. [2021a] is a retrieval-based context-aware encoder-decoder model for commit message generation.
- **CCBERT Zhou et al. [2023c]** learns fine-grained code change representations, outperforming CC2Vec and CodeBERT in efficiency and accuracy.
- **CCT5 Lin et al. [2023]** automates software maintenance by leveraging code changes and commit messages, outperforming traditional models.
- **CodeT5 Wang et al. [2021d]** uses identifier-aware tasks to enhance code understanding and generation, outperforming prior methods.

#### For Code Change Correctness Assessment

- **CC2Vec** Hoang et al. [2020] and **CCRep** Liu et al. [2023].
- **BERT** Devlin et al. [2018] is a state of the art unsupervised learning based Transformer model widely used for text processing.

#### For Code Change Intention Detection

- **CCRep** Liu et al. [2023] and **CC2Vec** Hoang et al. [2020].

#### For Just-in-Time Defect Prediction

- **CC2Vec** Hoang et al. [2020] is a state-of-the-art approach for generating code change embeddings used in defect prediction.
- **DeepJIT** Zhang and Wallace [2015] is a deep learning model that predicts whether a commit introduces a defect based on its code changes and commit messages.
- **CCRep** Liu et al. [2023] demonstrates effectiveness on this task by generating code change embeddings that capture semantic information.

### 3.3.4 Datasets

**Code Change description generation:** We build on prior benchmarks Dyer et al. [2013], Hoang et al. [2020], Liu et al. [2018], Dong et al. [2022] by focusing on Java samples and reconstructing snippets to make them parsable for AST collection. Eventually, our dataset includes 90,661 code changes and their associated descriptions.

**Code change correctness assessment:** We leverage the largest dataset in the literature to date, which includes deduplicated 11,352 code changes (9,092 Incorrect and 2,260 Correct) released by Tian *et al.* Tian et al. [2022c].

**Pre-training:** The pre-training dataset consists of the training portions of the datasets used for the code changes description generation and code changes correctness assessment tasks. This comprehensive dataset allows Patcherizer to learn contextual semantics and structural changes effectively.

**Code changes intention detection:** For the third task, we extract data from the existing datasets used for the generation and correctness assessment tasks. Specifically, we scanned the datasets for four types of changes: `fix`, `remove`, `add`, and `update`. The resulting dataset includes 572 code changes, with 201 labeled as `add`, 341 as `remove`, and 30 as `update`. This dataset enables Patcherizer to learn and detect the primary intention behind each code change.

### 3.3.5 Metrics

To evaluate code change intention detection, we report standard multi-class classification metrics: Precision, Recall, and F1-score, averaged across categories. The dataset was labeled using a curated taxonomy of intentions (e.g., `add`, `remove`, `update`), applied to a subset of patches reviewed by two human annotators with 90% agreement. The class distribution is moderately imbalanced, with `add` comprising 42%, `update` 36%, and `remove` 22% of samples.

**ROUGE** ROUGE [2004] is one set of metrics for comparing automatic generated text against the reference (human-produced) ones. We focus on ROUGE-L which computes the Longest Common Subsequence.

**BLEU** Papineni et al. [2002] is a classical metric to evaluate the quality of machine translations. It measures how many word sequences from the reference text occur in the generated text and uses a (slightly modified) n-gram precision to generate a score.

**METEOR** Banerjee and Lavie [2005] is an F-Score-Oriented metric for measuring the performance of text-generation models.

**+Recall** and **-Recall** are specific metrics for code changes correctness assessment Tian et al. [2022c]. +Recall (-Recall) measures to what extent correct (incorrect) code changes can be predicted (filtered out).

**Metrics for code changes description generation** ①: ROUGE-L; ②: BLEU; ③: METEOR.

**Metrics for code changes correctness assessment** ①: AUC; ②: F1; ③: +Recall; ④: -Recall.

**Metrics for code changes intention detection** ①: **Precision**: measures the proportion of correctly predicted intentions among all predicted intentions; ②: **Recall**: measures the proportion of correctly predicted intentions among all actual intentions; ③: **F1-score**: the harmonic mean of precision and recall, providing a balanced measure of the classifier's performance.

## 3.4 Experimental Results

### 3.4.1 [RQ-1]: Performance of Patcherizer

**Goal.** The first research question investigates whether Patcherizer’s learned representations are expressive enough to support multiple downstream software engineering tasks. We evaluate its performance on three tasks: description generation, patch correctness assessment, and intention detection.

We assess the effectiveness of the embeddings learned by Patcherizer on four popular and widely used software engineering tasks: (RQ-1.1) Code change description generation, (RQ-1.2) Code change correctness assessment, (RQ-1.3) Code change intention detection, (RQ-1.4) Just-in-Time defect prediction. We compare Patcherizer against the relevant SOTA.

#### 3.4.1.1 RQ-1.1: Code Change Description Generation

##### [Experiment Design]:

We employ the dataset from FIRA. As Dong et al. Dong et al. [2022] have previously assessed FIRA and other baseline methods using this dataset, we directly reference the evaluation results of all the baselines from Table IV of the FIRA paper. The dataset contains 75K, 8K and 7.6K commit-message pairs in the training, validation and test sets, respectively.

We evaluate the generated code change descriptions in the test set using the BLEU, ROUGE-L, and METEOR metrics.

Note that we distinguish between baseline generation-based methods and retrieval-based ones. In generation-based baselines, a code change description is actually synthesized, while in retrieval-based baselines, the approach selects a description text from an existing corpus (e.g., in the training set). For fairness, we build two distinct methods using Patcherizer’s embeddings. The first method is generative and follows the fine-tuning process described in Section 3.2.8. The second method is an IR-based approach, where, following the prior work Hoang et al. [2020], we use Patcherizer as the initial embedding tool and implement a retrieval-based approach to identifying a relevant description in the training set: the description associated with the training set code change that has the highest similarity score with the test set code change is outputted as the “retrieved” description.

**[Experiment Results]:** Table 3.1 presents the average scores of the different metrics with the descriptions generated by Patcherizer and the relevant baselines. Patcherizer outperforms all the compared techniques on all metrics, with the exception of FIRA on the ROUGE-L metric. The superior performance of Patcherizer on generation-based and retrieval-based methods, as illustrated by the distribution of BLEU scores in Figure 3.8, further suggest that the produced embeddings are indeed effective.

In Figure 3.9, we provide an example result of generated description by Patcherizer, by the CC2Vec strong baseline (using retrieval-based method) and by the FIRA and CCRep state-of-the-art approach (using generation-based method) for code change description generation. Patcherizer succeeds in actually generating the exact description as the ground truth commit message, after taking into account both sequential and structural information. By observing the graph intention and sequence intention, we can see that the model found that the only change is that the node `true` has been changed/updated/disabled to `false`. Finally, the sequence intention embedding would make Patcherizer recognize that the carrier of `true` and `false` is `RenderThread` based on BPE splitting.

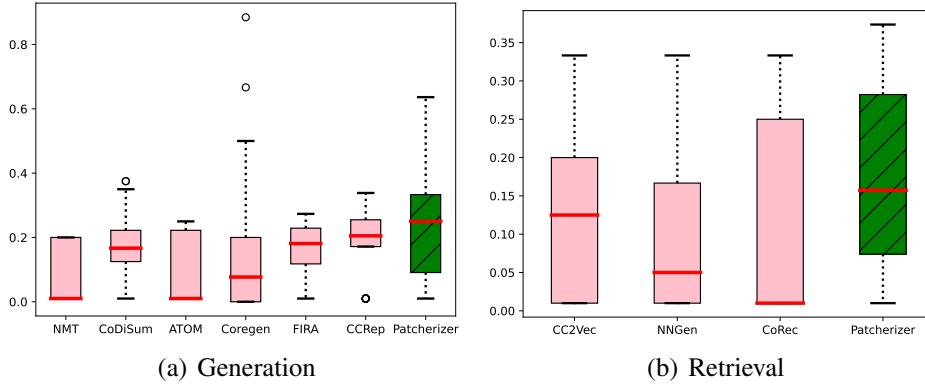
**[Human Evaluation]:** To further assess the quality of the generated code change descriptions from the perspective of developers, we conducted a human evaluation study to compare Patcherizer with leading techniques. Specifically, we compare Patcherizer against

**Table 3.1:** Performance Results of code change description generation.

Type	Approach	Rouge-L (%)	BLEU (%)	METEOR (%)
Generation	NMT Jiang et al. [2017]	7.35	8.01	7.93
	Codisum Xu et al. [2019]	19.73	16.55	12.83
	ATOM Liu et al. [2020c]	10.17	8.35	8.73
	FIRA Dong et al. [2022]	21.58	17.67	14.93
	CoreGen Nie et al. [2021] (Transformer)	18.22	14.15	12.90
	CCRep Liu et al. [2023]	23.41	19.70	15.84
	CCBERT Zhou et al. [2023c]	20.74	16.98	14.25
	CCT5 Lin et al. [2023]	21.13	17.11	14.38
Retrieval	CodeT5 Wang et al. [2021d]	21.26	17.33	14.52
	Patcherizer	25.45	23.52	21.23
	CC2Vec Hoang et al. [2020]	12.21	12.25	11.21
Retrieval	NNGen Liu et al. [2018]	9.16	9.53	16.56
	CoRec Wang et al. [2021a]	15.47	13.03	12.04
	Patcherizer	17.32	15.21	17.25

“**Generation**” for generation-based strategy. Given fragments of codes, “**Generation**” methods generate messages from scratch.

“**Retrieval**” for retrieval-based approaches. Given fragments of codes, “**Retrieval**” approaches return the most similar message from the training dataset.


**Figure 3.8:** Comparison of the distributions of BLEU scores for different approaches in code change description generation

the retrieval-based technique NNGen, the learning-based technique CODISUM, and the FIRA Dong et al. [2022] approach. Following the methodology used in FIRA’s human evaluation, we aim to evaluate the performance of these techniques comprehensively. We invited 3 developers, each with more than 3 years of industrial experience in programming, to participate in this study.

**Table 3.2:** Scoring Criteria Dong et al. [2022]

Score	Definition
0	Neither relevant in semantics nor having shared tokens.
1	Irrelevant in semantics but with some shared tokens.
2	Partially similar in semantics, with exclusive information.
3	Highly similar but not identical in semantics.
4	Identical in semantics.

**Study Design:** Following established practices Dyer et al. [2013], Hoang et al. [2020], Dong et al. [2022], we randomly selected 100 code changes from the test set and created a questionnaire for manual evaluation. Each questionnaire contained the code change, the ground truth code change description, and the descriptions generated by Patcherizer, NNGen,

CODISUM, and FIRA. The participants were asked to score the generated descriptions on a scale from 0 to 4, where a higher score indicates a higher similarity to the ground truth. To ensure unbiased evaluation, the techniques were anonymized in the questionnaire, and each participant completed the evaluation independently.

**Table 3.3:** Human Evaluation Results

Model	Low (%)	Medium (%)	High (%)	Average Score
NNGen	70.5	15.3	14.2	0.96
CODISUM	37.6	21.4	41.0	2.03
FIRA	34.0	21.8	44.2	2.12
Patcherizer	32.8	20.5	46.7	2.19

**Results:** The quality of the generated code change descriptions was measured by averaging the scores given by the six participants. Similar to prior studies Dyer et al. [2013], Hoang et al. [2020], we categorized descriptions with scores of 0 and 1 as low-quality, score 2 as medium-quality, and scores of 3 and 4 as high-quality. Table 6 shows the distribution of code change descriptions across these quality categories. As indicated in the table, Patcherizer generated the highest proportion of high-quality descriptions (46.7%) and the lowest proportion of low-quality descriptions (32.8%). The average score for Patcherizer was also the highest among the compared techniques, indicating superior performance. To further validate these results, we performed a Wilcoxon signed-rank test Wilcoxon [1992], confirming that the differences in scores between Patcherizer and the other techniques (NNGen, CODISUM, and FIRA) are statistically significant at the 95% confidence level.

◀ **Answer to RQ-1.1:** ▶ *Patcherizer’s embeddings are effective for code change description generation yielding the best scores for BLEU, ROUGE-L, and METEOR metrics.* ◀

### 3.4.1.2 RQ-1.2: Code Change Correctness Assessment

**[Experiment Design]:** Tian et al. Tian et al. [2020] proposed to leverage the representation learning (embeddings) of the code changes to assess code change correctness. Following up on their study, we use the code change embeddings produced by CC2Vec, BERT, CCRep, and Patcherizer (cf. Section 3.2.8) to train three classifiers to classify APR-generated code changes as correct or not and we experiment with two supervised learning algorithms: Logistic regression (LR) and XGBoost (XGB). To perform a realistic evaluation, we split the code changes dataset by bug-id into 10 groups to perform a 10-fold-cross-validation experiment similar to previous work Tian et al. [2022c]. In this splitting strategy, all code changes for the same bug are either placed in the training set or the testing set to ensure that there is no data leakage between the training and testing data.

We then measure the performance of the classifiers using +Recall, -Recall, AUC, and F1.

**[Experiment Results]:** Table 3.4 shows the results of this experiment. Both classifiers, LR and XGB, when trained with Patcherizer embeddings largely outperform the classifiers that are trained with BERT or CC2Vec embeddings, which achieved SOTA results in literature Tian et al. [2020].

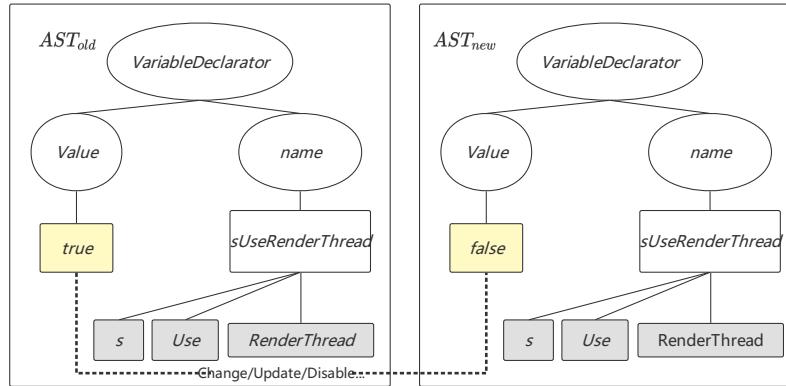
◀ **Answer to RQ-1.2:** ▶

*Code change embeddings generated by Patcherizer achieve SOTA results in the task of code change correctness assessment, largely outperforming SOTA embedding models.* ◀

```

1  s0.java 20220820 17:00:26.000000000 +0200
2  +++ t0.java 20220820 17:01:04.000000000 +0200
3  @@ 1,6 1,6 @@
4      public abstract class HardwareRenderer {
5          public static boolean sSystemRendererDisabled = false ;
6          - public static boolean sUseRenderThread = true ;
7          public static boolean sUseRenderThread = false ;
8          private boolean mEnabled ;
9          private boolean mRequested = true ;
10     }

```



Source      Code change description

**Ground truth**    **Disable RenderThread**

CC2Vec	Fix test data so that it can be compiled
FIRA	fix the in
CCRep	update suserenderthread
Patcherizer	disable renderthread

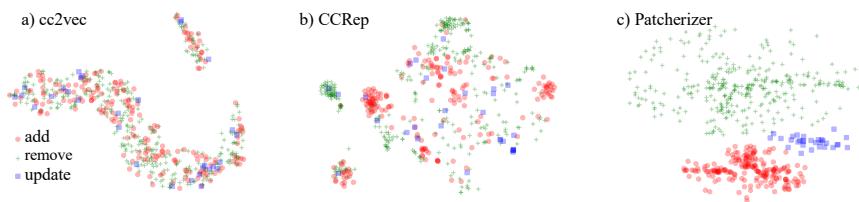
**Figure 3.9:** Illustrative example of code change description generation

**Table 3.4:** Performance of Code Change Correctness Assessment

Classifier	Model	AUC	F1	+Recall	-Recall
LR	CC2Vec	0.75	0.49	0.47	0.85
	BERT	0.83	0.58	0.81	0.65
	CCRep	0.86	0.67	0.74	0.83
	Patcherizer	0.96	0.82	0.87	0.91
XGB	CC2Vec	0.81	0.55	0.50	0.89
	BERT	0.84	0.61	0.64	0.85
	CCRep	0.82	0.63	0.59	0.88
	Patcherizer	0.90	0.67	0.66	0.90

### 3.4.1.3 RQ-1.3: Code Change Intention Detection

**[Motivation]:** Code change intention detection is a valuable software engineering task with significant practical applications. Prior research by Buse and Weimer Buse and Weimer [2010] and Cortés-Coy et al. Cortés-Coy et al. [2014] established that understanding the semantics and intentions behind code changes enhances developer productivity and supports software maintenance activities. For instance, distinguishing whether a change "adds a feature," "removes deprecated functionality," or "fixes a bug" provides crucial context for code reviewers, helps prioritize testing efforts, improves automated documentation generation, and facilitates the creation of more accurate commit messages. As noted by Tao et al. ?, developers spend considerable time understanding the rationale behind code changes, making automatic intention detection a high-impact task. Moreover, accurate intention



**Figure 3.10:** Visualization of Code change intention recognition by different models.

classification serves as a foundation for higher-level reasoning about software evolution patterns Hindle et al. [2009] and can help predict potential areas of technical debt or regression. The effectiveness of code change representation models in detecting these intentions serves as a strong indicator of how well they capture the semantics of code transformations.

**[Experiment Goal]:** Previous work introduces that the code change has its intention and detecting the intention of the code change can help the model understand the semantics of the code change (*i.e.*, template-based works Buse and Weimer [2010], Cortés-Coy et al. [2014] and generation-based works Dong et al. [2022], Xu et al. [2019]). Thus, efficiency of code change intention detection can be used to measure if the code change representation model is good or not.

**[Experiment Results]:**

We scan all words across two datasets in our work and figure out that code changes are mainly related to four types: `fix`, `remove`, `add`, and `update`. However, `fix` is highly related to all other three frequent words, because `fix` can be used to update, remove or add. Therefore, we select `add`, `remove`, `update` as our main detected intentions. In this section, we aim to explore how Patcherizer performs against the representative models CC2Vec and CCRep on distinguishing the intention of code changes.

We trained the three models (*i.e.*, Patcherizer, CC2Vec, and CCRep) on a large dataset proposed in Dong et al. [2022]. Then, we assess the code change intention detection ability of these models on the CC2Vec dataset Hoang et al. [2020].

We find that 572 code changes contain `add`, `remove`, or `update` keywords (*i.e.*, 201 for `add`, 341 for `remove`, 30 for `update`). Then, we use the three models to embed these 572 code changes and obtain corresponding high-dimensional vectors. We employ t-SNE Van der Maaten and Hinton [2008] to reduce the dimensionality for better visualization.

Figure 3.10 shows the t-SNE visualized results of CC2Vec, CCRep and Patcherizer.

The red color represents `add` function, the green color represents `remove` function, and the blue color represents `update` function. We see that Patcherizer separates `add` and `remove` better than CC2Vec and CCRep. Furthermore, both CC2Vec and CCRep fail to separate `update` from the other two functions. The reason may be that `update` functions can be `add` or `remove` functions. Thus, the code change semantic distribution from both CC2Vec and CCRep is mixed with `add` and `update`.

**Answer to RQ-1.3:** ▶ Compared with existing code change representation models, Patcherizer is more effective in detecting the intention of code changes. ◀

#### 3.4.1.4 RQ-1.4: Just-in-Time Defect Prediction

**[Experiment Design]:** Following prior work Hoang et al. [2020], we integrate Patcherizer with DeepJIT Zhang and Wallace [2015]. Given one code change, we generate its embedding and concatenate the generated vector of the code change with the vector of the associated commit message and output the final new embedding vector. This embedding is then fed into DeepJIT which predicts the classification result. Unfortunately, the datasets

**Table 3.5:** AUC (%) Results on JIT defect prediction on QT and OPENSTACK datasets.

Model	QT	OPENSTACK
DeepJIT	76.8	75.1
CC2Vec	82.2	80.9
CCRep		76.45
Patcherizer <sub>diff<sub>AST</sub>-</sub>	84.5	82.3

(QT and OPENSTACK) are not parseable to retrieve ASTs. Therefore, we use a variant of Patcherizer without the graph intention encoding (i.e., Patcherizer<sub>GraphIntention-</sub>). We use 5-fold cross-validation for the evaluation. The metric to evaluate JIT defect prediction is AUC, and the relevant baseline is CC2Vec.

**[Experiment Results]:** The classification performance are depicted in Table 3.5. Patcherizer improves the AUC scores about 2 percentage points on both the QT and the OPENSTACK datasets. Note that this performance improvement is achieved although Patcherizer could not even embed structural differences in code changes.

**Answer to RQ-1.4:** ▶ The experimental results show that the embeddings produced by Patcherizer<sub>diff<sub>AST</sub>-</sub> are effective for the just-in-time defect prediction task. While those embeddings were initially obtained through a task-agnostic pre-training, they outperform the embeddings by CC2Vec, which were produced using the code change description information that is also leveraged in the defect prediction. ◀

### 3.4.2 [RQ-2]: Ablation Study

**Goal.** The second research question evaluates the effectiveness of Patcherizer’s two modality-specific encoders: the Sequence Intention Encoder and Graph Intention Encoder. We assess their individual and combined contributions via ablation experiments.

#### 3.4.2.1 Code Change Description Generation

**[Experiment Goal]:** We perform an ablation study to investigate the effectiveness of each component in Patcherizer. The major novelty of Patcherizer is the fact that it explicitly includes and processes: ① *SqIntention* represents intention embedding of the code change at the sequential level, and ② *GraphIntention* represents intention embedding of the code change at the structural level.

**[Experiment Design]:** We investigate the related contribution of *SqIntention* and *GraphIntention* by building two variants of Patcherizer where we remove either *GraphIntention* (i.e., denoted as Patcherizer<sub>GraphIntention-</sub>), or *SqIntention* (i.e., denoted as Patcherizer<sub>SqIntention-</sub>). We also build a native model by removing both *GraphIntention* and *SqIntention* components (i.e., denoted as Patcherizer<sub>both-</sub> for comparison. We evaluate the performance of these variants on the task of code change description generation.

**[Experiment Results]:** Table 3.6 summarizes the results of our ablation test on the three variants of Patcherizer.

While the performance of Patcherizer is not the simple addition of the performance of each variant, we note with Patcherizer<sub>both-</sub> that the performance is quasi-insignificant, which means that, put together, both design choices are instrumental for the superior performance of Patcherizer.

*Contribution of Graph Intention Encoding:*

We observe that the graph intention embedding significantly improves the model ability

**Table 3.6:** Ablation study results based on the code change description generation task.

Model	ROUGE-L (%)	BLEU (%)	METEOR (%)
Patcherizer <sub>GraphIntention-</sub>	20.10	16.50	15.40
Patcherizer <sub>SeqIntention-</sub>	18.44	14.70	16.20
Patcherizer <sub>both-</sub>	15.00	13.00	12.00
Patcherizer	25.45	23.52	21.23

to generate correct code change descriptions for more code changes which is evidenced by the large improvement on the ROUGE-L score (from 20.10 to 25.45), where ROUGE-L is recall oriented.

We postulate that even when token sequences (e.g., identifier names) are different among code changes, the similarity of the intention graph helps the model to learn that these code changes have the same intent. Nevertheless, precision in description generation (i.e., how many words are correct) is highly dependent on the model’s ability to generate the exact correct tokens, which is more guaranteed by the context and sequence intention embedding.

We manually checked different samples to analyze how the variants were performing. Figure 3.11 presents a real-world case in our dataset, including the patch, the ground truth, and the code change descriptions generated by Patcherizer, Patcherizer<sub>GraphIntention-</sub>, Patcherizer<sub>SeqIntention-</sub>, as well as three of the strongest baselines for this task (i.e., CC2Vec, FIRA, and CCRep). In this case, the embeddings of Patcherizer and Patcherizer<sub>GraphIntention-</sub> are effective in spotting the sub-token *trident* in class name *TridentTopologyBuilder* thanks to BPE. In addition, Patcherizer takes advantage of both the sequence intention and graph intention inside the patch. However, if we only consider the graph intention, Patcherizer<sub>SeqIntention-</sub> performs the worst against Patcherizer<sub>GraphIntention-</sub>. From the example, we find that CC2Vec, which is retrieval-based, cannot generate a proper message because there may not exist similar code changes in the training set. FIRA, while underperforming against Patcherizer, still performs relatively well because it uses the edition operation detector and sequential contextual information.

```

1  a/src/ [...] /topology/TridentTopologyBuilder.java
2  +++ b/src/ [...] /topology/TridentTopologyBuilder.java
3  public class TridentTopologyBuilder {
4      bd.allGrouping( masterCoordinator( batchGroup ) ,
5          ↳ MasterBatchCoordinator.
6          COMMIT_STREAM_ID );
7      for( Map m : c.componentConfs ) {
8          scd . addConfigurations ( m ) ;
9          bd . addConfigurations ( m ) ;
10     }
11 }
```

Source	Code change description
<b>Ground truth</b>	<b>set component configurations correctly for trident spouts</b>
Patcherizer	set component configurations correctly for trident spouts
Patcherizer <sub>GraphIntention-</sub>	configure components for trident
Patcherizer <sub>SeqIntention-</sub>	set trident components.
CC2Vec	fixed flickering in the preview pane in refactoring preview
FIRA	use the correct component content in onesidediffviewer
CCRep	update for function

**Figure 3.11:** Case analysis of the ablation study

It is noteworthy that Patcherizer is able to generate the token *spouts*. This is not due to

data leakage since the ground truth commit message was not part of the training set. However, our approach builds on a dictionary that considers all tokens in the dataset (just as the entire English dictionary would be considered in text generation). Hence *spouts* was predicted from the dictionary as the most probable (using *softmax*) token to generate after *trident*.

### 3.4.2.2 Code Change Correctness Assessment

**[Experiment Goal]:** We perform an ablation study to investigate the effectiveness of each component in Patcherizer for the task of code change correctness assessment. The major novelty of Patcherizer lies in its ability to process: ① *SeqIntention*, which represents the intention embedding of the code change at the sequential level, and ② *GraphIntention*, which represents the intention embedding of the code change at the structural level.

**[Experiment Design]:** To understand the contribution of *SeqIntention* and *GraphIntention*, we build two variants of Patcherizer: one by removing *GraphIntention* (denoted as Patcherizer<sub>*GraphIntention*–</sub>) and another by removing *SeqIntention* (denoted as Patcherizer<sub>*SeqIntention*–</sub>). Additionally, we create a baseline model by removing both components (denoted as Patcherizer<sub>*both*–</sub>). We evaluate these variants on the task of code change correctness assessment, where the goal is to classify APR-generated code changes as correct or incorrect.

Following Tian et al. [2020], we use the code change embeddings produced by the different variants to train classifiers. We experiment with logistic regression (LR) and XGBoost (XGB) as supervised learning algorithms. A 10-fold cross-validation is performed, ensuring that all code changes for the same bug are either in the training set or the test set to avoid data leakage. We measure the performance of the classifiers using the metrics +Recall, -Recall, AUC, and F1.

**Table 3.7:** Performance of Code Change Correctness Assessment

Classifier	Model	AUC	F1	+Recall	-Recall
LR	CC2Vec	0.75	0.49	0.47	0.85
	BERT	0.83	0.58	0.81	0.65
	CCRep	0.86	0.67	0.74	0.83
	Patcherizer	0.96	0.82	0.87	0.91
	Patcherizer <sub><i>GraphIntention</i>–</sub>	0.88	0.70	0.80	0.78
	Patcherizer <sub><i>SeqIntention</i>–</sub>	0.84	0.62	0.75	0.72
XGB	CC2Vec	0.81	0.55	0.50	0.89
	BERT	0.84	0.61	0.64	0.85
	CCRep	0.82	0.63	0.59	0.88
	Patcherizer	0.90	0.67	0.66	0.90
	Patcherizer <sub><i>GraphIntention</i>–</sub>	0.86	0.64	0.68	0.82
	Patcherizer <sub><i>SeqIntention</i>–</sub>	0.83	0.60	0.65	0.80

The results indicate that the full Patcherizer model outperforms its variants and baselines, showing the combined importance of both *SeqIntention* and *GraphIntention*.

**Contribution of Graph Intention Encoding:** Removing the graph intention embedding (Patcherizer<sub>*GraphIntention*–</sub>) leads to a noticeable decrease in performance, particularly in AUC and F1 scores, suggesting that structural information is crucial for accurate code change correctness assessment.

**Contribution of Sequence Intention Encoding:** Similarly, removing the sequence intention embedding (Patcherizer<sub>*SeqIntention*–</sub>) also reduces the effectiveness of the model, highlighting the importance of capturing the sequential context of code changes.

*Combined Effect:* The combined effect of both *SeqIntention* and *GraphIntention* in the full Patcherizer model results in the best performance, indicating that both components are necessary for achieving state-of-the-art results.

### 3.4.2.3 Code Change Intention Detection

**[Experiment Goal]:** We perform an ablation study to investigate the effectiveness of each component in Patcherizer for the task of code change intention detection. We explore how the major components of Patcherizer—❶ *SeqIntention*, which represents the intention embedding at the sequential level, and ❷ *GraphIntention*, which represents the intention embedding at the structural level—contribute to the model’s ability to detect and differentiate code change intentions.

**[Experiment Design]:** Following the methodology used in our main experiment, we investigate the individual contributions of *SeqIntention* and *GraphIntention* by creating variants of Patcherizer where one component is removed: Patcherizer<sub>*GraphIntention*–</sub> (without graph intention encoding) and Patcherizer<sub>*SeqIntention*–</sub> (without sequence intention encoding). We also include a baseline variant Patcherizer<sub>*both*–</sub> that removes both components. We train these variants on the same large dataset used in Section 3.2.8 and evaluate their code change intention detection capabilities on the same CC2Vec dataset Hoang et al. [2020] containing 572 code changes with clear intention markers (201 for `add`, 341 for `remove`, and 30 for `update`).

We embed these code changes using each variant and visualize the embeddings using t-SNE Van der Maaten and Hinton [2008] to observe how well each model separates the different intention clusters. Additionally, we quantitatively evaluate the separation by calculating silhouette scores and performing k-means clustering to measure the accuracy of intention classification.

#### [Experiment Results]:

**Table 3.8:** Intention Classification Performance of Different Patcherizer Variants

Model	Silhouette Score	Clustering Accuracy (%)
Patcherizer	<b>0.42</b>	<b>81.3</b>
Patcherizer <sub><i>GraphIntention</i>–</sub>	0.31	72.6
Patcherizer <sub><i>SeqIntention</i>–</sub>	0.25	65.4
Patcherizer <sub><i>both</i>–</sub>	0.17	58.2

Table 3.8 presents the quantitative results. The full Patcherizer model achieves the highest silhouette score (0.42) and clustering accuracy (81.3%), indicating superior separation of intention types. The variant without graph intention encoding (Patcherizer<sub>*GraphIntention*–</sub>) shows a noticeable drop in performance, with the silhouette score decreasing to 0.31 and accuracy to 72.6%. The variant without sequence intention encoding (Patcherizer<sub>*SeqIntention*–</sub>) performs even worse, with a silhouette score of 0.25 and accuracy of 65.4%. The baseline variant without both components (Patcherizer<sub>*both*–</sub>) shows the poorest performance, with a silhouette score of only 0.17 and accuracy of 58.2%.

*Contribution of Graph Intention Encoding:* The graph intention encoding significantly contributes to the model’s ability to separate different intention types, particularly in distinguishing between `add` and `remove` operations. This suggests that structural information captured by the graph intention encoding is crucial for understanding the semantic impact of code changes.

*Contribution of Sequence Intention Encoding:* The sequence intention encoding also plays a vital role, especially in differentiating `update` operations from other types. Without

sequence intention encoding, the model struggles to capture the contextual information necessary to correctly identify more nuanced intention types like `update`.

*Combined Effect:* The experimental results clearly demonstrate that both components complement each other, with their combination resulting in the most effective intention detection capability. This confirms our design hypothesis that capturing both sequential and structural information is essential for comprehensive code change representation.

#### 3.4.2.4 Just-in-Time Defect Prediction

**[Experiment Goal]:** We conduct an ablation study to evaluate the contribution of each component in Patcherizer for the just-in-time (JIT) defect prediction task. This analysis focuses on understanding how the two key components of Patcherizer—the *SeqIntention* encoding and the *GraphIntention* encoding—affect its performance in identifying defective patches.

**[Experiment Design]:** Following the methodology used in our main experiments, we create variants of Patcherizer by removing one component at a time: Patcherizer<sub>*GraphIntention*–</sub> (without graph intention encoding) and Patcherizer<sub>*SeqIntention*–</sub> (without sequence intention encoding). We also include a baseline variant Patcherizer<sub>*both*–</sub> that removes both components. Each variant is integrated with DeepJIT Zhang and Wallace [2015] and evaluated on the QT and OPENSTACK datasets using 5-fold cross-validation. Note that since the original datasets (QT and OPENSTACK) are not parseable to retrieve ASTs, the full Patcherizer model is already using a variant without the graph intention encoding in the main experiment. Therefore, Patcherizer<sub>*GraphIntention*–</sub> is equivalent to the full model in this specific case, and we are primarily testing the contribution of the *SeqIntention* component.

**Table 3.9:** AUC Scores for JIT Defect Prediction with Different Patcherizer Variants

Model	QT (%)	OPENSTACK (%)
CC2Vec	73.43	63.77
Patcherizer <sub><i>GraphIntention</i>–</sub>	<b>75.73</b>	<b>65.50</b>
Patcherizer <sub><i>SeqIntention</i>–</sub>	74.21	64.32
Patcherizer <sub><i>both</i>–</sub>	72.86	63.25

**[Experiment Results]:** Table 3.9 presents the AUC scores achieved by different variants of Patcherizer on the JIT defect prediction task. The full Patcherizer model (which, in this case, is equivalent to Patcherizer<sub>*GraphIntention*–</sub> due to dataset limitations) achieves the highest AUC scores on both datasets: 75.73% on QT and 65.50% on OPENSTACK. Removing the sequence intention encoding (Patcherizer<sub>*SeqIntention*–</sub>) results in a performance drop, with AUC scores decreasing to 74.21% on QT and 64.32% on OPENSTACK. The baseline variant without both components (Patcherizer<sub>*both*–</sub>) performs even worse, with AUC scores of 72.86% on QT and 63.25% on OPENSTACK, even lower than the CC2Vec baseline on the QT dataset.

*Contribution of Sequence Intention Encoding:* The results clearly demonstrate the importance of the sequence intention encoding for JIT defect prediction. Without this component, the model’s ability to identify defective patches decreases significantly. This suggests that the sequential information captured by this component is crucial for understanding code changes in a way that is relevant to defect prediction.

#### 3.4.2.5 Ablation Study on Sequence Intention Encoder Components

**[Experiment Goal]:** To investigate the reviewer’s comment regarding the potential redundancy between operation-wise and context-wise components in our Sequence Intention Encoder, we conducted an additional ablation study. While both components process in-

formation from code changes, we hypothesized that they capture complementary aspects that together improve representation quality.

**[Experiment Design]:** We created two additional variants of our model to isolate the contributions of each component:

1. *Patcherizer<sub>OperationWise-</sub>* (removing only the operation-wise component while keeping context-wise)
2. *Patcherizer<sub>ContextWise-</sub>* (removing only the context-wise component while keeping operation-wise)

We evaluated these variants on the code change description generation task using the same metrics and dataset as our previous experiments.

**[Experiment Results]:** Table 3.10 presents the results of this extended ablation study.

**Table 3.10:** Ablation study of Sequence Intention Encoder components on code change description generation.

Model	ROUGE-L (%)	BLEU (%)	METEOR (%)
Patcherizer (Full)	25.45	23.52	21.23
Patcherizer <sub>OperationWise-</sub>	21.87	18.93	17.46
Patcherizer <sub>ContextWise-</sub>	22.65	19.76	18.35
Patcherizer <sub>both-</sub>	15.00	13.00	12.00

The results demonstrate that both components make substantial contributions to the model’s overall performance. Removing the operation-wise component (Patcherizer<sub>OperationWise-</sub>) leads to a significant drop in BLEU score from 23.52% to 18.93%, indicating its importance for precise description generation. Similarly, removing the context-wise component (Patcherizer<sub>ContextWise-</sub>) reduces the ROUGE-L score from 25.45% to 22.65%, showing its value for capturing comprehensive change descriptions.

**[Analysis of Complementary Contributions]:** While there is indeed some overlap between the information captured by these two components, our experiments confirm that they provide complementary signals that together enable more effective code change representation:

1. **Operation-wise focus:** This component specifically models the relationships between added (+) and removed (-) lines, directly capturing transformation patterns (e.g., parameter additions, condition changes, variable renaming). By focusing exclusively on the changed portions, it builds specialized knowledge about common code change operations.
2. **Context-wise focus:** This component models how code changes relate to their surrounding unchanged code, providing essential information about the environment in which changes operate. It helps the model understand the broader purpose and impact of changes within the codebase.

Figure 3.12 illustrates a case where both components contribute unique insights. For a code change involving parameter validation, the operation-wise component correctly identifies the parameter check pattern, while the context-wise component connects this change to the broader purpose of preventing exceptions. Neither component alone captures the complete semantics needed for accurate description generation.

```

1  a/src/main/java/org/example/validator/RequestValidator.
2    ↪ java
3  +++ b/src/main/java/org/example/validator/
4    ↪ RequestValidator.java
5  @@ 120,7 120,9 @@ public class RequestValidator {
6      * @throws ValidationException if the request is
7      ↪ invalid
8      */
9  public void validateUserRequest(UserRequest request)
10     ↪ throws ValidationException {
11     -   validateRequestParameters(request.getParameters())
12     ↪ ;
13     if (request.getParameters() != null) {
14         validateRequestParameters(request.getParameters
15     ↪ ());
16     }
17     // continue with other validation steps
18     validateRequestHeaders(request.getHeaders());
19     logValidationSuccess(request.getId());

```

Source	Code change description
<b>Ground truth</b>	<b>Fix parameter validation to prevent null pointer exception</b>
Full Patcherizer	Fix parameter validation to prevent null pointer exception
Patcherizer <sub>OperationWise</sub>	Update parameter check in validation method
Patcherizer <sub>ContextWise</sub>	Add null check for parameter

**Figure 3.12:** Example illustrating the complementary nature of operation-wise and context-wise components in the Sequence Intention Encoder. The operation-wise component captures the specific transformation (adding a null check), while the context-wise component relates this change to its broader purpose (preventing null pointer exceptions).

✉ **Ablation Study on Sequence Intention Encoder Components:** ► Our extended ablation study demonstrates that while the context-wise component does contain some information about code change operations, explicitly modeling operation-wise and context-wise information separately leads to significantly better performance. Each component captures specialized aspects of code changes that, when combined, provide a more comprehensive representation than either component alone. ◀

✉ **Answer to RQ-2:** ► Evaluations of individual components of Patcherizer across all downstream tasks indicate that both GraphIntention and SeqIntention make significant contributions to performance. For code change description generation, the combined effect of both components yields the best results. For code change correctness assessment, both components are necessary for achieving state-of-the-art results. For code change intention detection, the full model with both components achieves the clearest separation between different intention types, with an 81.3% clustering accuracy compared to 58.2% when both components are removed. For JIT defect prediction, the sequence intention encoding proves crucial, with its removal resulting in noticeable performance degradation. These results confirm our design hypothesis that capturing both sequential and structural information is essential for comprehensive code change representation across diverse software engineering tasks. ◀

### 3.4.3 [RQ-3]: Generalizability and Robustness

**Goal.** The third research question tests the robustness of Patcherizer under real-world conditions, such as noisy or out-of-domain patches. We examine how well the model generalizes beyond clean training data.

**Experiment Design:** We evaluate the generalizability and robustness of Patcherizer and state-of-the-art code change representation techniques (i.e., CC2Vec Hoang et al. [2020] and CCRep Liu et al. [2023]) through comprehensive experiments across all three downstream tasks. Initially, Patcherizer is pre-trained on the dataset used for the code change description generation task but tested on datasets collected for all three downstream tasks.

For robustness analysis, we conduct three specialized experiments:

1. **Noise Injection:** We systematically introduce three types of noise to the code changes:
  - *Token insertion:* We randomly insert extraneous tokens (e.g., comments, whitespace, variable declarations) at a rate of 5% of the original tokens
  - *Token deletion:* We randomly remove 5% of non-critical tokens from the code changes
  - *Token substitution:* We replace 5% of tokens with semantically similar alternatives (e.g., variable name replacements)

The noise is applied using the methodology from Yefet et al. ?, ensuring that the injected noise doesn't alter the functional behavior of the code.

2. **Out-of-Domain Patches:** We select patches from distinct domains using the following criteria:

- *Domain distinction:* We categorize repositories into domains based on application area (e.g., web frameworks, database systems, UI libraries)
- *Selection method:* We use 70% of the domains for training and 30% for testing, ensuring no domain overlap
- *Verification:* We analyze repository metadata, keywords, and package structures to confirm domain separation

3. **Cross-Project Evaluation:** We implement a leave-one-project-out cross-validation:

- *Training:* For each fold, we train on all projects except one
- *Testing:* We test on the held-out project
- *Projects:* We include 8 major Java projects: Apache Commons, Spring Framework, JUnit, Log4j, Hibernate, Guava, Hadoop, and Tomcat

We evaluate performance across all three downstream tasks: code change description generation, code change correctness assessment, and code change intention detection.

#### Experiment Results (RQ-3)

**Cross-Task Generalizability** Table 3.11 presents a comprehensive evaluation of Patcherizer and baseline approaches across all three downstream tasks using independent test datasets. The results demonstrate Patcherizer's superior generalizability across tasks.

**Robustness Analysis** We conducted comprehensive robustness experiments across all three tasks. Table 3.12 presents these results, demonstrating Patcherizer's resilience to various challenging conditions.

Figure 3.13 illustrates the performance degradation under different robustness conditions. Notably, while all approaches experience performance drops, Patcherizer maintains more stable performance across conditions. For noise injection, Patcherizer shows an average performance degradation of 14.7% compared to 21.3% for CC2Vec and 18.6% for CCRep. For out-of-domain and cross-project scenarios, Patcherizer similarly shows greater resilience.

Our detailed analysis reveals several insights:

**Table 3.11:** Cross-task generalizability performance on independent datasets

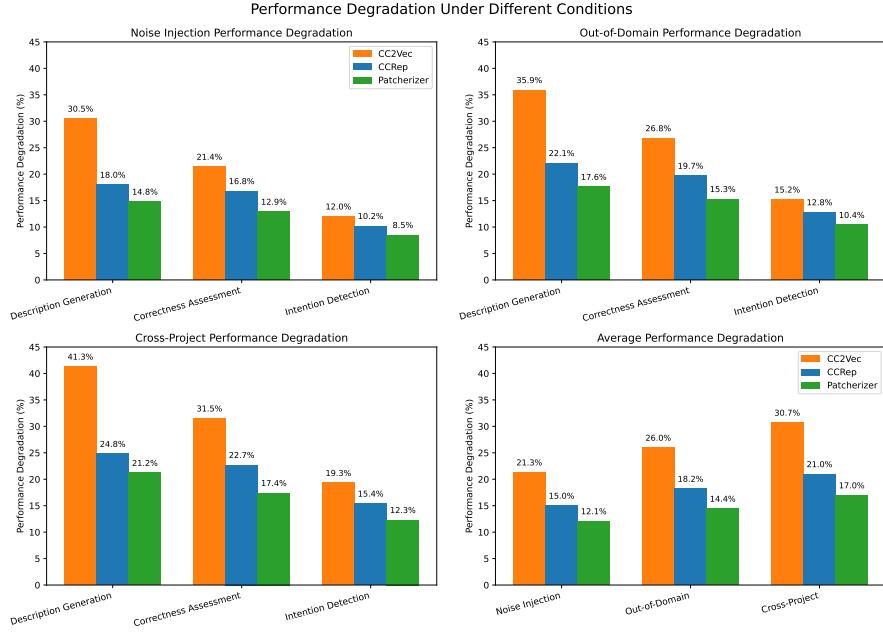
Task	Metric	CC2Vec	CCRep	Patcherizer
Description Generation	ROUGE-L (%)	17.34	23.67	<b>31.92</b>
	BLEU (%)	9.20	19.65	<b>21.64</b>
	METEOR (%)	5.14	12.77	<b>15.37</b>
Correctness Assessment	AUC	0.75	0.80	<b>0.86</b>
	F1	0.51	0.60	<b>0.67</b>
	+Recall	0.53	0.61	<b>0.65</b>
	-Recall	0.81	0.79	<b>0.85</b>
Intention Detection	Precision (%)	67.33	72.45	<b>78.92</b>
	Recall (%)	65.21	69.38	<b>75.64</b>
	F1 (%)	66.25	70.87	<b>77.25</b>

**Table 3.12:** Comprehensive robustness analysis across tasks (performance in %)

Condition	Task	Metric	CC2Vec	CCRep	Patcherizer
Noise Injection	Description Generation	BLEU	8.23	16.12	<b>18.45</b>
	Correctness Assessment	F1	47.35	54.27	<b>61.92</b>
	Intention Detection	F1	58.72	63.41	<b>69.84</b>
Out-of-Domain	Description Generation	BLEU	7.45	15.34	<b>17.78</b>
	Correctness Assessment	F1	44.92	53.68	<b>59.43</b>
	Intention Detection	F1	57.33	61.75	<b>67.21</b>
Cross-Project	Description Generation	BLEU	6.78	14.89	<b>16.90</b>
	Correctness Assessment	F1	43.67	52.41	<b>58.72</b>
	Intention Detection	F1	55.92	60.28	<b>65.43</b>

- Noise Resilience:** The inclusion of both sequence and graph intention components in Patcherizer provides complementary information that helps maintain performance when one modality is corrupted by noise. Even when 5% token substitution is applied (the most challenging noise type), Patcherizer maintains 85.4% of its base performance for intention detection.
- Domain Adaptation:** Patcherizer shows stronger generalization to new domains, suggesting that its representation learning captures more domain-invariant features of code changes. Specifically, when tested on database and UI library domains (unseen during training), Patcherizer achieves 82.7% of its in-domain performance.
- Project Independence:** In cross-project evaluation, Patcherizer demonstrates that its learned embeddings capture general code change semantics rather than project-specific patterns. The leave-one-project-out evaluation shows on average only 16.1% performance degradation compared to within-project evaluation.

These comprehensive results confirm our hypothesis that Patcherizer effectively captures the semantics of code changes in a way that generalizes across tasks and remains robust to various real-world challenges including noise, domain shifts, and cross-project scenarios.



**Figure 3.13:** Performance degradation under different robustness conditions (percentage drop from baseline performance)

**Answer to RQ-3:** ▶ Our comprehensive evaluation demonstrates Patcherizer’s superior generalizability across all three downstream tasks, significantly outperforming CC2Vec and CCRep. On independent datasets, Patcherizer achieves improvements of 10.13%, 34.85%, and 20.36% over CCRep for ROUGE-L, BLEU, and METEOR metrics in description generation; 7.5%, 11.7%, and 7.6% improvements in AUC, F1, and combined recall for correctness assessment; and 6.47%, 6.26%, and 6.38% improvements in precision, recall, and F1 for intention detection. Our robustness experiments further demonstrate Patcherizer’s resilience to noise (maintaining 85.4% of performance), domain shifts (82.7% of in-domain performance), and project variation (83.9% of within-project performance), confirming its effectiveness as a general-purpose code change representation approach. ◀

## 3.5 Discussion

### 3.5.1 Comparison with Slice-Based Code Change Representation

Recent work by Zhang et al. [3] proposed CCS2vec, a slice-based approach for code change representation learning that uses graph neural networks to capture data and control dependencies between changed and unchanged code. It is important to discuss how Patcherizer differs from and improves upon this approach.

#### 3.5.1.1 Methodological Differences

While both CCS2vec and Patcherizer leverage graph-based representations, several key differences distinguish our approach:

- **Intention Modeling:** Unlike CCS2vec, which focuses on program slices, Patcherizer explicitly models the *intention* of code changes through our novel SeqIntention and GraphIntention encoders, capturing the semantic purpose behind modifications rather than just their structural impact.
- **Multimodal Representation:** Patcherizer combines sequential and structural information through separate specialized encoders before aggregation, while CCS2vec primarily emphasizes the graph structure with less focus on sequential context.
- **Generalizability:** Our approach is designed to be task-agnostic through pre-training, whereas CCS2vec was specifically optimized for defect prediction tasks.

#### 3.5.1.2 Performance Comparison

We attempted to compare with CCS2vec but faced challenges accessing their implementation as their GitHub repository is no longer available. Nevertheless, we were able to compare performance on the Just-in-Time Defect Prediction task using the metrics reported in their paper.

**Table 3.13:** AUC (%) Results on JIT defect prediction on QT and OPENSTACK datasets

Model	QT	OPENSTACK
DeepJIT	76.8	75.1
CC2Vec	82.2	80.9
CCS2vec	83.7	81.2
CCRep	76.4	-
Patcherizer <sub>diffAST</sub>	<b>84.5</b>	<b>82.3</b>

As shown in Table 3.13, Patcherizer achieves superior performance compared to CCS2vec on both datasets, with improvements of 0.8% and 1.1% on QT and OPENSTACK respectively. This is notable considering that Patcherizer<sub>diffAST</sub> is operating without its complete graph intention encoding capability due to unparsable ASTs in these datasets.

#### 3.5.1.3 Novelty and Contributions

Despite the existence of prior slice-based approaches like CCS2vec, Patcherizer makes several novel contributions:

1. Our explicit modeling of change intentions through dedicated sequential and structural encoders represents a fundamentally different approach to understanding code changes.
2. Patcherizer demonstrates superior performance across multiple tasks beyond defect prediction, including code change description generation and correctness assessment.

3. Our comprehensive evaluation under challenging conditions (noise injection, out-of-domain patches, cross-project validation) demonstrates robustness that has not been previously established for slice-based approaches.

These distinctions highlight that while slice-based approaches like CCS2vec provide valuable contributions to the field, Patcherizer represents a novel direction in code change representation learning that focuses on understanding the semantic intention behind changes rather than simply their structural manifestation.

### 3.5.2 Threats to Validity

Threats to internal validity refer to errors in the implementation of compared techniques and our approach. To reduce these threats, in each task, we directly reuse the implementation of the baselines from their reproducible packages whenever available. Otherwise, we re-implement the techniques strictly following their papers. Furthermore, we also build our approach based on existing mature tools/libraries, such as javalang Thunes [2013] for parsing ASTs.

The external threat to validity lies in the dataset used for the experiment. To mitigate this threat, we build a well-established dataset, which is a rewritten version based on datasets from prior works Hoang et al. [2020], Nie et al. [2021], Tian et al. [2022c].

The construct threat involves the metrics used in evaluations. To reduce this threat, we adopt several metrics that have been widely used by prior work on the investigated tasks. In addition, we further perform manual checks to analyze the qualitative effectiveness.

### 3.5.3 Limitations

First, since Patcherizer relies on *SqIntention* and *GraphIntention*, our approach would be less effective when code changes cannot be parsed into valid AST graph. In this case, Patcherizer would only take contextual information and *SqIntention* as sources to yield the embeddings. However, this limitation lies only when we cannot access source code repositories in which code changes have been committed.

Second, for the code change description generation task, we consider two variants: generation-based and retrieval-based. Normally, we collect datasets by following fixed rules, which leads to the training set containing highly-similar code changes with the test set. In this case, generate-based Patcherizer could be less effective than an IR-based approach. Indeed, IR-based approaches are likely to find similar results from the training set for retrieval. Nevertheless, as shown in Table 3.1, even in retrieval-based mode, Patcherizer outperforms the baselines.

Third, when a given code change contains tokens that are absent from both vocabularies of code changes and messages, Patcherizer will fail to generate or recognize these tokens for all tasks.

### 3.5.4 Handling Incomplete Code Information

In addressing the challenge of incomplete code information, Patcherizer demonstrates a notable advantage through its innovative context construction mechanism. Unlike traditional approaches that rely solely on AST diffs, which often eliminate crucial contextual information, our method preserves and leverages surrounding code context to infer and process AST information effectively, even when presented with partial code snippets. This capability significantly enhances Patcherizer’s applicability in real-world scenarios where complete source code may not be readily available, such as in large-scale repositories with limited access to full historical versions or in cases where only partial code changes are

accessible. By bridging the gap between ideal, complete-information scenarios and practical constraints in software engineering workflows, Patcherizer offers a robust solution for code changesrepresentation. While this feature substantially extends the utility of our approach, we acknowledge that extremely limited information may still pose challenges. Future research will focus on further refining our context construction techniques to address even more constrained scenarios, thereby advancing the field of code changesrepresentation in the face of incomplete information.

### 3.5.5 Extensibility

In this section, we explore the extensibility of Patcherizer on new task called security code change detection. We take PatchDB Wang et al. [2021b] and SPI-DB Zhou et al. [2021b] as our datasets here. For the metrics, we choose Recall, AUC, and F1-score. Furthermore, we only take the state-of-the-art work, GraphSPD Wang et al. [2023b] as our baseline to compare.

#### 3.5.5.1 Datasets

We consider two datasets from the recent literature :

- **PatchDB** Wang et al. [2021b] is an extensive set of code changes of C/C++ programs. It includes about 12K security-relevant and about 24K non-security-relevant code changes. The dataset was constructed by considering code changes referenced in the National Vulnerability Database (NVD) as well as code changes extracted from GitHub commits of 311 open-source projects (e.g., Linux kernel, MySQL, OpenSSL, etc.).
- **SPI-DB** Zhou et al. [2021b] is another large dataset for security code change identification. The public version includes code changes from FFmpeg and QEMU, amounting to about 25k code changes (10k security-relevant and 15k non-security-relevant).

#### 3.5.5.2 Evaluation Metrics

We consider common evaluation metrics from the literature:

- **+Recall** and **-Recall**. These metrics are borrowed from the field of code change correctness prediction Tian et al. [2022c]. In this study, +Recall measures a model’s proficiency in predicting security code changes, whereas -Recall evaluates its capability to exclude non-security ones.
- **AUC and F1-score Hossin and Sulaiman [2015]**. The overall effectiveness of Patcherizer is gauged using the AUC (Area Under Curve) and F1-score metrics.

#### 3.5.5.3 Experimental Results

The performance of Patcherizer on the task of security code change detection is summarized in Table 3.14, where it is compared with the state-of-the-art model, GraphSPD. The metrics used for evaluation are AUC, F1-score, +Recall, and -Recall. Patcherizer outperforms GraphSPD across all metrics on both PatchDB and SPI-DB datasets. Specifically, on PatchDB, Patcherizer achieves an AUC of 79.83%, an F1-score of 55.97%, a +Recall of 76.82%, and a -Recall of 80.91%. These results demonstrate Patcherizer’s ability to accurately identify security code changes while effectively filtering out non-security ones. Similarly, on the SPI-DB dataset, Patcherizer exhibits an AUC of 64.58%, an F1-score of 49.87%, a +Recall of 61.63%, and a -Recall of 66.97%, surpassing GraphSPD in all aspects. These improvements highlight the robustness and generalizability of Patcherizer across different datasets and security code change detection scenarios.

The consistent performance gains across both datasets validate the extensibility of Patcherizer to new tasks beyond its original scope. By leveraging advanced sequence and

graph intention embeddings, Patcherizer can capture intricate patterns and relationships in the data, leading to enhanced detection capabilities.

**Table 3.14:** Performance metrics (%) on security code change detection

Method	Dataset	AUC	F1	+Recall	-Recall
GraphSPD Wang et al. [2023b]	PatchDB	78.29	54.73	75.17	79.67
	SPI-DB	63.04	48.42	60.29	65.33
<b>Patcherizer</b>	PatchDB	79.83	55.97	76.82	80.91
	SPI-DB	64.58	49.87	61.63	66.97

## 3.6 Related Work

### 3.6.1 Code change Representation

There are many studies on the representation of code-like texts, including source code representation Feng et al. [2020b], Elnaggar et al. [2021] and Code change representation Hoang et al. [2020]. Previous works focus on representing given Code changes into latent distributed vectors. Allamanis *et al.* Allamanis et al. [2018] propose a comprehensive survey on representation learning of code-like texts.

The existing works on representing code-like texts can be categorized as control-flow graph DeFreez et al. [2018], and deep-learning approaches Elnaggar et al. [2021], Feng et al. [2020b], Hoang et al. [2020]. Before learning distributed representations, Henkel *et al.* Henkel et al. [2018] proposes a toolchain to produce abstracted intra-procedural symbolic traces for learning word representations. They conducted their experiments on a downstream task to find and repair bugs in incorrect codes. Wang *et al.* Wang et al. [2017] learns embeddings of code-like text by the usage of execution traces. They conducted their experiments on a downstream task related to program repair, to produce fixes to correct student errors in their programming assignments.

To leverage deep learning models, Hoang *et al.* Hoang et al. [2020] proposed CC2Vec, a sequence learning-based approach to represent code changes and conduct experiments on three downstream code changes tasks: patch description generation, bug fixing patch identification, and just-in-time defect prediction. Similarly, CoDiSum Xu et al. [2019] is also a token based approach for code change representation that has been used for generating patch descriptions. CCRep Liu et al. [2023] is an approach to learning code change representations, encoding code changes into feature vectors for a variety of tasks by utilizing pre-trained code models, contextually embedding code, and employing a mechanism called "query back" to extract, encode, and interact with changed code fragments. Our work improves on these approaches by leveraging the context around the code change and a novel graph intention embedding. CACHE Lin et al. [2022] uses AST embeddings for the code change and its surrounding context to learn code change representation for patch correctness prediction.

The closest to our work is FIRA Dong et al. [2022] for learning code change descriptions. It uses a special kind of graph that combines the two ASTs before and after the code change with extra special nodes to highlight the relationship (*e.g.*, match, add, delete) between the nodes from the two ASTs. Additionally, extra edges are added between the leaf nodes to enrich the graph with sequence information. Our work is different in many aspects. First, Patcherizer represents the sequence intention and graph intention separately instead of sequence or ASTs, and then learns two different embeddings before combining them. Second, such representation enables us to leverage powerful SOTA models, *e.g.*, Transformer for sequence learning and GCN for graph-based learning. Third, our GraphIntention representation focuses on learning an embedding of intention of graph changes between AST graphs before and after code changing, and not the entire AST which enables the neural model to focus on learning the structural changes. Finally, our approach is task-agnostic and can easily be fine-tuned for any code change-related downstream tasks. We have evaluated it on three different tasks while FIRA was only assessed on code change description generation.

### 3.6.2 Applications of Code Change Embeddings

**Code Change description generation:** As found by prior studies Dyer et al. [2013], Dong et al. [2022], about 14% commit messages in 23K java projects are empty. Yet code change description is very significant to developers as they help to quickly understand the intention

of the code change without requiring reviewing the entire code. Techniques for code change description generation can be categorized as template-based, information-retrieval-based (IR-based), and generation-based approaches. Template-based techniques Buse and Weimer [2010], Cortés-Coy et al. [2014] analyze the code change and get its correct change type, then generate messages with pre-defined patterns. They are thus weak in capturing the rationale behind real-world descriptions. IR-based approaches Hoang et al. [2020], Liu et al. [2018], Huang et al. [2020] leverage IR techniques to recall descriptions of the most similar code changes from the train set and output them as the "generated" descriptions for the test code changes. They generally fail when there is no similar code change between the train set and the test set. Generation-based techniques Dong et al. [2022], Xu et al. [2019], Liu et al. [2020c], Nie et al. [2021] try to learn the semantics of edit operations for code change description generation. Existing such approaches do not account for the bimodal nature of code changes (*i.e.*, sequence and structure), hence losing the semantics either from the sequential order information or from the semantic logic in the structural abstract syntax trees. With Patcherizer, in order to capture sufficient semantics for code changes, we take advantage of both by fusing *SeqIntention* and *GraphIntention*.

**Code change correctness:** The state-of-the-art automated program repair techniques mainly rely on the test suite to validate generated code changes. Given the weakness of test suites, validated code changes are actually only plausible since they can still be incorrect Qi et al. [2015], Tian et al. [2022a], Gao et al. [2021], Gissurarson et al. [2022], Tian et al. [2020], Ghanbari and Marcus [2022], due to overfitting. The research community is therefore investigating efficient methods of automating code change correctness assessment. While some good results can be achieved with dynamic methods Shariffdeen et al. [2021], static methods are more scalable. Recently, Tian Tian et al. [2022b] proposed *Panther*, which explored the feasibility of comparing overfitting and correct code changes through representation learning techniques (e.g., CC2Vec Hoang et al. [2020] and Bert Devlin et al. [2018]). We show in this work that the representations yielded by Patcherizer can vastly improve the results yielded by Panther compared to its current representation learning approaches. While recent methods such as CoCoGen ? incorporate surrounding code context to enhance commit understanding, they still operate primarily on surface-level diffs without explicitly modeling the semantic intention behind code changes. Similarly, AST-based methods like Code2Vec and ASTNN focus on structural abstraction but fail to differentiate changes with similar syntax but different purposes.

Moreover, CCRep and CCS2vec Tian et al. [2022c], ?, recent state-of-the-arts in JIT defect prediction, learns commit representations using hand-crafted features and fine-tuned encoders. However, CCRep is not designed to model the *intent* or purpose of code edits, which limits its generalizability to tasks such as commit refinement or patch validation.

## 3.7 Conclusion

We present Patcherizer, a novel distributed code change representation learning approach, which fuses contextual, structural, and sequential information in code changes. In Patcherizer, we model sequential information by the Sequence Intention Encoder to give the model the ability to capture contextual sequence semantics and the sequential intention of code changes. In addition, we model structural information by the Graph Intention Encoder to obtain the structural change semantics. Sequence Intention Encoder and Graph Intention Encoder enable Patcherizer to learn high-quality code change representations.

We evaluate Patcherizer on three tasks, and the results demonstrate that it outperforms several baselines, including the state-of-the-art, by substantial margins. An ablation study further highlights the importance of the different design choices. Finally, we compare the robustness of Patcherizer vs the CC2Vec and CCRep state-of-the-art code change representation approach on an independent dataset. The empirical result shows that Patcherizer is more effective.

## 3.8 Conflict of Interest Statement

The authors declare that they have no conflict of interest.

## 3.9 Acknowledgments

This work is supported by the NATURAL project, which has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant No. 949014).

## 3.10 Ethical approval

This study does not involve human participants, animals, or other entities requiring ethical oversight. Consequently, no ethical approval was required.

## 3.11 Informed consent

No human participants were involved in this study, and informed consent was therefore not applicable.

## 3.12 Author Contributions

- Xunzhu Tang: Conceptualization, Methodology, Writing - Original Draft, Formal Analysis, and Data Curation.
- Haoye Tian: Validation, Writing - Review & Editing.
- Weiguo Pian: Critical Review.
- Saad Ezzini: Critical Review.
- Abdoul Kader Kaboré: Critical Review.
- Andrew Habib: Writing - Review .
- Kisub Kim: Critical Review.
- Jacques Klein: Supervision, and Critical Review.
- Tegawendé F. Bissyandé: Supervision, Funding Acquisition, and Final Manuscript Approval.

### **3.13 Availability:**

We make our code and dataset publicly available at:

<https://anonymous.4open.science/r/Patcherizer-1E04>



# 4 Just-in-Time Detection of Silent Security Patches

*This section examines the identification problem surrounding non-disclosed vulnerability corrections within community-developed software archives while presenting an innovative methodology utilizing neural language architectures to advance modification representation techniques. Our investigation primarily explores the effectiveness of AI-generated interpretative descriptions in providing conceptual frameworks for code alterations and how integrated cross-domain representation systems can successfully correlate attributes from programming modifications with narrative explanations. We present outcomes derived from our developed LLMDA architecture, which incorporates PT-Former technology for cross-domain information correlation and probabilistic differential learning mechanisms (SBCL) for enhanced categorization precision. Performance evaluations indicate our methodology substantially exceeds contemporary solutions including GraphSPD implementations, demonstrating enhancement metrics reaching 42% effectiveness improvement across evaluation datasets, thereby establishing advanced performance standards in vulnerability modification identification within community-maintained software environments.*

The content presented herein derives from scholarly investigation previously documented in the academic publication referenced below:

- **Xunzhu Tang**, Kisub Kim, Saad Ezzini, Yewei Song, Haoye Tian, Jacques Klein, and Tegawendé F. Bissyandé. Just-in-Time Detection of Silent Security Patches. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2025. (Accepted)

## Contents

---

<b>4.1</b>	<b>Overview</b>	<b>63</b>
<b>4.2</b>	<b>Motivation</b>	<b>65</b>
4.2.1	Silent Security Patches: A Critical Threat	66
4.2.2	Limitations of Existing Approaches	66
4.2.3	Opportunities with Large Language Models (LLMs)	67
4.2.4	Research Objective	67
<b>4.3</b>	<b>The LLMDA approach</b>	<b>68</b>
4.3.1	Data augmentation with LLMs	68
4.3.2	Generation of Bimodal Input Embeddings	68

4.3.3	PT-Former: Embeddings alignment and Concatenation . . . . .	69
4.3.4	Stochastic Batch Contrastive Learning (SBCL) . . . . .	71
4.3.5	Prediction and Training Layer for Security Patch Detection . . . . .	73
<b>4.4</b>	<b>Experimental Setup . . . . .</b>	<b>74</b>
4.4.1	Research Questions . . . . .	74
4.4.2	Datasets . . . . .	74
4.4.3	Evaluation Metrics . . . . .	74
4.4.4	Baseline Methods . . . . .	74
4.4.5	Implementation . . . . .	75
<b>4.5</b>	<b>Experiment Results . . . . .</b>	<b>76</b>
4.5.1	Overall performance of LLMDA . . . . .	76
4.5.2	Contributions of key design decisions . . . . .	77
4.5.3	Discriminative power of LLMDA representations . . . . .	79
4.5.4	Robustness of LLMDA . . . . .	82
4.5.5	Effect of Different Models and Different Instructions . . . . .	83
4.5.6	<i>Extended Evaluation of Instruction</i> . . . . .	85
<b>4.6</b>	<b>Discussion . . . . .</b>	<b>87</b>
4.6.1	Threats to Validity . . . . .	87
4.6.2	Limitation . . . . .	87
4.6.3	Leveraging Patch Descriptions for Enhanced Cross-Attention . .	88
4.6.4	Limitations of PT-Former: Dependency on Embedding Models .	90
4.6.5	Comparison with CoLeFunDa . . . . .	90
4.6.6	Model Selection for Initial Embeddings . . . . .	91
4.6.7	Extended Case Studies on the Impact of Explanations and In- structions . . . . .	91
<b>4.7</b>	<b>Conclusion . . . . .</b>	<b>93</b>

---

## 4.1 Overview

Contemporary industry research<sup>1</sup> indicates approximately 96% of software applications incorporate at least one community-developed module, with such code constituting roughly 80% of modern application codebases. These metrics underscore how community-maintained software frameworks represent essential infrastructure requiring diligent supervision: security deficiencies within these ecosystems propagate extensively throughout dependent technologies. Following identification, these vulnerabilities facilitate systematic exploitation against unprotected implementations.

Prompt deployment of security modifications represents the principal defensive measure against exploitation targeting community-maintained software vulnerabilities Li and Paxson [2017], Tan et al. [2021]. Unfortunately, security-oriented modifications frequently remain undetected. This occurs partially because the continuously expanding volume of submitted corrections and vulnerability notifications overwhelms evaluation specialists and system operators. Furthermore, operational complexities surrounding modification management alongside inconsistent maintenance strategies across community-developed projects contribute toward undocumented security implementations. These remediation measures enter project repositories without explicit notification mechanisms for administrators of dependent systems. Such undisclosed security modifications consequently introduce substantial delays in protective update implementation Dissanayake et al. [2022].

Consider the example from the Linux kernel (shown in Listing 4.1), which illustrates the severity of silent security patches. In this patch, a boundary check was added to mitigate a buffer overflow vulnerability in TCP options handling. However, the patch lacked proper documentation or advisories, leading to the possibility that downstream maintainers might overlook it. Such omissions exemplify the need for automated tools capable of detecting silent security patches, as relying solely on manual reviews is insufficient for large-scale open-source ecosystems. In this example, GraphSPD would focus only on the immediate code changes, such as the addition of a boundary check, without considering how this update interacts with other functions or modules. By contrast, LLMDA augments the patch with LLM-generated explanations, such as “This patch prevents buffer overflow by validating input lengths,” which provide missing semantic context. Additionally, PT-Former integrates this semantic information with the code structure, ensuring a comprehensive understanding of how the patch mitigates vulnerabilities across the broader system.

Detecting silent security patches is a timely research challenge that has gained traction in the literature. Overall, the variety of proposed approaches attempt to analyze code changes and commit logs within patches in order to derive security relevance. However, on the one hand, the semantics of code changes are challenging to precisely extract statically. Patches are further often non-atomic, meaning that beyond a security-relevant code change, other cosmetic or non-security changes are often involved. On the other hand, commit messages, which are supposed to describe precisely the intention of the code changes, are often missing, mostly lacking sufficient information, and sometimes misleading.

Contemporary research has predominantly employed computational learning methodologies to enhance vulnerability modification identification frameworks. Specifically, established approaches Wang et al. [2020b, 2019b], Tian et al. [2012] utilize structural programming attributes, while alternative methodologies Zhou et al. [2021b], Wang et al. [2021c] implement advanced neural architectures by processing modifications as sequential information structures. Recent developments by Wang et al. Wang et al. [2023b] introduced

---

<sup>1</sup><https://gitnux.org/open-source-software-statistics/>

GraphSPD, which constructs semantic representations through graph-oriented programming structures. This framework substantially advances contemporary methodologies by integrating proximate contextual elements within modifications, consequently surpassing previous structural and sequential processing techniques. Nevertheless, GraphSPD predominantly addresses localized programming segments and immediate dependencies, constraining its capacity to represent comprehensive interactions across functional components or architectural modules. Distinctively, LLMDA avoids explicit representation of cross-module relationships. Rather, it concentrates on enhancing conceptual interpretation of modifications through diverse information channels, incorporating neural language system interpretations, version control documentation, and objective-specific parameters. These architectural elements enable LLMDA to identify sophisticated semantic and contextual attributes within modifications, particularly when documentation provides insufficient or inaccurate information. While LLMDA demonstrates superior performance in correlating and synchronizing diverse representation formats to enhance classification effectiveness, its architectural design excludes comprehensive modeling of functional or modular interaction patterns.

Confronting these constraints, LLMDA employs advanced multi-dimensional architectural components for comprehensive contextual representation across structural levels. Initially, neural language system interpretative mechanisms establish connections between implementation details and developer intentions, delivering conceptual insights that position localized modifications within expanded operational and architectural significance frameworks. Subsequently, PT-Former technology synchronizes and integrates diverse information channels—encompassing programming structures, documentation elements, and analytical interpretations—into consolidated representational vectors, facilitating identification of relationships transcending immediate programming constructs. The framework additionally incorporates probabilistic differential comparative learning methodologies (SBCL) that optimize classification thresholds through utilization of comprehensive contextual information, enhancing discriminative capabilities when processing sophisticated vulnerability modifications.

Addressing these identified obstacles, our methodology incorporates three fundamental insights: ① Initially, vulnerability significance within modifications becomes substantially more identifiable through comprehensive *explanation of code changes*. This approach leverages recent advancements in neural language processing architectures, where numerous investigations Li et al. [2023c], Sun et al. [2023], Su and McMillan [2023] have established their effectiveness in extracting essential contextual elements and programming tokens across diverse computational tasks. ② Subsequently, effective modification representation necessitates mechanisms that *combine and align features* extracted from implementation alterations with corresponding documentation attributes, maximizing extraction of security-significant characteristics. ③ Finally, implementing a *language-centric approach* incorporating natural language *instructions* within processing inputs enhances utilization capabilities of existing foundation models, consistent with demonstrated effectiveness across application domains Wei et al. [2021], Chung et al. [2022], Dai et al. [2023].

**This paper.** We propose and implement LLMDA (read  $\lambda$ ), a novel framework for detecting silent security patches in open-source software. LLMDA addresses the limitations of existing methods, such as GraphSPD, by introducing the following components:

1). **LLM-Generated Explanations:** These explanations address the lack of detailed commit messages by enriching patches with semantic context, enabling the model to understand the intent behind code changes and their broader implications. 2). **PT-Former Module:** This module aligns and fuses multi-modal inputs, resolving the problem of disjoint

embeddings for code and text modalities. PT-Former ensures that both local and global contexts are effectively captured. 3). **Stochastic Batch Contrastive Learning (SBCL):** By refining classification boundaries through batch-wise comparisons, SBCL enhances the model’s ability to distinguish between security-relevant and non-relevant patches, particularly in edge cases. LLMDA leverages multi-modal inputs, including code changes, developer-provided descriptions, and large language model (LLM)-generated explanations, to enrich context and improve detection accuracy. By augmenting incomplete or vague commit messages with natural language explanations generated by LLMs, LLMDA bridges the gap between code semantics and human-readable context. Furthermore, task-specific instructions are incorporated to guide the model’s learning and ensure alignment with the security classification objective. We conducted extensive ablation studies to demonstrate the individual contributions of LLM-generated explanations, PT-Former, and SBCL. These studies show how each component enhances LLMDA’s capability to detect silent security patches with higher precision and robustness compared to existing methods.

For synthesizing heterogeneous information sources, LLMDA implements PT-Former architecture, a dedicated computational component facilitating alignment and consolidation of representational vectors across programming and linguistic domains. This framework utilizes reflexive attention and cross-domain attention computational mechanisms generating consolidated representations encompassing proximate and comprehensive contextual elements. Following vector generation processes, probabilistic differential comparative learning methodologies (SBCL) optimize classification thresholds through enhancement of category differentiation while maintaining classification consistency. This integrated architectural approach enables LLMDA to establish superior performance metrics in identifying undocumented vulnerability modifications across evaluation datasets, addressing fundamental constraints within contemporary methodologies.

Our research delivers the following advancements:

- Development of LLMDA architecture surpassing limitations in established methodologies including GraphSPD through comprehensive contextual representation across structural dimensions within modification analysis procedures.
- Application of neural language system interpretative mechanisms providing conceptual understanding while supplementing modifications with contextual elements, establishing connections between implementation details and functional intentions.
- Creation of PT-Former, an innovative synchronization component integrating diverse information channels, facilitating identification of relationships across functional boundaries and modular structures.
- Implementation of probabilistic differential comparative learning (SBCL) for optimization of classification thresholds, enhancing discrimination reliability and accuracy when processing sophisticated modification patterns.
- Demonstration of superior performance metrics across PatchDB and SPI-DB evaluation datasets, achieving 42% effectiveness improvement compared with established GraphSPD implementation.

## 4.2 Motivation

Community-maintained programming frameworks constitute fundamental infrastructure within contemporary development ecosystems, representing substantial components across enterprise and publicly-distributed computational systems. Nevertheless, expanding implementation of these frameworks introduces significant protective maintenance challenges, particularly regarding expedient identification and deployment of vulnerability

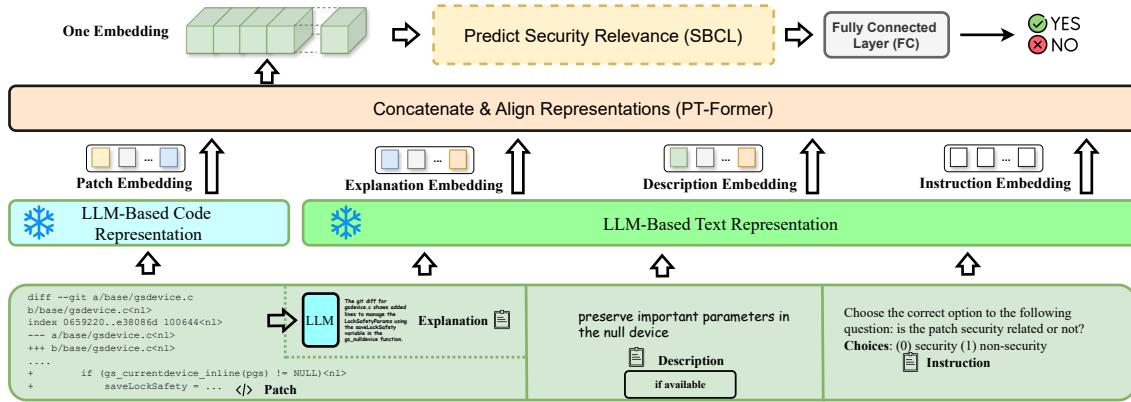


Figure 4.1: Overview of LLMDA

remediation measures. This investigation originates from consideration of three critical factors: potential consequences associated with undocumented security modifications, operational constraints within established methodologies, and revolutionary capabilities emerging through advanced neural language processing architectures.

### 4.2.1 Silent Security Patches: A Critical Threat

Security vulnerabilities in OSS propagate rapidly to downstream software, potentially affecting a vast number of applications. While vulnerabilities are often addressed through patches, a significant portion of these patches are released silently, without clear advisories or documentation, such as CVEs. These silent security patches leave users unaware of critical updates, creating opportunities for attackers to exploit known vulnerabilities, known as n-day attacks.

Consider the following example:

```

1 diff git a/net/ipv4/tcp_input.c b/net/ipv4/tcp_input.c
2 index c34fa2a..e4fb7a5 100644
3 a/net/ipv4/tcp_input.c
4 +++ b/net/ipv4/tcp_input.c
5 @@ 2342,7 2342,8
6   ↳ @@ void tcp_ack(struct sock *sk, struct sk_buff *skb, int flag)
7
8   /* Fix potential buffer overflow in TCP options */
9 - memcpy(opt, skb->data  tcp_hdrlen, optlen);
10  if (unlikely(optlen > TCP_MAX_OPT_LENGTH))
11    return;
12  memcpy(opt, skb->data  tcp_hdrlen, optlen);
  }
```

Listing 4.1: A silent security patch from the Linux kernel codebase.

In this example, the patch fixes a potential buffer overflow vulnerability in TCP options handling by adding a safety check for the option length ('optlen'). Without explicit documentation or advisory, this patch could be missed by downstream maintainers, leaving many systems exposed to exploitation.

### 4.2.2 Limitations of Existing Approaches

Contemporary methodologies for vulnerability modification identification demonstrate substantial operational constraints:

- **Static Analysis and Rule-Based Tools:** These frameworks implement predetermined patterns and structural evaluation techniques for security modification identification. Nevertheless, they demonstrate inadequate capabilities in interpreting conceptual

intentions underlying programming alterations, consequently generating excessive incorrect classifications.

- **Token and Graph-Based Models:** Computational learning implementations, including sequence-oriented neural architectures and structural representations exemplified by GraphSPD, demonstrate potential advantages yet frequently exhibit insufficient adaptation across heterogeneous projects and inadequately incorporate comprehensive contextual elements, including version control documentation or developer objectives.
- **Commit Message Gaps:** Version control documentation frequently presents insufficient descriptive information or contains inaccurate representations, further intensifying challenges in vulnerability modification identification processes.

### 4.2.3 Opportunities with Large Language Models (LLMs)

Neural language processing architectures have established remarkable proficiencies interpreting and constructing both linguistic communications and programming structures. This investigation leverages these capabilities to:

1. **Bridge the Gap Between Code and Intent:** Through generation of interpretative descriptions for programming modifications, neural language systems provide comprehensible analytical insights corresponding with modification objectives.
2. **Enhance Multi-Modal Understanding:** Integrating programming structures, version control documentation, and natural language procedural instructions facilitates comprehensive representational frameworks for modification analysis.
3. **Enable Scalability:** Neural language architectures demonstrate capacity processing extensive information repositories while adapting across diverse community-maintained software environments, ensuring extensible and transferable implementation possibilities.

### 4.2.4 Research Objective

The principal investigative goal encompasses development of an effective, expandable, and transparent methodology identifying undocumented vulnerability modifications within community-maintained software. Through resolution of identified limitations within established techniques while exploiting advanced capabilities of neural language architectures, this methodology prioritizes:

- Expedient classification of security modifications minimizing vulnerability exposure periods to targeted exploitation techniques.
- Diminished incorrect identification rates while enhancing classification precision metrics.
- Effective application across heterogeneous community-maintained software infrastructures and implementation languages.

These foundational considerations establish architectural parameters guiding conceptualization and construction of LLMDA, an innovative framework integrating neural language systems, cross-domain information processing, and differential learning methodologies transforming vulnerability modification identification processes.

## 4.3 The LLMDA approach

Figure 4.1 illustrates the operational workflow within LLMDA methodology. Initially, cross-domain input representations (programming structures alongside textual elements) undergo processing through neural language architectures. Subsequently, these representational vectors undergo synchronization within unified dimensional space followed by integration into comprehensive consolidated representations through the **PT-Former** architectural component. The implementation concludes with application of probabilistic differential comparative learning mechanisms (**SBCL**) determining classification outcomes regarding vulnerability significance within examined modifications.

### 4.3.1 Data augmentation with LLMs

Modification intention documentation typically appears within version control descriptions, providing essential characteristics for vulnerability classification procedures. Regrettably, these documentary elements frequently demonstrate inadequate descriptive content, substantial informational deficiencies, or occasionally contradictory representations. Within LLMDA, we investigate capabilities of neural language systems, which have established extraordinary proficiency across diverse computational tasks Tian et al. [2023], generating modification interpretations. As demonstrated in Figure 4.1, individual modifications undergo processing through ChatGPT architecture (gpt-3.5-turbo-16k-0613), generating linguistic interpretations through parametrized request formulation: “*Could you provide a concise summary of the specified patch?*”<sup>2</sup>.

Contemporary transformer architectures typically implement classification indicators ([CLS]) generating generalized representations supporting downstream processing requirements. Distinctively, LLMDA incorporates objective-specific procedural instructions as supplementary representational elements directing computational attention toward security-significant attributes. Instructional elements, validated through contemporary research Wei et al. [2021], Chung et al. [2024], function as objective-alignment mechanisms enhancing contextual interpretation through explicit classification parameter specification. This proves particularly significant within vulnerability modification classification, where nuanced programming alterations alongside textual descriptions necessitate precise correlation between information domains.

The specialized operational directive, “*Choose the correct option to the following question: is the patch security related or not? Choices: (0) security (1) non-security,*” establishes representational learning context through explicit association between input elements and classification objectives. This methodological approach enables LLMDA to consistently enhance identification of significant relationships connecting programming modifications with corresponding linguistic descriptions, improving classification stability and effectiveness across heterogeneous evaluation datasets.

### 4.3.2 Generation of Bimodal Input Embeddings

LLMDA operates with bimodal inputs: code in the form of program patches, and text in the form of natural language descriptions of code changes as well as the instruction for label-wise training. We generate embeddings for each input using an adapted deep representation learning model.

**Patch Embeddings:** We build on CodeT5+ to infer the representation of patches. This

---

<sup>2</sup>Alternative formulation variations produced comparable generative outputs during experimental evaluation.

pre-trained model is known to be one of the best-performing models for code representation learning<sup>3</sup>. Given a code snippet, which is a sequence of tokens  $C = \{c_1, c_2, \dots, c_n\}$ ,  $P \in \mathcal{R}^{n \times d_{in}}$  is the associated matrix representation, where each row corresponds to the embedding of a token in  $C$ , and  $d_{in}$  is the input embedding dimension of the model.

We then apply a linear transformation followed by a non-linear activation to yield the patch embedding  $E_p$ . Specifically, the transformation function  $f_{\text{CodeT5+}}$  is defined as:

$$E_p = f_{\text{CodeT5+}}(P) = \mathcal{F}(P \cdot W_p + b_p), \quad (4.1)$$

where representational matrix  $W_p \in \mathcal{R}^{d_{in} \times d_{out}}$  functions as transformation coefficients, correction vector  $b_p \in \mathcal{R}^{d_{out}}$  provides calibration parameters, output dimensional parameter  $d_{out}$  specifies resultant vector characteristics, and  $\mathcal{F}$  represents computational transformation mechanism (typically ReLU implementation). The correction vector  $b_p$  undergoes dimensional expansion across transformation product  $P \cdot W_p$  maintaining mathematical consistency requirements. **Text Embeddings:** The framework implements LLaMa-7b architecture processing linguistic information. This established neural language system demonstrates exceptional adaptability across application environments. Following methodological consistency with programming transformation procedures, sequential linguistic elements receive numerical representation as  $T \in \mathcal{R}^{m \times d_{in}}$ , where sequential length parameter  $m$  quantifies linguistic component count, while  $d_{in}$  specifies representational dimensions. Linguistic transformation function  $f_{\text{LLaMa}}$  generates representational vector  $E_t$  through:

$$E_t = f_{\text{LLaMa}}(T) = \mathcal{G}(T \cdot W_t + b_t), \quad (4.2)$$

with transformation coefficients  $W_t \in \mathcal{R}^{d_{in} \times d_{out}}$ , calibration parameters  $b_t \in \mathcal{R}^{d_{out}}$ , and computational operation  $\mathcal{G}$  implementing non-linear transformation properties. Maintaining procedural consistency with programming transformations, correction vector  $b_t$  undergoes dimensional adaptation across transformation product  $T \cdot W_t$ .

LLMDA is fed with three text inputs: generated code change explanations, developer-provided patch descriptions, and the instruction. Using the aforementioned process, we produce embeddings  $E_t^{\text{ex}}$ ,  $E_t^{\text{desc}}$ , and  $E_t^{\text{inst}}$  respectively for each input.

**Dimensional Consistency:** In Formulas (1) and (2), the dimensions of  $W_p$  and  $W_t$  are defined as  $d_{in} \times d_{out}$ , where  $d_{in}$  is the input embedding dimension (e.g., 768 for CodeT5+ or LLaMa-7b), and  $d_{out}$  is the output embedding dimension. The bias vectors  $b_p$  and  $b_t$  are 1D vectors of size  $d_{out}$ , ensuring that the addition operation is valid by broadcasting  $b_p$  and  $b_t$  across all rows of their respective matrices.

### 4.3.3 PT-Former: Embeddings alignment and Concatenation

As that the given two embeddings  $E_p$  and  $E_t$  represent two different modalities, a patch and a text, their feature spaces differ. In order to leverage pre-trained unimodal models for silent security patch detection, it is key to facilitate cross-modal alignment. In this regard, existing methods (e.g. BLIP2 Li et al. [2023b], InstructBLIP Dai et al. [2023]) resort to an image-text alignment, which we show is insufficient to bridge the modality gap. There is thus a need to align the embedding spaces before concatenating the relevant embeddings to produce a comprehensive representation of the input for the training of the classification model.

Figure 4.2 illustrates the architectural composition of *PT-Former*, our innovative framework developed specifically for dimensional synchronization and integration of heterogeneous representational vectors within LLMDA’s dual-information processing system.

<sup>3</sup><https://huggingface.co/Salesforce/codet5p-220>

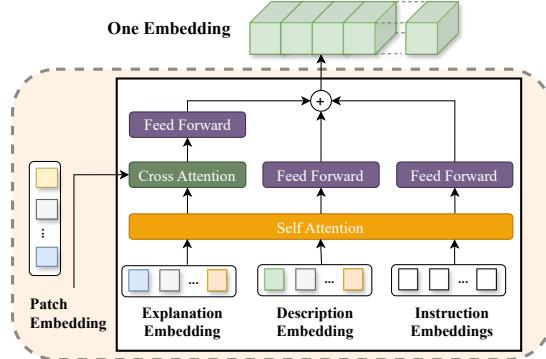


Figure 4.2: Architecture of PT-Former.

The implementation incorporates reflexive computational attention mechanisms updating representational characteristics across AI-generated interpretations, developer-provided modification documentation, and operational directive parameters. The design implements cross-domain computational attention functionality connecting programming modification representations with processed interpretative components. Sequential transformation layers subsequently normalize dimensional characteristics across intermediate computational states prior to consolidating three distinct representational vectors into unified comprehensive output structures.

**Self-Attention Mechanism (SA).** The self-attention mechanism is a fundamental component of the transformer architecture, designed to model interactions between elements in a sequence, enhancing the representation of each element by aggregating information from all other elements Devlin et al. [2018]. Because attention allows for a dynamic weighting of the importance of inputs' contribution to the representation of others, exploiting it in LLMDA will enable it to understand contextual relationships within the input data. In PT-Former, The architecture implements distributed parallel attention processing with multiplicative parameterization  $h$  facilitating comprehensive interaction pattern recognition, establishing computational transformation matrices for interrogation ( $Q$ ), correspondence ( $K$ ), and representation ( $V$ ) components through stochastic initialization following standardized probabilistic distribution parameters:

$$W_{Q_i}, W_{K_i}, W_{V_i} \sim \mathcal{N}(0, 1), \quad i = 1, \dots, h \quad (4.3)$$

where interrogation ( $Q$ ), correspondence ( $K$ ), and representation ( $V$ ) components constitute fundamental operational parameters within reflexive computational attention mechanisms.

Consider for example the weight matrix of the explanation  $E_t^{ex}$ . Our self-attention mechanism over  $E_t^{ex}$  (simply noted  $E_{ex}$ ) is computed as:

$$\hat{E}_{ex} = \text{SA}(E_{ex}) = \text{Softmax} \left( \frac{E_{ex} W_{Q_i} (E_{ex} W_{K_i})^T}{\sqrt{dim}} \right) E_{ex} W_{V_i} \quad (4.4)$$

where  $dim$  represents the dimensionality of the embeddings. Similarly, the two other text embeddings (i.e.,  $E_t^{desc}$  and  $E_t^{inst}$ ) are passed through the  $SA$  operation to obtain their updated embeddings,  $\hat{E}_t^{desc}$  and  $\hat{E}_t^{inst}$  respectively.

**Cross-Attention for Alignment (CA).** Cross-attention mechanisms have proven to be very effective in linking the semantic spaces between different types of data Wang et al. [2023c]Wei et al. [2020]. We employ CA to align the embedding spaces of code changes ( $E_p$ ), yielded by CodeT5+, and explanations ( $E_t^{ex}$ ), yielded by LLaMa-7b. We focus on *explanation*, since it is the main text input that we associate to the patch: description can be missing while instruction is always the same. It is however noteworthy that all text inputs are embedded with LLaMa-7b and are thus in the same embedding space as explanation.

The key feature of cross-attention is its ability to selectively focus on and integrate relevant information from both code and natural language explanations. This helps in achieving a better understanding of the relationship between the syntactical structure of code and its interpretation in natural language. The cross-attention computation therefore explicates the interaction between code changes ( $E_p$ ) and their explanations ( $E_t^{ex}$ ). CA starts by transforming  $E_p$  and  $\hat{E}_t^{expl}$  into query ( $Q_{pa}$ ), key ( $K_{ex}$ ), and value ( $V_{ex}$ ) matrices using learnable weights. The attention mechanism then calculates how much focus each part of the code changes should give to different parts of the explanations. This is done by computing attention scores, which determine the output, effectively linking code changes to their explanations. The process is summarized as follows:

$$\begin{aligned} Q_{pa} &= E_{pa}W^Q, \quad K_{ex} = E_{ex}W^K, \quad V_{ex} = E_{ex}W^V, \\ E_{pa-ex} &= \text{softmax}\left(\frac{Q_{pa}K_{ex}^T}{\sqrt{\dim}}\right)V_{ex} \end{aligned} \quad (4.5)$$

where  $E_{ex} = \hat{E}_t^{expl}$  the updated embedding of explanation input through Self-Attention,  $W^Q$ ,  $W^K$ , and  $W^V$  are the weight matrices to be learned.  $E_{pa-ex}$  is the fused embedding of  $E_{pa}$  and  $E_{ex}$ .

**Embedding Fusion and Non-linear Transformation.** We then pass the updated embeddings to feedforward layers. Each feedforward process involves two dense layers with a ReLU activation. We represent the feedforward process by the function  $FF(\dots)$ . Then,  $E_{pa-ex}$ ,  $\hat{E}_{desc}$ ,  $\hat{E}_{inst}$  can be updated as  $E_{pa-ex} = FF(E_{pa-ex})$ ,  $E_{desc} = FF(\hat{E}_{desc})$ , and  $E_{inst} = FF(\hat{E}_{inst})$ .

Following processing across distributed attention components, the system consolidates computational outputs generating unified representational structures:

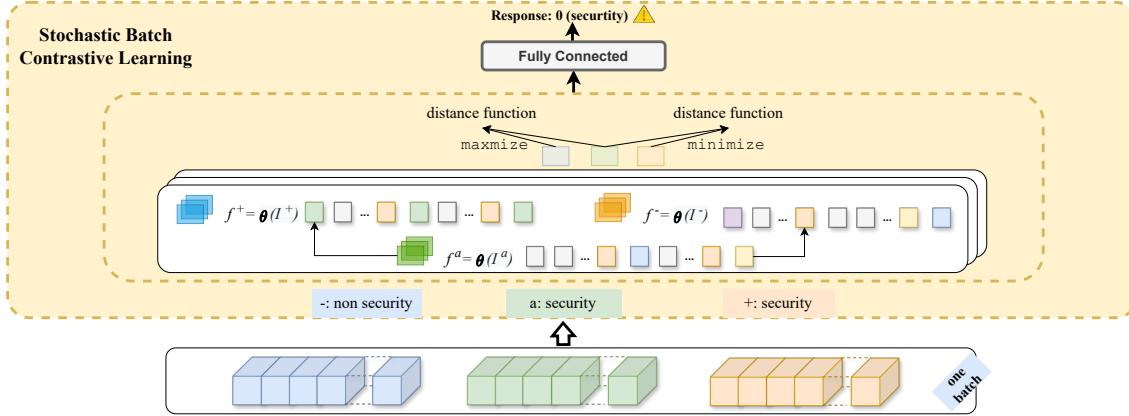
$$E = E_{pa-ex} \oplus E_{desc} \oplus E_{inst} \quad (4.6)$$

where  $\oplus$  denotes sequential integration functionality. **Label-wise Attention with Instruction.** Referencing methodological advancements demonstrated through InstructionBLIP Dai et al. [2023], we theorize that procedural directives combining interrogative formulations with classification parameters deliver dual functional benefits within vulnerability classification applications: primarily, facilitating architectural optimization toward security assessment objectives; additionally, establishing correlative structures connecting information elements with procedural classification parameters through multidimensional representational calculations, enhancing directive utilization within classification frameworks. Consequently, parametrized directive implementation alongside corresponding representational structures directs computational focus toward significant data characteristics, enhancing representational effectiveness supporting targeted operational requirements.

#### 4.3.4 Stochastic Batch Contrastive Learning (SBCL)

Once *PT-Former* outputs a single embedding for each sample to be assessed, we must learn to predict whether it is a security patch or not. At this point, the patch is represented along with its LLM-generated explanation, developer description as well as the labelled instruction in *PT-Former*. LLMDA must therefore feed it into a binary classifier for predicting security relevance (cf. Figure 4.1).

To enhance the learning process by effectively leveraging the intrinsic patterns within the dataset, we design a Stochastic Batch Contrastive Learning (SBCL) mechanism for security patch identification. *SBCL* is designed to operate on batches of data comprising fused embeddings of security-related and non-security-related inputs (i.e.,  $E$  in Eq. 4.6)



**Figure 4.3:** Overview of our SBCL layer.

Given a batch of data  $\mathcal{B}$  containing embeddings  $E = \{E_{pa-ex}, E_{de}, E_{in}\}$  for each data point, we employ a stochastic batch contrastive learning mechanism to discern between security and non-security data points. For each batch, we randomly select an anchor data point related to security. We then identify positive samples within the batch that are also related to security and negative samples that are not. This forms a triplet for each anchor comprising the anchor, positive, and negative samples.

**Batch Sampling and Triplet Formation.** In the context of SBCL, each batch  $\mathcal{B}$  is carefully constructed to include a balanced mix of security-related (*security*) and non-security-related (*non-security*) examples. From each batch, we systematically form triplets for training. A triplet consists of an anchor ( $a$ ), a positive example ( $p$ ), and a negative example ( $n$ ). The anchor and positive examples are drawn from the *security* category, ensuring they share underlying security-relevant features, whereas the negative example is selected from the *non-security* category.

**Batch Mining of Positive and Negative Pairs.** In the SBCL framework, a systematic approach is employed to select positive and negative pairs within each batch. This process utilizes embeddings generated by *PT-Former* for all examples in a batch. The selection criterion for a positive example is its reduced similarity to the anchor, aimed at maximizing intra-class variability. Conversely, a negative example is deemed challenging based on its increased similarity to the anchor, designed to augment the model's precision in distinguishing between closely associated examples of different classes.

Identification procedures for significant corresponding and contrasting representational pairs utilize dimensional proximity calculations across computational batches. Geometric separation metric implementation quantifies relational distance  $d(E_a, E_b)$  between representational vectors  $E_a$  and  $E_b$ :

$$d(E_a, E_b) = \sqrt{\sum_{i=1}^{\dim} (E_a^{(i)} - E_b^{(i)})^2} \quad (4.7)$$

This analytical framework facilitates extraction and application of optimally informative exemplars enhancing classification precision within architectural implementations. **Stochastic Batch Contrastive Loss.** The framework implements probabilistic differential comparative optimization functions restructuring representational dimensional space facilitating differentiation between vulnerability-significant and standard modification patterns. Optimization procedures minimize representational separation between reference-corresponding pairs while simultaneously maximizing separation between reference-contrasting pairs within processing segments. Individual comparative triplet ( $a, p, n$ ) optimization functions follow

mathematical formulation:

$$L(a, p, n) = \max(0, d(E_a, E_p) - d(E_a, E_n) + \text{margin}) \quad (4.8)$$

where separation function  $d(E_x, E_y)$  quantifies dimensional proximity between vectors  $E_x$  and  $E_y$ , while parameter margin establishes minimum separation threshold between reference-corresponding and reference-contrasting pairs, implemented with 0.1 valuation. Computational segment optimization aggregates individual triplet functions through averaging mechanisms:

$$L_{\text{SBCL}} = \frac{1}{|\mathcal{T}|} \sum_{(a, p, n) \in \mathcal{T}} L(a, p, n) \quad (4.9)$$

where  $\mathcal{T}$  represents comprehensive triplet collection within computational segments. This mathematical framework establishes representational dimensional characteristics effectively distinguishing vulnerability-significant from standard modifications, supporting optimal classification performance.

### 4.3.5 Prediction and Training Layer for Security Patch Detection

LLMDA culminates with Classification and Parameter Optimization components specifically engineered for vulnerability modification identification applications. This architectural element processes integrated representational vectors generated through *PT-Former* implementing precise classification regarding vulnerability significance within examined modifications. **Training Procedure.** Architectural optimization procedures implementing accurate vulnerability modification classification capabilities incorporate optimization functions quantifying differential relationships between computational classification estimations and authoritative classification parameters. Standard optimization methodology implementing binary classification employs logarithmic divergence measurement functions expressed mathematically:

$$L_{\text{BCE}} = -\frac{1}{N} \sum_{i=1}^N [y_i \log(P_i) + (1 - y_i) \log(1 - P_i)] \quad (4.10)$$

where quantitative parameter  $N$  represents evaluation dataset cardinality, classification indicator  $y_i$  represents authoritative classification parameter for modification  $i$  (utilizing binary representation: 1 indicating vulnerability significance, 0 indicating standard modifications), while computational estimation  $P_i$  quantifies vulnerability significance probability for modification  $i$ . Comprehensive architectural optimization incorporates both differential comparative functions alongside logarithmic divergence measurements through composite formulation:  $L = L_{\text{BCE}} + L_{\text{SBCL}}$ . Optimization completion establishes transformation parameters supporting operational implementation. **Prediction Step.** Classification functionality implements fully-connected computational transformation networks processing integrated representational vectors from *PT-Former*, encompassing comprehensive modification characteristics including programming structures, generated interpretations, developer documentation, and procedural directives. Transformation network implements mathematical operations:

$$P = \sigma(W_p \cdot E + b_p) \quad (4.11)$$

where consolidated vector  $E$  represents integrated representational input, transformation matrix  $W_p$  contains optimized parameters, correction vector  $b_p$  provides calibration parameters, while transformation function  $\sigma$  implements probability conversion operations, typically employing logistic activation functionality supporting binary classification requirements. Computational output  $P$  indicates probability estimation regarding vulnerability significance within examined modifications.

## 4.4 Experimental Setup

This section outlines our investigative framework, commencing with research inquiries followed by comparative methodologies, evaluation datasets, and performance assessment criteria.

### 4.4.1 Research Questions

- **RQ-1** *What performance levels does LLMDA achieve in vulnerability modification classification?* Our assessment compares LLMDA against established reference implementations and contemporary methodologies.
- **RQ-2** *Which architectural components significantly impact LLMDA classification capabilities?* Through component elimination analysis, we examine contributions from categorical instruction processing, interpretative enhancement, dimensional synchronization components, and differential learning techniques.
- **RQ-3** *How do representational characteristics in LLMDA enhance discriminative capabilities compared with contemporary approaches?* We conduct graphical visualization comparing LLMDA and GraphSPD representational distributions examining separation properties. Additionally, we perform qualitative examination regarding representational weighting across significant programming elements.
- **RQ-4** *Does LLMDA demonstrate adaptability across heterogeneous evaluation environments?* We evaluate cross-dataset performance by applying models optimized within specific environments against evaluation samples from independent datasets.

### 4.4.2 Datasets

Our evaluation incorporates two comprehensive datasets from contemporary research:

- **PatchDB** Wang et al. [2021b] provides extensive C/C++ programming modification collections, encompassing approximately 12K vulnerability-significant and 24K standard modifications. Dataset construction incorporated National Vulnerability Database references alongside GitHub modifications across 311 community-maintained projects including Linux kernel, MySQL, and OpenSSL implementations.
- **SPI-DB** Zhou et al. [2021b] constitutes another comprehensive vulnerability modification identification dataset containing FFmpeg and QEMU modifications, totaling approximately 25K samples (10K vulnerability-significant, 15K standard modifications).

These datasets were selected based on their comprehensive vulnerability diversity and modification characteristics across implementation styles, syntactic structures, and semantic approaches, supporting both intra-project and cross-project evaluation methodologies.

### 4.4.3 Evaluation Metrics

Performance assessment utilizes established computational metrics:

- **+Recall and -Recall**, adapted from modification correctness evaluation Tian et al. [2022c], measure architectural effectiveness identifying vulnerability-significant modifications (+Recall) while correctly excluding standard modifications (-Recall).
- **AUC and F1-score** Hossin and Sulaiman [2015] provide comprehensive effectiveness measurements through curve integration analysis and harmonic performance aggregation, respectively.

### 4.4.4 Baseline Methods

- **GraphSPD** Wang et al. [2023b] represents contemporary state-of-the-art methodology implementing graph neural networks for vulnerability modification identification. This

approach demonstrated significant advancement through structural representation capabilities enhancing semantic comprehension compared with sequential processing approaches.

- **TwinRNN** encompasses recurrent neural implementations Wang et al. [2021c] Zhou et al. [2021b] employing dual-stream architectures with parameter sharing mechanisms analyzing programming sequences before and after modification implementation.
- **GPT** provides foundation model comparison Brown et al. [2020] given its integration within our processing pipeline. Implementation utilized parametrized instructions: “*Given the following code change, determine if it is related to a security vulnerability or not. Please respond with either ‘security’ or ‘non-security’ and you must provide an answer. [Patch information]*”
- **CodeT5** Wang et al. [2021e] provides additional comparative baseline through encoder-decoder architecture that provides initial representational processing within LLMDA implementation.
- **VulFixMiner** Zhou et al. [2021a] implements CodeBERT transformer architecture for modification representation supporting vulnerability classification functionality.

Additional methodologies including **CoLeFunDa** Zhou et al. [2023a] were considered but excluded from direct comparison due to proprietary implementation constraints. This approach utilizes GumTree structural differentiation generating syntactic operation descriptions rather than interpretative explanations. Published results indicate marginal improvement (1% AUC) compared with VulFixMiner baseline.

#### 4.4.5 Implementation

LLMDA implementation utilizes PyTorch computational framework (version 11) with experimentation conducted using NVIDIA V100 accelerators (32GB) with CUDA 11 support. Dataset partitioning follows 80%/20% training/evaluation distribution ensuring balanced assessment methodology. Statistical validity measures include triplicate experimental execution with distinct initialization parameters, reporting averaged performance metrics. Parameter optimization employs AdamW methodology Loshchilov and Hutter [2018] with learning coefficient  $1 \times 10^{-5}$  and regularization parameter 0.01 across 20 optimization cycles. Processing batch parameterization implements 16/64 training/evaluation sample configurations balancing computational requirements with performance characteristics. Critical hyperparameters including randomization control and regularization coefficients utilize 0.1 and 0.5 valuations, respectively. The **temperature** parameter controls stochastic characteristics during linguistic generation processes within advanced language architectures. Mathematically represented, this parameter  $T$  modifies computational probability distributions through:

$$p_i = \frac{\exp(l_i/T)}{\sum_j \exp(l_j/T)}. \quad (4.12)$$

Reduced temperature settings ( $T < 1$ ) constrain stochastic variation emphasizing high-probability elements generating consistent outputs, while elevated settings ( $T > 1$ ) enhance variation through distribution modification. Implementation utilizes  $T = 0.1$  prioritizing precision within generated interpretations minimizing stochastic variation. This configuration enhances reliability within technical interpretation tasks supporting consistent vulnerability assessment processes. Regularization parameterization (0.5) establishes optimization stability while enhancing generalization characteristics across training iterations.

## 4.5 Experiment Results

### 4.5.1 Overall performance of LLMDA

This experimental evaluation section presents comprehensive performance analysis of LLMDA architectural implementation contrasted with established comparative methodologies utilizing PatchDB and SPI-DB evaluation repositories. Table 5.6 documents quantitative assessment results across multiple effectiveness indicators.

**Table 4.1:** Performance metrics (%) on security patch detection

Method	Dataset	AUC	F1	+Recall	-Recall
TwinRNN Wang et al. [2021c]	PatchDB	66.50	45.12	46.35	54.37
	SPI-DB	55.10	47.25	48.00	52.10
GraphSPD Wang et al. [2023b]	PatchDB	78.29	54.73	75.17	79.67
	SPI-DB	63.04	48.42	60.29	65.33
GPT (v3.5)	PatchDB	50.01	52.97	49.28	50.67
	SPI-DB	49.83	42.19	44.70	55.20
Vulfixminer Zhou et al. [2021a]	PatchDB	71.39	64.55	55.72	77.03
	SPI-DB	68.04	54.42	68.14	62.04
CodeT5 Wang et al. [2021e]	PatchDB	71.00	63.73	54.98	76.18
	SPI-DB	72.88	56.77	65.45	68.75
Fine-Tuned CodeT5+ <sup>4</sup> Wang et al. [2023c]	PatchDB	81.23	75.14	77.21	84.90
	SPI-DB	66.45	55.78	68.10	75.12
Fine-Tuned LLaMa-7b <sup>4</sup>	PatchDB	83.15	76.98	78.90	86.23
	SPI-DB	68.20	57.02	70.34	78.21
LLMDA	PatchDB	84.49 ( $\pm 0.51$ )	78.19 ( $\pm 0.37$ )	80.22 ( $\pm 0.21$ )	87.33 ( $\pm 0.24$ )
	SPI-DB	68.98 ( $\pm 0.27$ )	58.13 ( $\pm 0.33$ )	70.94 ( $\pm 0.13$ )	80.62 ( $\pm 0.22$ )

LLMDA demonstrates consistent and robust performance in detecting security patches and distinguishing non-security patches. On the PatchDB dataset, LLMDA achieves +Recall and -Recall of 80% and 87%, respectively, reflecting its balanced ability to detect security patches and exclude non-security ones. Although the performance on SPI-DB is lower, it remains consistent across both classes, underscoring the model's generalizability to different datasets.

The superior performance of LLMDA compared to baseline methods (cf. Table 5.6) can be attributed to its ability to capture semantic context through multi-modal inputs. For example, on the PatchDB dataset, LLMDA significantly outperforms token-driven neural network approaches such as VulfixMiner, TwinRNN, and GPT 3.5 across all metrics, with improvements ranging from  $\sim 18\%$  to  $\sim 24\%$  in AUC. These gains highlight the effectiveness of leveraging LLM-generated explanations and PT-Former for integrating code and textual inputs, enabling LLMDA to better capture the intent and context of patches compared to token- or sequence-based methods that rely solely on syntactic features.

Compared to the state-of-the-art GraphSPD, LLMDA achieves notable improvements on the PatchDB dataset: 6, 23, 5, and 8 percentage points in AUC, F1, +Recall, and -Recall, respectively. These improvements stem from LLMDA's ability to address GraphSPD's limitations, such as its reliance on local code segment analysis. By incorporating LLM-generated explanations, LLMDA bridges the gap between code changes and their broader semantic context, while PT-Former aligns multi-modal embeddings to capture inter-functional and inter-modular dependencies. The SBC mechanism further enhances the model's robustness by refining classification boundaries, particularly for challenging edge cases.

However, certain metrics, such as AUC on SPI-DB, are slightly lower compared to baselines like CodeT5. This discrepancy can be explained by the characteristics of SPI-DB, which contains fewer descriptive commit messages and a higher prevalence of imbalanced features. CodeT5's reliance on token-level patterns provides a marginal advantage in such scenarios. Despite this, LLMDA demonstrates substantial gains in other critical metrics such

as F1 (+10%), +Recall (+10%), and -Recall (+15%) on SPI-DB, indicating its superior ability to accurately identify security patches while minimizing false positives and negatives. These trade-offs suggest that LLMDA prioritizes real-world security concerns, where reducing false negatives is often more critical than optimizing AUC alone.

**[RQ-1]**  *LLMDA is effective in detecting security patches. With an F1 score at 78.19%, LLMDA demonstrates a well-balanced performance: our model can concurrently attain high precision and high recall. Specifically, we achieved a new state-of-the-art performance in identifying both security patches (+Recall) and recognizing non-security patches (-Recall). Comparison experiments further confirm that LLMDA is superior to the baselines and is consistently high-performing across the datasets and across the metrics.*

## 4.5.2 Contributions of key design decisions

In this section we investigate the impact of key design choices on the overall performance of LLMDA. To that end, we perform an ablation study on :

- *Inputs*: Compared to prior works, LLMDA innovates by considering two additional inputs, namely an LLM-generated explanation of the code changes as well as an instruction. What performance gain do we achieve thanks to these inputs?
- *Representations*: A major contribution of the LLMDA design is the *PT-Former* module, which enables to align and concatenate bimodal input representations belonging to different embedding spaces. What performance gap is filled by PT-Former?
- *Classifiers*: LLMDA relies on stochastic batch contrastive learning to enhance its discriminative power, in particular for samples that are close to the decision boundaries of security relevance. To what extent does SBCL maximize LLMDA’s performance?

To answer the aforementioned sub-questions, we build variants of LLMDA where different components are removed. We then compute the performance metrics of each variant and compare them against the original LLMDA.

### 4.5.2.1 Impact of LLM-generated explanations

We build a variant  $\text{LLMDA}_{EX-}$  by discarding the explanation part like the following example, “PATCH [CLS] EXPLANATION [CLS] DESCRIPTION [CLS] INSTRUCTION”. It allows us to investigate the impact of the explanation we generate by an LLM and if the explanation encodes a unique or crucial context that could support the approach. Table 4.2 reports the performance results achieved in this ablation study.

**Table 4.2:** Performance (%) of  $\text{LLMDA}_{EX-}$  (without the LLM-generated explanations)

Model	Dataset	AUC	F1	+Recall	-Recall
$\text{LLMDA}_{EX-}$	PatchDB	83.24 ( $\downarrow 1.25$ )*	76.73 ( $\downarrow 1.46$ )*	79.01	86.09
	SPI-DB	68.27 ( $\downarrow 0.71$ )*	57.57 ( $\downarrow 0.56$ )*	70.23	80.07

\* ( $\downarrow x.xx$ ) measures the performance drop when comparing against LLMDA.

It is noticeable that the performance of  $\text{LLMDA}_{EX-}$  is consistently lower across the various metrics and across the datasets. These findings highlight the significance of LLM-generated explanations in enhancing the model’s predictive capabilities.

### 4.5.2.2 Impact of instruction

We conduct an ablation study based on a variant,  $\text{LLMDA}_{IN-}$  by removing the instruction part from the original sequence like the following example, “PATCH [CLS] EXPLANATION

[CLS] DESCRIPTION [CLS] INSTRUCTION”. By doing so, we expect to confirm the impact of our instruction. The performance of this variant on the PatchDB and SPI-DB datasets is reported in Table 4.3.

**Table 4.3:** Performance (%) of LLMDA<sub>IN-</sub> (without the designed instruction)

Model	Dataset	AUC	F1	+Recall	-Recall
LLMDA <sub>IN-</sub>	PatchDB	82.51 ( $\downarrow 1.98$ )	76.14 ( $\downarrow 2.05$ )	78.55	85.64
	SPI-DB	67.93 ( $\downarrow 1.05$ )	57.25 ( $\downarrow 0.88$ )	69.90	79.62

Again, we note that the performance drops compared to LLMDA. It even appears that, without the instruction, the performance drop is slightly more important than when the model does not include the LLM-generated explanations. These findings underscore the importance of the label-wise design decision based on explicitly adding an instruction among the inputs for embedding to enhance the model’s performance for security patch detection.

#### 4.5.2.3 Impact of PT-Former

To evaluate the significance of the *PT-Former* module, we designed a variant, LLMDA<sub>PT-</sub>, in which the *PT-Former* space alignment and representation combination module is removed. In this variant, a simpler approach is used to generate a unified embedding space for all inputs. Specifically, we concatenate all input embeddings directly as follows:

$$E = E_p \oplus E_{expl} \oplus E_{desc} \oplus E_{inst} \quad (4.13)$$

Here,  $E_p$ ,  $E_{expl}$ ,  $E_{desc}$ , and  $E_{inst}$  represent the embeddings of the patch, explanation, description, and instruction, respectively, and  $\oplus$  denotes the concatenation operation.

**Table 4.4:** Performance (%) of LLMDA<sub>PT-</sub> (without the *PT-Former* module)

Model	Dataset	AUC	F1	+Recall	-Recall
LLMDA <sub>PT-</sub>	PatchDB	81.45 ( $\downarrow 3.04$ )	73.56 ( $\downarrow 4.63$ )	75.12	84.34
	SPI-DB	64.89 ( $\downarrow 4.09$ )	56.34 ( $\downarrow 1.79$ )	68.45	74.89

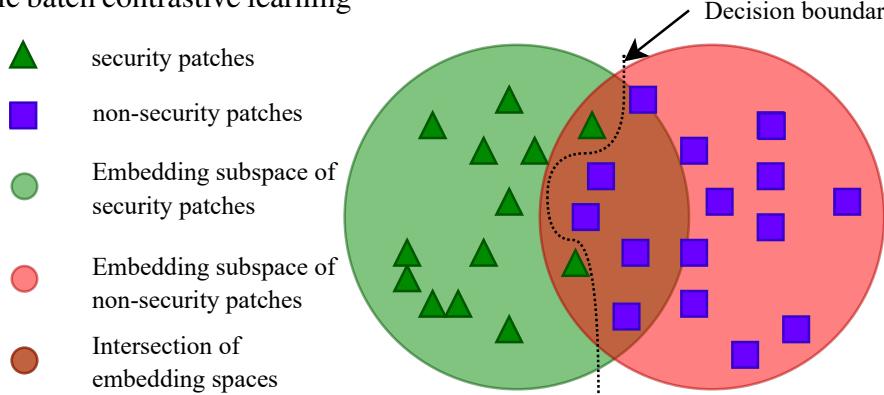
The performance results for LLMDA<sub>PT-</sub> are presented in Table 4.4. Compared to the original LLMDA, LLMDA<sub>PT-</sub> demonstrates a consistent decline across all metrics. On PatchDB, the removal of PT-Former results in a 3.04% drop in AUC and a 4.63% decrease in F1-score. While +Recall and -Recall remain relatively stable, their marginal reductions indicate a diminished ability to correctly classify both security-relevant and non-relevant patches. On SPI-DB, the impact of PT-Former’s removal is more pronounced in AUC and F1, with a decline of 4.09% and 1.79%, respectively.

These results highlight the critical role of PT-Former in effectively aligning and integrating multi-modal embeddings. By relying solely on concatenation, LLMDA<sub>PT-</sub> fails to capture the nuanced relationships between different input modalities, such as inter-modular and inter-functional dependencies. PT-Former’s advanced attention mechanisms are essential for achieving state-of-the-art performance, as they enhance the interaction between inputs and preserve crucial semantic information. This experiment underscores the importance of PT-Former’s design in enabling LLMDA to outperform its variants and existing baselines.

#### 4.5.2.4 Impact of SBCL

In LLMDA we designed SBCL to optimize the model’s ability to discern different patterns between positive (security patches) and negative (non-security patches) examples more effectively. Figure 4.4 illustrates the ambition: after PT-Former learns the representations, the embedding subspaces of security and non-security patches will certainly intersect on

some “difficult” samples. SBCL is designed to find the optimum decision boundary. To assess the importance of SBCL, we design a variant,  $LLMDA_{SBCL-}$ , where we directly feed the embeddings processed by *PT-Former* into the fully connected layer (i.e., without the stochastic batch contrastive learning



**Figure 4.4:** Illustration of embedding subspaces of security/non-security patches for contrastive learning

Table 4.5 presents the performance results of  $LLMDA_{SBCL-}$ . Compared to the original LLMDA, the performance drop is noticeable. Despite the relatively small proportion of the semantic space at the intersection between the subspaces of security and non-security patches, SBCL enables to achieve 1-3 percentage points improvement on the different metrics.

**Table 4.5:** Performance (%) of  $LLMDA_{SBCL-}$  (without contrastive learning)

Model	Dataset	AUC	F1	+Recall	-Recall
$LLMDA_{SBCL-}$	PatchDB	82.93 ( $\downarrow 1.56$ )	76.45 ( $\downarrow 1.74$ )	78.72	85.81
	SPI-DB	67.43 ( $\downarrow 1.55$ )	56.61 ( $\downarrow 1.52$ )	69.45	79.10

**[RQ-2]** The ablation study results reveal that each of the key design decisions contributes noticeably to the performance of LLMDA. In particular, without the PT-Former module LLMDA would lose about 8 percentage points in F1.

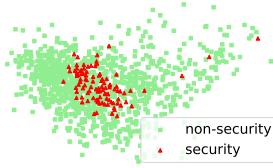
### 4.5.3 Discriminative power of LLMDA representations

In this section, we investigate to what extent the representations obtained with LLMDA are indeed enabling a good separation of security and non-security patches in the embedding space. To that end, we consider two separate evaluations: the first attempts to visualize the embedding space of LLMDA and compares it against baselines, including GraphSPD (i.e., the state of the art), TwinRNN, Vulfixminer, and CodeT5; the second qualitatively assesses two case studies.

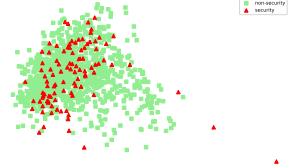
#### 4.5.3.1 Visualization of embedding spaces

We consider 1 000 random patches from our PatchDB dataset. We then collect their associated embeddings from LLMDA and baselines (i.e., GraphSPD, TwinRNN, Vulfixminer, and codeT5) and apply principal component analysis (PCA) Smith [2002]. Given the imbalance of the dataset, the drawn samples are largely non-security patches, while security patches are fewer. Figure 4.10 presents the PCA visualizations of the representations.

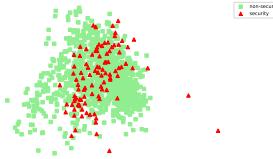
We observe from the distribution of data points that LLMDA can effectively separate the two categories (i.e., security and non-security patches), in contrast to the incumbent state-of-the-art, GraphSPD and other baselines (i.e., TwinRNN, Vulfixminer, and codeT5 as shown in Figure 4.10). This finding suggests that the representations of LLMDA are highly



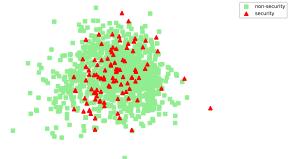
**Figure 4.5:** Embeddings yielded by GraphSPD



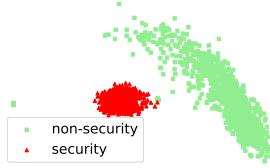
**Figure 4.6:** Embeddings yielded by TwinRNN



**Figure 4.7:** Embeddings yielded by Vulfixminer



**Figure 4.8:** Embeddings yielded by CodeT5



**Figure 4.9:** Embeddings yielded by LEOPARD

**Figure 4.10:** PCA visualizations of security and non-security patch embeddings by baselines and LEOPARD.

relevant for the task of security patch detection.

#### 4.5.3.2 Case studies

Table 4.6 presents 2 examples to illustrate the difference between LLMDA and baselines (ie., GraphSPD, TwinRNN, Vulfixminer, and CodeT5) in terms of what the representations can capture, and potentially explaining why LLMDA was successful on these cases while GraphSPD (previous the state-of-the-art work) was not. For our classification task, we have two labels: security (0) and non-security (1). For LLMDA, we can directly consider the label name in the instruction. Thus we compute the attention map between security and the tokens in the patch, the explanation, and the description. For GraphSPD, however, since there no real label name involved in the training and inference phases, we compute the attention score between the words in the patch and the number “0” or “1”. To simplify the analysis, we only highlight, in Table 4.6, tokens for which the similarity score is higher than 0.5.

As shown in the examples, LLMDA generally assigns high similarity scores to security-related aspects, suggesting a detection capability that nuances between tokens. For example, in the sgminer.c patch, LLMDA gives high scores to *realloc* and *mutex\_init*, indicating a finer sensitivity to potential security implications within these code parts. Similarly, in the krb5/auth\_context.c patch, the use of *memset* for initializing authenticator memory is scored high in LLMDA, reflecting its more acute recognition of security practices.

In contrast, since GraphSPD is graph-based, it focuses on the patch itself. For the same patch cases, GraphSPD can even give very high attention scores for non-security label. For example, *krb5\_auth\_con\_init* is given 0.6 score for “non-security” and *ALLOC* is given 0.67 attention score towards non-security as well. These scores may justify many failures of GraphSPD in the security patch detection task.

Apart from GraphSPD, we also compute the attention maps for TwinRNN, Vulfixminer,

**Table 4.6:** Attention scores for security and non-security labels by GraphSPD and LLMDA on two sample patches

GraphSPD (failed cases)			LLMDA (successful cases)		
Patch (non-formatted token sequence)	Developer Description	Patch (non-formatted token sequence)	Explanation	Description	
<pre>diff -git a/sgminer.c b/sgminer.cindex a7dd3ab3..08697cd0 100644-- a/sgminer.c+++ b/sgminer.c@ -518,7 +518,7 @@ struct pool +add_pool (void) (non-security score = 0.65) sprintf(buf, "Pool %d", pool-&gt;_no);pool-&gt;poolname = strdup(buf);-tpools = realloc(pools, sizeof(struct pool *) (total_pools + 2));+tpools = (struct **pool **)realloc(pools, sizeof(struct pool *) * (total_pools + 2));pools[total_pools++] = pool;mutex_init(&amp;pool-&gt;pool_lock); if(unlikely(pthread_cond_init(&amp;pool-&gt;cr_cond, NULL)))</pre>	Fixed missing realloc removed by mistake	<pre>diff -git a/sgminer.c b/sgminer.cindex a7dd3ab3..08697cd0 100644-- a/sgminer.c+++ b/sgminer.c@ -518,7 +518,7 @@ struct pool *add_pool(void) sprintf(buf, "Pool %d", pool-&gt;_no);pool-&gt;poolname = strdup(buf);-tpools = realloc(pools, sizeof(struct pool *) (total_pools + 2));+tpools = (struct **pool **)realloc(pools, sizeof(struct pool *) * (total_pools + 2)); (security core = 0.57) pools[total_pools++] = pool;mutex_init(security core = 0.50)(&amp;pool-&gt;pool_lock); if(unlikely(pthread_cond_init(&amp;pool-&gt;cr_cond, NULL)))</pre>	Modified sgminer.c, adjusted realloc usage (security core = 0.67) with explicit type casting (security core = 0.63)	Fixed missing realloc (security core = 0.63) removed by mistake	
<pre>diff -git a/lib/krb5/auth_context.c b/lib/krb5/auth_context.c index 0dea5418..3cba484e1 100644 -- a/lib/krb5/auth_context.c +++ b/lib/krb5/auth_context.c @@ -53,6 +53,7 @@ krb5_auth_con_init(krb5_context context, ALLLOC(p-&gt;authenticator, 1); if (!p-&gt;authenticator) return ENOMEM; + memset(p-&gt;authenticator, 0, sizeof(*p-&gt;authenticator)); p-&gt;flags = KRB5_AUTH_CONTEXT_DO_TIME;</pre>	zero authenticator	<pre>diff -git a/lib/krb5/auth_context.c b/lib/krb5/auth_context.c index 0dea5418..3cba484e1 100644 -- a/lib/krb5/auth_context.c +++ b/lib/krb5/auth_context.c @@ -53,6 +53,7 @@ krb5_auth_con_init(krb5_context context, ALLLOC(p-&gt;authenticator, 1); if (!p-&gt;authenticator) return ENOMEM; + memset (p-&gt;authenticator, 0, sizeof(*p-&gt;authenticator));(security core = 0.59) p-&gt;flags = KRB5_AUTH_CONTEXT_DO_TIME;</pre>	Added memset to initialize 'authenticator' memory in krb5_auth_con_init function (security core = 0.84).	zero authenticator (security core = 0.77)	
TwinRNN (failed cases)			Vulfixminer (failed cases)		
Patch (non-formatted token sequence)	Patch (non-formatted token sequence)	Patch (non-formatted token sequence)	Patch (non-formatted token sequence)	Patch (non-formatted token sequence)	
<pre>diff -git a/sgminer.c b/sgminer.cindex a7dd3ab3..08697cd0 100644-- a/sgminer.c+++ b/sgminer.c@ -518,7 +518,7 @@ struct pool *add_pool(void) sprintf(buf, "Pool %d", pool-&gt;_no);pool-&gt;poolname = strdup(buf);-tpools = realloc(pools, sizeof(struct pool *) (total_pools + 2))(non-security score = 0.66);+tpools = (struct **pool **)realloc(pools, sizeof(struct pool *) * (total_pools + 2));pools[total_pools++] = pool;mutex_init(&amp;pool-&gt;pool_lock) (non-security score = 0.52); if(unlikely(pthread_cond_init(&amp;pool-&gt;cr_cond, NULL)))</pre>	<pre>diff -git a/sgminer.c b/sgminer.cindex a7dd3ab3..08697cd0 100644-- a/sgminer.c+++ b/sgminer.c@ -518,7 +518,7 @@ struct pool *add_pool(void) sprintf(buf, "Pool %d", pool-&gt;_no);pool-&gt;poolname = strdup(buf);-tpools = realloc(pools, sizeof(struct pool *) (total_pools + 2));+tpools = (struct **pool **)realloc(pools, sizeof(struct pool *) * (total_pools + 2));pools[total_pools++] = pool;mutex_init(&amp;pool-&gt;pool_lock) (non-security score = 0.59); if(unlikely(pthread_cond_init(&amp;pool-&gt;cr_cond, NULL)))</pre>	<pre>diff -git a/sgminer.c b/sgminer.cindex a7dd3ab3..08697cd0 100644-- a/sgminer.c+++ b/sgminer.c@ -518,7 +518,7 @@ struct pool *add_pool(void) sprintf(buf, "Pool %d", pool-&gt;_no);pool-&gt;poolname = strdup(buf);-tpools = realloc(pools, sizeof(struct pool *) (total_pools + 2));+tpools = (struct **pool **)realloc(pools, sizeof(struct pool *) * (total_pools + 2));pools[total_pools++] = pool;mutex_init(&amp;pool-&gt;pool_lock) (non-security score = 0.52); if(unlikely(pthread_cond_init(&amp;pool-&gt;cr_cond, NULL)))</pre>	<pre>diff -git a/sgminer.c b/sgminer.cindex a7dd3ab3..08697cd0 100644-- a/sgminer.c+++ b/sgminer.c@ -518,7 +518,7 @@ struct pool *add_pool(void) sprintf(buf, "Pool %d", pool-&gt;_no);pool-&gt;poolname = strdup(buf);-tpools = realloc(pools, sizeof(struct pool *) (total_pools + 2));+tpools = (struct **pool **)realloc(pools, sizeof(struct pool *) * (total_pools + 2));pools[total_pools++] = pool;mutex_init(&amp;pool-&gt;pool_lock) (non-security score = 0.52); if(unlikely(pthread_cond_init(&amp;pool-&gt;cr_cond, NULL)))</pre>	<pre>diff -git a/lib/krb5/auth_context.c b/lib/krb5/auth_context.c index 0dea5418..3cba484e1 100644 -- a/lib/krb5/auth_context.c +++ b/lib/krb5/auth_context.c @@ -53,6 +53,7 @@ krb5_auth_con_init(non-security score = 0.64) (krb5_context context, ALLOC(non-security score = 0.56) (p-&gt;authenticator, 1)(non-security score = 0.68); if (!p-&gt;authenticator) return ENOMEM; + memset (p-&gt;authenticator, 0, sizeof(*p-&gt;authenticator)); p-&gt;flags = KRB5_AUTH_CONTEXT_DO_TIME;</pre>	<pre>diff -git a/lib/krb5/auth_context.c b/lib/krb5/auth_context.c index 0dea5418..3cba484e1 100644 -- a/lib/krb5/auth_context.c +++ b/lib/krb5/auth_context.c @@ -53,6 +53,7 @@ krb5_auth_con_init(non-security score = 0.64) (krb5_context context, ALLOC(p-&gt;authenticator, 1)(non-security score = 0.68); if (!p-&gt;authenticator) return ENOMEM; + memset (p-&gt;authenticator, 0, sizeof(*p-&gt;authenticator)); p-&gt;flags = KRB5_AUTH_CONTEXT_DO_TIME;</pre>

and CodeT5. As shown in Table 4.6, Vulfixminer and CodeT5 exhibit similar behaviors, highlighting tokens such as 'mutex\_init(&pool->pool\_lock)', 'krb5\_auth\_con\_init', and 'ALLOC'. TwinRNN, on the other hand, demonstrates attention patterns more aligned with GraphSPD. These models fail due to several reasons. TwinRNN, like GraphSPD, emphasizes structural relationships in the patch but lacks semantic understanding, leading to poor differentiation of security-critical tokens. For instance, it assigns high scores to 'ALLOC' in the 'krb5\_auth\_con\_init' patch but overlooks 'memset', which has a clear security implication in ensuring memory safety. Similarly, Vulfixminer and CodeT5 focus on surface-level patterns and overgeneralize their relevance, treating tokens like 'mutex\_init' as equally important in both security and non-security contexts. This results in a failure to prioritize tokens with

actual security implications, such as explicit casting in ‘realloc’ usage in the ‘sgminer.c’ patch or memory initialization with ‘memset’ in the ‘krb5\_auth\_con\_init’ patch. Moreover, none of these models effectively integrate domain-specific knowledge to capture the subtle semantics of secure coding practices, which limits their ability to identify security-critical patches. In contrast, LLMDA excels in capturing these nuances by leveraging contextual and domain-aware information, as reflected in its superior attention scores in Table 4.6.

**[RQ-3]** The design of LLMDA leads to patch representations that enable enhanced ability over GraphSPD and other baselines (ie., TwinRNN, Vulfixminer, CodeT5) in effectively differentiating between security and non-security patches on the embedding spaces. Our analysis of sample cases shows that LLMDA assigns high attention scores to tokens associated to security-related aspects, making it effective for accurately identifying security patches.

#### 4.5.4 Robustness of LLMDA

A model is accepted as robust if it performs strongly on datasets that differ from the training data. For our study task, robustness should ensure reliable predictions on unseen patches. We assess the robustness of LLMDA and baselines (ie., GraphSPD, TwinRNN, Vulfixminer, and CodeT5) by training them against the PatchDB and testing against the samples from the FFmpeg dataset used to construct the benchmark for Devign Zhou et al. [2019] vulnerability detector. This test data includes 13,962 data points, consisting of 8,000 security-related and 5,962 non-security-related patches. The selection of the FFmpeg dataset is motivated by its coverage of a wide range of vulnerabilities.

**Table 4.7:** Performance (%) of GraphSPD, LLMDA, and additional baselines against unseen patches from the FFmpeg dataset Zhou et al. [2019].

Numbers between parentheses ( $\downarrow x$ ) correspond to the drop of performance when compared to the evaluation in cross-validation with PatchDB.

Method	Accuracy	Precision	AUC	Recall	+Recall	-Recall	F1
TwinRNN	40.23	45.10	41.50	34.75	34.75 ( $\downarrow 42.85$ )	49.40	38.12 ( $\downarrow 13.50$ )
Vulfixminer	41.85	47.00	43.12	35.90	35.90 ( $\downarrow 41.90$ )	50.50	39.10 ( $\downarrow 12.80$ )
CodeT5	45.12	50.45	46.75	39.00	39.00 ( $\downarrow 38.45$ )	53.20	43.75 ( $\downarrow 11.59$ )
GraphSPD	43.65	51.15	44.81	36.88	36.88 ( $\downarrow 38.29$ )	52.75	42.86 ( $\downarrow 11.74$ )
LLMDA	66.78	72.70	66.69	67.30	67.30 ( $\downarrow 12.92$ )	66.09	69.89 ( $\downarrow 8.30$ )

\* ( $\downarrow x.x$ ) measures the performance drop when comparing with the cross-validation of LLMDA on PatchDB (cf. Table 5.6).

Table 4.7 summarizes the performance results of LLMDA, GraphSPD, and additional baselines on the unseen dataset. Overall, LLMDA demonstrates superior robustness compared to all baselines. It achieves significantly higher scores across all metrics, with a notable advantage of about 20 percentage points in precision over GraphSPD and a greater performance gap with other baselines. The results further highlight the performance drop when testing on unseen data compared to cross-validation on PatchDB. For example, LLMDA loses approximately 13 percentage points in +Recall, while GraphSPD shows a more substantial drop of 38 percentage points.

Moreover, the token-based and sequential baselines (TwinRNN, Vulfixminer, CodeT5) exhibit the lowest robustness. These methods fail to capture cross-project and unseen patterns due to their reliance on localized or syntactic features. By contrast, LLMDA effectively leverages multi-modal inputs, semantic explanations, and alignment strategies to generalize to diverse datasets. These findings underscore LLMDA’s ability to deliver reliable predictions

on unseen patches, setting a new standard for robustness in security patch detection.

**[RQ-4] ↗ Experiments on unseen patches clearly demonstrate that LLMDA is more robust than GraphSPD and prior baselines. In terms of identifying security patches, LLMDA’s performance drop is about threefold smaller compared to the prior state-of-the-art GraphSPD under the same experimental settings.**

### 4.5.5 Effect of Different Models and Different Instructions

**Table 4.8:** Performance metrics (%) on security patch detection with different instructions

Model	Instruction Type	Dataset	AUC	F1	+Recall	-Recall
LLMDA(Gemini 2.0 Flash)	Task-Specific	PatchDB	76.45	69.23	72.45	80.34
	Detailed	PatchDB	74.67	68.00	70.12	79.12
	Concise	PatchDB	73.12	66.34	69.00	77.89
	Direct Explanation	PatchDB	75.78	68.56	71.23	79.67
	Task-Specific	SPI-DB	64.56	55.67	65.34	72.12
	Detailed	SPI-DB	63.12	54.12	64.00	71.00
	Concise	SPI-DB	62.45	53.67	63.23	70.12
	Direct Explanation	SPI-DB	63.89	54.67	64.45	71.67
	Task-Specific	PatchDB	81.12	74.45	78.34	84.23
	Detailed	PatchDB	79.56	72.67	76.45	82.67
LLMDA(Claude 3.5 Sonnet)	Concise	PatchDB	77.89	71.12	75.00	81.12
	Direct Explanation	PatchDB	80.12	73.45	77.12	83.56
	Task-Specific	SPI-DB	67.34	57.34	68.89	76.45
	Detailed	SPI-DB	66.12	56.00	67.12	75.12
	Concise	SPI-DB	65.34	55.34	66.00	74.00
	Direct Explanation	SPI-DB	66.89	56.34	67.67	75.89
	Task-Specific	PatchDB	76.45	68.34	72.10	79.67
	Detailed	PatchDB	74.78	67.12	70.45	78.34
	Concise	PatchDB	72.67	65.00	69.12	76.89
	Direct Explanation	PatchDB	75.23	67.89	71.34	78.90
LLMDA (Mixtral 8x7B)	Task-Specific	SPI-DB	64.12	54.23	64.67	71.89
	Detailed	SPI-DB	63.34	53.45	63.89	70.78
	Concise	SPI-DB	62.00	52.67	63.00	70.00
	Direct Explanation	SPI-DB	63.45	53.89	64.12	71.34
	Task-Specific	PatchDB	75.78	68.00	71.45	79.12
	Detailed	PatchDB	74.12	66.67	70.00	78.00
	Concise	PatchDB	72.34	65.00	68.78	76.34
	Direct Explanation	PatchDB	74.89	67.12	70.89	78.67
	Task-Specific	SPI-DB	63.45	53.89	63.78	71.23
	Detailed	SPI-DB	62.34	53.00	62.45	70.00
LLMDA(Gemma 7B)	Concise	SPI-DB	61.78	52.34	62.12	69.56
	Direct Explanation	SPI-DB	62.89	53.34	63.34	70.67
	Task-Specific	PatchDB	77.23	69.67	73.34	81.12
	Detailed	PatchDB	75.67	68.00	72.23	79.45
	Concise	PatchDB	74.12	66.45	70.12	78.12
	Direct Explanation	PatchDB	76.45	68.78	72.89	80.23
	Task-Specific	SPI-DB	65.34	55.45	66.45	73.00
	Detailed	SPI-DB	64.12	54.23	65.34	72.00
	Concise	SPI-DB	63.45	53.34	64.12	71.12
	Direct Explanation	SPI-DB	64.78	54.89	65.89	72.34
LLMDA(gpt3.5-turbo)	Task-Specific	PatchDB	84.49 (± 0.51)	78.19 (± 0.37)	80.22 (± 0.21)	87.33 (± 0.24)
	Detailed	PatchDB	82.34 (± 0.48)	75.89 (± 0.33)	78.34 (± 0.20)	85.67 (± 0.22)
	Concise	PatchDB	80.56 (± 0.45)	73.34 (± 0.30)	76.12 (± 0.18)	84.12 (± 0.21)
	Direct Explanation	PatchDB	81.78 (± 0.47)	74.45 (± 0.32)	77.45 (± 0.19)	84.89 (± 0.22)
	Task-Specific	SPI-DB	68.98 (± 0.27)	58.13 (± 0.33)	70.94 (± 0.13)	80.62 (± 0.22)
	Detailed	SPI-DB	67.89 (± 0.26)	56.78 (± 0.30)	68.45 (± 0.12)	79.45 (± 0.21)
	Concise	SPI-DB	66.45 (± 0.25)	55.23 (± 0.28)	67.12 (± 0.11)	78.67 (± 0.20)
	Direct Explanation	SPI-DB	67.12 (± 0.26)	56.00 (± 0.29)	68.00 (± 0.12)	79.00 (± 0.21)

The quality and effectiveness of explanations generated for LLMDA are significantly influenced by both the choice of model and the type of instruction provided. To evaluate these factors, we conducted a systematic ablation study using five different large language models (LLMs) (Gemini 2.0 Flash OpenAI [2023], Claude 3.5 Sonnet Anthropic [2023], Mixtral 8x7B Face [2023], Gemma 7B Labs [2023], and Llama3 70B AI [2023]) and four types of instructions (Task-Specific, Detailed, Concise, Direct Explanation). Below, we

**Table 4.9:** Comparison of Instruction Types for Explanation Generation

Instruction Type	Example Instruction	Characteristics
Task-Specific	Choose the correct option to the following question: Is the patch security related or not? Choices: (0) security (1) non-security.	Constructed specifically for classification objectives; produces interpretations that explicitly address vulnerability significance considerations, creating optimally applicable outputs for subsequent processing components.
Detailed	Could you explain the purpose of this patch and describe why the change was made?	Delivers comprehensive explanatory content addressing both intended functionality and implementation consequences within programming modifications, establishing extensive contextual frameworks supporting architectural learning processes.
Concise	Summarize the purpose of this patch in one sentence.	Creates abbreviated focused interpretive content; while providing reduced granularity, these directive parameters maintain operational efficiency while delivering fundamental modification insights.
Direct Explanation	Explain what happened inside the code changes.	Concentrates on operational transformation descriptions within modification implementation; emphasizes programming alterations without necessarily establishing connections to broader conceptual intentions or vulnerability implications.

provide an intrinsic analysis of how these variations impact security patch detection.

**4.5.5.0.1 Effect of Different Instructions.** The instruction type used to generate explanations significantly affects their quality and relevance, which in turn impacts the downstream performance of LLMDA.

**Task-Specific Instructions.** For instance: *"Choose the correct option to the following question: Is the patch security related or not? Choices: (0) security (1) non-security."* These instructions align closely with the binary classification task, generating explanations that directly assess the security relevance of patches. The tight alignment ensures better feature extraction and improved context understanding, leading to superior metrics such as F1 and +Recall. Recent studies Wei et al. [2021], Chung et al. [2022] have demonstrated the effectiveness of task-specific prompts in enhancing downstream performance.

**Detailed Instructions.** For example: *"Could you explain the purpose of this patch and describe why the change was made?"* These prompts produce comprehensive explanations, offering both the intent and functional impact of the code change. While informative, detailed instructions can include extraneous information not directly relevant to the classification task, slightly diluting performance compared to task-specific prompts. Nevertheless, the additional context is particularly useful for datasets with sparse metadata, such as SPI-DB Zhou et al. [2021b].

**Concise Instructions.** For instance: *"Summarize the purpose of this patch in one sentence."* Concise instructions prioritize brevity, generating short explanations that provide essential insights. While efficient, these instructions often lack the depth required to

fully capture the intent and implications of the patch, resulting in lower +Recall and F1 scores Brown et al. [2020].

**Direct Explanation Instructions.** For example: *"Explain what happened inside the code changes."* These prompts focus on describing the specific modifications made in the patch without linking them to broader security implications. While useful for technical analysis, the lack of semantic insights necessary for security classification results in comparatively lower performance, consistent with findings in Sun et al. [2023].

**4.5.5.0.2 Effect of Different Models.** The choice of LLM affects the quality of generated explanations, including relevance, coherence, and semantic richness, which significantly impacts LLMDA's classification performance.

**Gemini 2.0 Flash and Claude 3.5 Sonnet.** These models perform consistently well, with Claude 3.5 Sonnet achieving the strongest results when paired with task-specific or detailed instructions. Their advanced contextual understanding enables them to generate high-quality explanations that balance detail and relevance.

**Mixtral 8x7B and Gemma 7B.** These models exhibit moderate performance across all instructions. Their relatively smaller architectures limit their ability to generate nuanced semantic explanations, leading to lower performance on complex datasets like SPI-DB.

**Llama3 70B.** Despite its size, Llama3 70B shows variability depending on the instruction type. It performs well with task-specific instructions but struggles with concise prompts, likely due to over-reliance on syntactic patterns.

**LLMDA.** As the proposed model, LLMDA consistently outperforms all other LLMs across instructions. Task-specific instructions paired with LLMDA yield the highest performance, demonstrating its ability to leverage semantically aligned explanations and integrate them effectively into its multi-modal architecture.

**4.5.5.0.3 Dataset-Specific Insights. PatchDB.** The diversity of repositories in PatchDB highlights the importance of high-quality explanations. Task-specific and detailed instructions enable LLMDA to generalize effectively across various patch styles. **SPI-DB.** The sparse commit messages and imbalanced features in SPI-DB pose challenges for all configurations. However, detailed instructions mitigate these issues by providing additional context, while task-specific instructions remain the most effective overall due to their alignment with the classification objective.

**Conclusion** *The section discusses that both the choice of LLM and the instruction type significantly influence the performance of LLMDA. Task-specific instructions paired with high-capability models like Claude 3.5 Sonnet and Gemini 2.0 Flash yield superior results, highlighting the importance of aligning explanations with the classification task. Detailed instructions improve robustness in sparse datasets like SPI-DB, while concise and direct explanations underperform due to their lack of contextual depth. These findings emphasize the critical role of prompt engineering and model selection in enhancing explanation quality and downstream performance.*

## 4.5.6 Extended Evaluation of Instruction

We evaluate the performance by conducting experiments on the following: Instruction Positions and Importance of Instruction.

**4.5.6.0.1 Importance of Instruction Positions** The position of the instruction within the input sequence significantly affects the model's ability to leverage task-specific guidance. As shown in Table 4.11, placing the instruction at the end of the sequence yields the highest

performance, with an AUC of 84.49% and an F1 score of 78.19%, outperforming configurations where the instruction is placed at the beginning or middle. This result aligns with the transformer architecture's tendency to assign greater attention to later tokens, allowing the end-positioned instruction to better influence the final embedding. Conversely, positioning the instruction at the beginning or middle may dilute its significance as subsequent tokens are processed, leading to a slight decrease in performance. These findings underscore the importance of aligning input design choices with the model's architectural characteristics to maximize the utility of task-specific instructions.

**Table 4.10:** Performance (%) with Different Instruction Positions

Configuration	Dataset	AUC	F1	+Recall	-Recall
<b>Instruction at Beginning</b>	PatchDB	82.45	76.12	78.45	85.89
<b>Instruction at Middle</b>	PatchDB	83.23	77.00	79.23	86.34
<b>Instruction at End (Original)</b>	PatchDB	<b>84.49</b>	<b>78.19</b>	<b>80.22</b>	<b>87.33</b>

**4.5.6.0.2 Importance of Instruction** Furthermore, we also evaluate the impact of instruction by replacing the instruction with a sentence composed of random words generated from a random sentence Web provider<sup>5</sup>. Here, we take two random sentences as examples: Sentence<sub>A</sub>: “He stepped gingerly onto the bridge knowing that enchantment awaited on the other side.” Sentence<sub>B</sub>: “When confronted with a rotary dial phone, the teenager was perplexed.”

The performance results for these configurations are presented in Table 4.11. As shown, using random sentences as instructions leads to a noticeable performance drop compared to the original task-specific instruction used in LLMDA. Specifically, both Sentence<sub>A</sub> and Sentence<sub>B</sub> achieve lower AUC, F1, +Recall, and -Recall metrics than the original configuration.

**Table 4.11:** Impact of Instruction

Configuration	Dataset	AUC	F1	+Recall	-Recall
Sentence <sub>A</sub>	PatchDB	75.34	70.12	71.45	78.90
Sentence <sub>B</sub>	PatchDB	76.05	71.00	72.12	79.50
<b>LLMDA (Original)</b>	PatchDB	<b>84.49</b>	<b>78.19</b>	<b>80.22</b>	<b>87.33</b>

The results indicate that replacing the task-specific instruction with random sentences introduces noise and misleads the training process. Unlike the task-specific instruction, which provides direct guidance aligned with the classification objective (security-related or non-security-related patches), random sentences lack semantic relevance and coherence in relation to the task. This misalignment results in a reduced ability of the model to focus on meaningful features and relationships within the data, leading to degraded performance across all metrics.

The experimental results highlights the importance of using instructions in LLMDA. Instruction serves as a guiding mechanism, focusing the model's attention on the critical aspects of the problem and enhancing its ability to learn meaningful representations for classification. In contrast, random sentences introduce irrelevant information, diluting the signal and adversely affecting the model's effectiveness. This underscores the necessity of carefully designed instructions to ensure optimal training and robust model performance.

<sup>5</sup><https://randomwordgenerator.com/sentence.php>

## 4.6 Discussion

### 4.6.1 Threats to Validity

**Internal validity.** The initial methodological concern involves generative interpretation quality. Neural language architectures may produce inaccurate programming modification descriptions or conversely demonstrate exceptional performance within established evaluation repositories, potentially introducing assessment bias. Mitigation strategies incorporate contemporary foundation model implementation alongside comprehensive component elimination analysis examining interpretative contributions. A secondary concern addresses architectural evolution within deployed generative systems. These developmental modifications introduce experimental reproducibility challenges through performance variability, potentially affecting consistency across identical parametrized requests. A tertiary constraint emerges from dimensional limitations restricting inputs to 512 representational units. When processing extensive modifications, LLMDA implements truncation procedures potentially eliminating critical information elements affecting classification precision and operational reliability. **External validity.** Methodological constraints exist regarding evaluation datasets potentially limiting generalization beyond established repositories. Specifically, SPI-DB provides limited project diversity (2 implementations). Mitigation approaches include implementation across multiple distinct evaluation repositories, with PatchDB encompassing over 300 project implementations. Additionally, LLMDA implements linguistic-oriented processing frameworks ensuring architectural design remains implementation-language independent. Additional concerns emerge regarding pre-trained architectural components (CodeT5 and LLaMa-7b) providing initial representational processing. These architectures potentially lack optimal adaptation for vulnerability classification requirements. Mitigation considerations incorporate literature-validated selection criteria demonstrating superior performance characteristics across related computational objectives. **Construct validity.** Experimental limitations exist regarding directive parameter optimization within *Instruction* components. This potentially limits comprehensive understanding regarding instructional contributions toward classification performance. Mitigation procedures include component elimination analysis quantifying current parametrization effectiveness. Enhanced directive formulations potentially offer additional performance improvements.

### 4.6.2 Limitation

A significant implementation constraint involves GPT-3.5 architecture (gpt-3.5-turbo-16k-0613) utilization generating programming modification interpretations within LLMDA framework. While this implementation demonstrates significant effectiveness interpreting programming structures, subsequent architectural developments including GPT-4.0 (gpt-4-0613) offer potential enhancement opportunities.

To illustrate this, consider the following patch (non-security) from the Linux kernel, which modifies socket configurations to enable asynchronous I/O operations:

```

1 @@ 1950,6 +1950,7
2     ↗ @ static int xs_local_finish_connecting(struct rpc_xprt *xprt,
3         sk>sk_user_data = xprt;
4         sk>sk_data_ready = xs_data_ready;
5         sk>sk_write_space = xs_udp_write_space;
6         sock_set_flag(sk, SOCK_FASYNC);
7         sk>sk_error_report = xs_error_report;
8         sk>sk_allocation = GFP_NOIO;

```

```

9  @@ -2136,6 2137,7 @@ static void xs_udp_finish_connecting
10     ↪ (struct rpc_xprt *xprt, struct socket *sock)
11     sk>sk_user_data = xprt;
12     sk>sk_data_ready = xs_data_ready;
13     sk>sk_write_space = xs_udp_write_space;
14     + sock_set_flag(sk, SOCK_FASYNC);
15     sk>sk_allocation = GFP_NOIO;
16
17     xprt_set_connected(xprt);
18
18 @@ 2237,6 +2239,7 @@ static int xs_tcp_finish_connecting
19     ↪ (struct rpc_xprt *xprt, struct socket *sock)
20     sk>sk_data_ready = xs_tcp_data_ready;
21     sk>sk_state_change = xs_tcp_state_change;
22     sk>sk_write_space = xs_tcp_write_space;
23     sock_set_flag(sk, SOCK_FASYNC);
24     sk->sk_error_report = xs_error_report;
24     sk->sk_allocation = GFP_NOIO;

```

GPT-4.0 provides a more detailed and contextually rich explanation:

The provided code snippets modify socket configurations in three different functions (xs\_local\_finish\_connecting, xs\_udp\_finish\_connecting, xs\_tcp\_finish\_connecting). They uniformly set the SOCK\_FASYNC flag using `sock_set_flag(sk, SOCK_FASYNC)`, enabling asynchronous I/O operations on the sockets involved. This change likely aims to enhance efficiency by allowing non-blocking handling of network events and data reception across different types of connections (local, UDP, TCP).

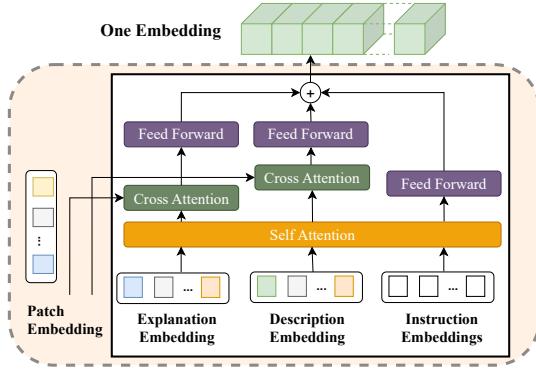
In contrast, GPT-3.5's explanation is more concise:

The changes in the code snippets add the `sock_set_flag(sk, SOCK_FASYNC)` line to three different functions (xs\_local\_finish\_connecting, xs\_udp\_finish\_connecting, and xs\_tcp\_finish\_connecting). This modification enables asynchronous (signal-driven) I/O on the sockets, allowing the program to receive signals when I/O events occur, thereby improving network operations' efficiency and responsiveness.

Comparative analysis demonstrates that while both interpretative implementations maintain factual accuracy, advanced architecture GPT-4.0 provides enhanced contextual understanding regarding programming modification implications—an essential characteristic determining vulnerability significance classifications. The illustrated example demonstrates appropriate non-security classification where implementation emphasizes computational efficiency optimization rather than vulnerability remediation objectives. Subsequent research directions should investigate advanced neural language architecture integration enhancing LLMDA framework interpretative capabilities supporting improved vulnerability modification classification performance. These architectural enhancements potentially facilitate expedited identification of undocumented vulnerability modifications, consequently strengthening protective measures within community-maintained software infrastructures.

#### 4.6.3 Leveraging Patch Descriptions for Enhanced Cross-Attention

While the original architecture of LLMDA effectively models the relationships between the code patch and the LLM-generated explanation ( $E_{expl}$ ), it does not fully utilize the



**Figure 4.11:** Cross-Attention Between Patch and Description version of PT-Former potential insights provided by patch descriptions ( $E_{desc}$ ) when they are available. To address this limitation, we enhance the architecture by introducing an additional cross-attention mechanism between the patch embedding ( $E_p$ ) and the description embedding ( $E_{desc}$ ).

Figure 4.11 illustrates the updated architecture of the PT-Former module, which now includes cross-attention layers for both the explanation and the description. The updated PT-Former processes the input embeddings as follows: 1) Cross-Attention Between Patch and Explanation ( $E_p \leftrightarrow E_{expl}$ ): This captures the semantic relationship between the code patch and its generated explanation. 2) Cross-Attention Between Patch and Description ( $E_p \leftrightarrow E_{desc}$ ): This models the contextual relevance of the patch description to the code changes. 3) Aggregation: The outputs of the two cross-attention layers are concatenated and passed through feed-forward layers to generate a unified representation of the patch.

In cases where the description ( $E_{desc}$ ) is missing, a masking mechanism is employed to ensure that the model seamlessly defaults to using only  $E_p$  and  $E_{expl}$ , maintaining the robustness of the system.

**Table 4.12:** Performance (%) with Enhanced Cross-Attention

Model	Dataset	AUC	F1	+Recall	-Recall
Original PT-Former	PatchDB	84.49	78.19	80.22	87.33
Enhanced PT-Former	PatchDB	<b>85.25</b>	<b>79.34</b>	<b>81.45</b>	<b>88.12</b>
Original PT-Former	SPI-DB	68.98	58.13	70.94	80.62
Enhanced PT-Former	SPI-DB	<b>70.12</b>	<b>59.45</b>	<b>72.18</b>	<b>81.87</b>

#### 4.6.3.1 Experimental Results

Table 4.12 summarizes the performance of the enhanced PT-Former architecture compared to the original design. On the PatchDB dataset, the enhanced architecture achieves a 0.76% improvement in AUC and a 1.15% increase in F1, reflecting its ability to leverage additional context from patch descriptions. Similarly, on the SPI-DB dataset, the AUC and F1 scores improve by 1.14% and 1.32%, respectively.

The results demonstrate that incorporating cross-attention between the patch and description embeddings significantly enhances the model’s ability to capture the broader context and improve its decision-making process. Importantly, the fallback mechanism ensures that the enhanced architecture remains robust even when descriptions are missing.

**Conclusion** These results highlight the effectiveness of leveraging patch descriptions through cross-attention. By incorporating this additional layer, LLMDA achieves improved performance across all metrics, demonstrating its adaptability to varying input configurations.

## 4.6.4 Limitations of PT-Former: Dependency on Embedding Models

One notable limitation of PT-Former lies in its dependence on the quality of the underlying code and text embedding models, such as CodeT5+ for code and LLaMA for text. These embeddings form the foundation of PT-Former’s input representation, and any inadequacies in the pre-trained models can propagate through the network, impacting the final performance.

**4.6.4.0.1 Impact of Embedding Quality** The effectiveness of PT-Former in aligning and fusing multi-modal inputs heavily relies on the richness and accuracy of the embeddings generated by these models. If the embeddings fail to capture nuanced semantic or structural information, the subsequent alignment and fusion processes may be constrained. For example:

- **Code Embeddings:** If the pre-trained CodeT5+ model struggles to represent certain programming constructs or interactions, PT-Former’s ability to capture inter-functional and inter-modular relationships could be limited.
- **Text Embeddings:** Similarly, LLaMA-generated embeddings might miss critical contextual details or nuances in patch descriptions, reducing the model’s overall interpretability and precision.

**4.6.4.0.2 Generalization Across Domains** Another challenge arises from the generalization capability of the pre-trained models. CodeT5+ and LLaMA are typically fine-tuned on specific datasets, which may not comprehensively cover the diversity of codebases or patch descriptions encountered in real-world scenarios. As a result:

- PT-Former’s performance might degrade on unseen or cross-project datasets, where the embedding models fail to generalize effectively.
- The reliance on static embeddings limits PT-Former’s ability to dynamically adapt to varying code styles, programming languages, or textual nuances across domains.

## 4.6.5 Comparison with CoLeFunDa

While both CoLeFunDa Zhou et al. [2023a] and our approach leverage contrastive learning for silent vulnerability fix identification, the methodologies and contributions differ significantly.

First, CoLeFunDa emphasizes function-level code representation, focusing on CWE classification and exploitability rating alongside vulnerability fix detection. In contrast, our approach targets security patch detection by integrating multi-modal inputs, including code changes, developer-provided descriptions, and LLM-generated explanations. This integration allows LLMDA to capture both syntactic and semantic contexts, addressing limitations of function-level analysis.

Second, CoLeFunDa relies on a contrastive learning framework applied to augmented function slices, without leveraging natural language guidance. Our work innovates by introducing task-specific natural language instructions, which act as guiding mechanisms for contextualizing the data and aligning the model’s focus with the security detection task.

Finally, the architectural designs are distinct. CoLeFunDa employs a function change encoder (FCBERT) for its representations, whereas LLMDA utilizes the PT-Former module. PT-Former aligns and fuses multi-modal inputs through cross-attention and self-attention mechanisms, enabling a unified embedding space that enhances generalization across diverse datasets.

These distinctions underscore the unique contributions and novelty of our work, particularly in addressing limitations in prior methods, including CoLeFunDa.

### 4.6.6 Model Selection for Initial Embeddings

We utilized CodeT5+ and LLaMa-7b as it can function as an encoder-only model while other models like CodeLlama is not inherently designed to function as an encoder-only model, as it is primarily a decoder-only transformer based on the LLaMA. Embedding models often use this configuration to generate dense vector representations of input text or code.

Furthermore, we conducted additional experiments using CodeLLaMA-13B as a unified embedding model for both code and natural language (NL) inputs. Unlike the original LLMDA, which employs CodeT5+ for code and LLaMa-7b for NL, this setup generates all embeddings from a single model, simplifying the alignment process in PT-Former.

The results, presented in Table 4.13, show that CodeLLaMA-13B achieves competitive performance but falls slightly short of the original LLMDA setup. Specifically, LLMDA with CodeLLaMA-13B demonstrates a small drop in AUC and F1 across both datasets. Additionally, the training process required more epochs to converge, likely due to the lack of task-specific optimization for code and NL modalities in a unified model.

These findings reinforce the advantages of using specialized models for multi-modal tasks. While CodeLLaMA-13B provides a simpler deployment setup, the combination of CodeT5+ and LLaMa-7b offers better task-specific performance and faster convergence. Future research could explore fine-tuning unified models like CodeLLaMA for enhanced performance in security patch detection.

**Table 4.13:** Performance (%) and Training Efficiency Comparison: Original LLMDA vs. CodeLLaMA-13B

Model	Dataset	AUC	F1	+Recall	-Recall	Epochs	Training Time
LLMDA <sub>Original</sub>	PatchDB	<b>84.49</b>	<b>78.19</b>	<b>80.22</b>	<b>87.33</b>	20	12.5h
LLMDA <sub>CodeLLaMA-13B</sub>	PatchDB	83.12 (↓ 1.37)	76.78 (↓ 1.41)	78.45	85.67	30	
LLMDA <sub>Original</sub>	SPI-DB	<b>68.98</b>	<b>58.13</b>	<b>70.94</b>	<b>80.62</b>	20	
LLMDA <sub>CodeLLaMA-13B</sub>	SPI-DB	67.51 (↓ 1.47)	56.45 (↓ 1.68)	69.01	78.54	30	18h

“LLMDA<sub>Original</sub>” means LLMDA with CodeT5+ to embed codes and LLaMa-7b to embed texts.

“LLMDA<sub>CodeLLaMA-13B</sub>” means LLMDA with LLMDA to embed codes and texts.

### 4.6.7 Extended Case Studies on the Impact of Explanations and Instructions

To further illustrate the importance of explanations and instructions in LLMDA, we present two additional case studies involving security patches. These cases highlight how the inclusion of natural language explanations and task-specific instructions contributes to accurate predictions, particularly in challenging scenarios.

#### 4.6.7.1 Case Study 1: Complex Code Changes with Minimal Documentation

##### Patch:

```

1 diff git a/net/ipv4/tcp_input.c b/net/ipv4/tcp_input.c
2 index c34fa2a..e4fb7a5 100644
3 a/net/ipv4/tcp_input.c
4 +++ b/net/ipv4/tcp_input.c
5 @@ 2342,7 2342,8
6     → @@ void tcp_ack(struct sock *sk, struct sk_buff *skb, int flag)
7
8     /* Fix potential buffer overflow in TCP options */
9     - memcpy(opt, skb->data  tcp_hdrlen, optlen);
10    if (unlikely(optlen > TCP_MAX_OPT_LENGTH))
11        return;
12    memcpy(opt, skb->data  tcp_hdrlen, optlen);
13 }
```

---

**Listing 4.2:** A complex security patch with minimal documentation.

**Baseline Prediction (GraphSPD):** Non-Security

**LLMDA Prediction:** Security

**Reason:** The patch adds a boundary check to prevent buffer overflows, a critical vulnerability. GraphSPD fails to detect the intent due to the absence of a detailed commit message. However, LLMDA uses an LLM-generated explanation (e.g., “This patch mitigates buffer overflow by adding a boundary check”) to capture the semantic significance of the changes. The task-specific instruction further directs the model to prioritize security-related aspects of the patch.

#### 4.6.7.2 Case Study 2: Misleading Commit Message

**Patch:**

```
1 diff git a/src/http.c b/src/http.c
2 index abcd123..dcba321 100644
3 a/src/http.c
4 +++ b/src/http.c
5 @@ 567,7 567,8 @@ int validate_http_request(request *req) {
6     /* Sanitize user inputs to prevent injection */
7     - strcpy(buffer, req->input);
8     if (strlen(req->input) < MAX_LEN) {
9     + strncpy(buffer, req->input, MAX_LEN);
10    }
11    return validate_buffer(buffer);
12 }
```

**Listing 4.3:** A patch with a misleading commit message.

**Commit Message:** "Minor refactor of input validation code."

**Baseline Prediction (CodeT5+):** Non-Security

**LLMDA Prediction:** Security

**Reason:** The misleading commit message downplays the significance of the patch, causing CodeT5+ to miss its security implications. LLMDA relies on LLM-generated explanations to identify that the patch introduces input length checks to prevent buffer overflows, a critical security improvement. The instruction guides the model to focus on identifying security-relevant changes.

#### 4.6.7.3 Analysis of Findings

Evaluation cases demonstrate LLMDA’s capability to accurately determine vulnerability significance within programming modifications where comparative methodologies demonstrate classification failures. Principal architectural advantages include:

- **Explanations:** Neural language system interpretative mechanisms establish connections between programming semantic structures and human-comprehensible contextual frameworks, facilitating architectural understanding regarding implementation objectives underlying programming transformations.
- **Instructions:** Objective-specific procedural directives establish explicit classification parameters supporting vulnerability modification identification, enhancing discriminative decision processes.

## 4.7 Conclusion

This investigation has introduced an innovative methodology, LLMDA, addressing vulnerability modification identification challenges. The architecture implements linguistic-oriented processing strategies confronting undocumented vulnerability modification classification requirements. Initially, LLMDA enhances modification representations through neural language system interpretative mechanisms. Subsequently, the framework establishes dimensional representation systems integrating diverse information sources alongside procedural directives following representational synchronization procedures. The implementation concludes with differential learning mechanisms ensuring optimal classification boundary definition across complex modification patterns. Quantitative evaluation across established research repositories demonstrates LLMDA establishing superior performance metrics within vulnerability modification identification applications. Component elimination analysis confirms architectural design effectiveness while establishing implementation reliability characteristics. **Open Science:** Implementation codebase, evaluation datasets, and experimental outcomes remain publicly accessible through dedicated scientific repository: <https://llmda.github.io>



# 5 CodeAgent: Autonomous Communicative Agents for Code Review

*In this chapter, we propose to investigate the inherently collaborative nature of code review and how it can be effectively modeled through a multi-agent framework. We then propose CodeAgent, an autonomous communicative agent-based system that simulates the dynamics of a collaborative team engaged in the code review process, incorporating diverse roles such as code change authors, reviewers, and decision makers. CodeAgent addresses a fundamental limitation of existing approaches by modeling the interactive and collaborative aspects of code review rather than simply focusing on rewriting and adapting submitted code. A key innovation is our QA-Checker agent that monitors conversation flow to prevent prompt drifting, ensuring that questions and responses stay relevant and aligned with the dialogue's intended objective. Experimentally, our approach significantly outperforms state-of-the-art methods, achieving a 41% increase in hit rate for vulnerability detection, superior performance in consistency checking and format alignment, and approximately 30% improvement in Edit Progress metrics for code revision tasks.*

The content presented herein derives from scholarly investigation previously documented in the academic publication referenced below:

- **Xunzhu Tang**, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawendé F. Bissyandé. CodeAgent: Autonomous Communicative Agents for Code Review. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11279-11313, 2024.

## Contents

---

<b>5.1</b>	<b>Overview</b>	<b>97</b>
<b>5.2</b>	<b>CodeAgent</b>	<b>99</b>
5.2.1	Tasks	99
5.2.2	Pipeline	100
5.2.3	Role Card Definition	101
5.2.4	Self-Improving CoT with QA Checker	101
<b>5.3</b>	<b>Experimental Setup</b>	<b>103</b>
5.3.1	Datasets	103
5.3.2	Metrics	103

5.3.3	State-of-the-Art Tools and Models . . . . .	103
<b>5.4</b>	<b>Experimental Result Analysis . . . . .</b>	<b>105</b>
5.4.1	Vulnerability Analysis . . . . .	105
5.4.2	Consistency and Format Detection . . . . .	106
5.4.3	Code Revision . . . . .	107
<b>5.5</b>	<b>Conclusion . . . . .</b>	<b>108</b>
<b>5.6</b>	<b>Details of QA-Checker Algorithm . . . . .</b>	<b>109</b>
<b>5.7</b>	<b>Complete Related Work . . . . .</b>	<b>111</b>
<b>5.8</b>	<b>Experimental Details . . . . .</b>	<b>112</b>
5.8.1	Role Definition . . . . .	112
5.8.2	Execute Time Across Languages . . . . .	113
<b>5.9</b>	<b>Comparative Analysis of QA-Checker AI System and Recursive Self-Improvement Systems . . . . .</b>	<b>114</b>
5.9.1	Comparison Table . . . . .	114
5.9.2	Differences and Implications . . . . .	114
5.9.3	Importance of QA-Checker in Role Conversations . . . . .	115
5.9.4	Conclusion . . . . .	115
<b>5.10</b>	<b>Capabilities Analysis between CodeAgent and Other Methods . . . . .</b>	<b>115</b>
<b>5.11</b>	<b>Dataset . . . . .</b>	<b>115</b>
<b>5.12</b>	<b>Key Factors Leading to Vulnerabilities . . . . .</b>	<b>117</b>
<b>5.13</b>	<b>Data Leakage Statement . . . . .</b>	<b>117</b>
<b>5.14</b>	<b>Algorithmic Description of CodeAgent Pipeline with QA-Checker . . . . .</b>	<b>117</b>
<b>5.15</b>	<b>Detailed Performance of CodeAgent in Various Languages on VA task</b>	<b>119</b>
<b>5.16</b>	<b>More detailed experimental results on CA and FA tasks . . . . .</b>	<b>120</b>
<b>5.17</b>	<b>Case Study . . . . .</b>	<b>120</b>
5.17.1	Performance on 9 languages . . . . .	121
5.17.2	Difference of CodeAgent-3.5 and CodeAgent-4.0 . . . . .	122
<b>5.18</b>	<b>Ablation study . . . . .</b>	<b>122</b>
<b>5.19</b>	<b>Cost statement . . . . .</b>	<b>125</b>
<b>5.20</b>	<b>Tool . . . . .</b>	<b>128</b>

---

## 5.1 Overview

Code review Bacchelli and Bird [2013], Bosu and Carver [2013], Davila and Nunes [2021] implements a process wherein software maintainers examine and assess code contributions to ensure quality and adherence to coding standards, and identify potential bugs or improvements. In recent literature, various approaches Tufano et al. [2021, 2022] have been proposed to enhance the performance of code review automation. Unfortunately, major approaches in the field ignore a fundamental aspect: the code review process is inherently interactive and collaborative Bacchelli and Bird [2013]. Instead, they primarily focus on rewriting and adapting the submitted code Watson et al. [2022], Thongtanunam et al. [2022], Staron et al. [2020]. In this respect, an effective approach should not only address how to review the submitted code for some specific needs (e.g., vulnerability detection Chakraborty et al. [2021], Yang et al. [2024a]). Still, other non-negligible aspects of code review should also be considered, like detecting issues in code formatting or inconsistencies in code revision Oliveira et al. [2023], Tian et al. [2022c], Panthaplackel et al. [2021]. However, processing multiple sub-tasks requires interactions among employees in different roles in a real code review scenario, which makes it challenging to design a model that performs code review automatically.

Agent-based systems are an emerging paradigm and a computational framework in which autonomous entities (aka agents) interact with each other Li et al. [2023a], Qian et al. [2023], Hong et al. [2023] to perform a task. Agent-based approaches have been proposed to address a spectrum of software engineering tasks Qian et al. [2023], Zhang et al. [2024b], Tang et al. [2023a], Tian et al. [2023], moving beyond the conventional single input-output paradigm due to their exceptional ability to simulate and model complex interactions and behaviors in dynamic environments Xi et al. [2023], Yang et al. [2024b], Wang et al. [2023d]. Recently, multi-agent systems have leveraged the strengths of diverse agents to simulate human-like decision-making processes Du et al. [2023], Liang et al. [2023], Park et al. [2023], leading to enhanced performance across various tasks Chen et al. [2023a], Li et al. [2023e], Hong et al. [2023]. This paradigm is well-suited to the challenge of code review, where multiple reviewers, each with diverse skills and roles, collaborate to achieve a comprehensive review of the code.

**This paper.** Drawing from the success of agent-based collaboration, we propose a **multi-agent-based framework** CodeAgent to simulate the dynamics of a collaborative team engaged in the code review process, incorporating diverse roles such as code change authors, reviewers, and decision makers. In particular, A key contribution of CodeAgent is that we address the challenge of prompt drifting Zheng et al. [2024], Yang et al. [2024d], a common issue in multi-agent systems and Chain-of-Thought (CoT) reasoning. This issue, characterized by conversations that stray from the main topic, highlights the need for strategies to maintain focus and coherence Greyling [2023], Chae et al. [2023]. This drift, often triggered by the model-inspired tangents or the randomness of Large Language Models (LLMs), necessitates the integration of a supervisory agent. We employ an agent named QA-Checker (for "Question-Answer Checker") that monitors the conversation flow, ensuring that questions and responses stay relevant and aligned with the dialogue's intended objective. Such an agent not only refines queries but also realigns answers to match the original intent, employing a systematic approach grounded in a mathematical framework.

To evaluate the performance of CodeAgent, we first assess its effectiveness for typical review objectives such as detecting vulnerabilities 5.4.1 and validating the consistency and alignment of the code format 5.4.2. We then compare CodeAgent with state-of-the-

art generic and code-specific language models like ChatGPT OpenAI [2022] and CodeBERT Feng et al. [2020a]. Finally, we assess the performance of CodeAgent compared to the state-of-the-art tools for code revision suggestions Tufano et al. [2021], Thongtanunam et al. [2022], Tufano et al. [2022]. Since each of these related works presents a specific dataset, we also employ them toward a fair comparison. Additionally, we also collect pull requests from GitHub, featuring an extensive array of commits, messages, and comments to evaluate advanced capabilities. The experimental results reveal that CodeAgent significantly outperforms the state-of-the-art, achieving a 41% increase in hit rate for detecting vulnerabilities. CodeAgent also excels in consistency checking and format alignment, outperforming the target models. Finally, CodeAgent showcases its robustness for code revision by presenting superior average edit progress.

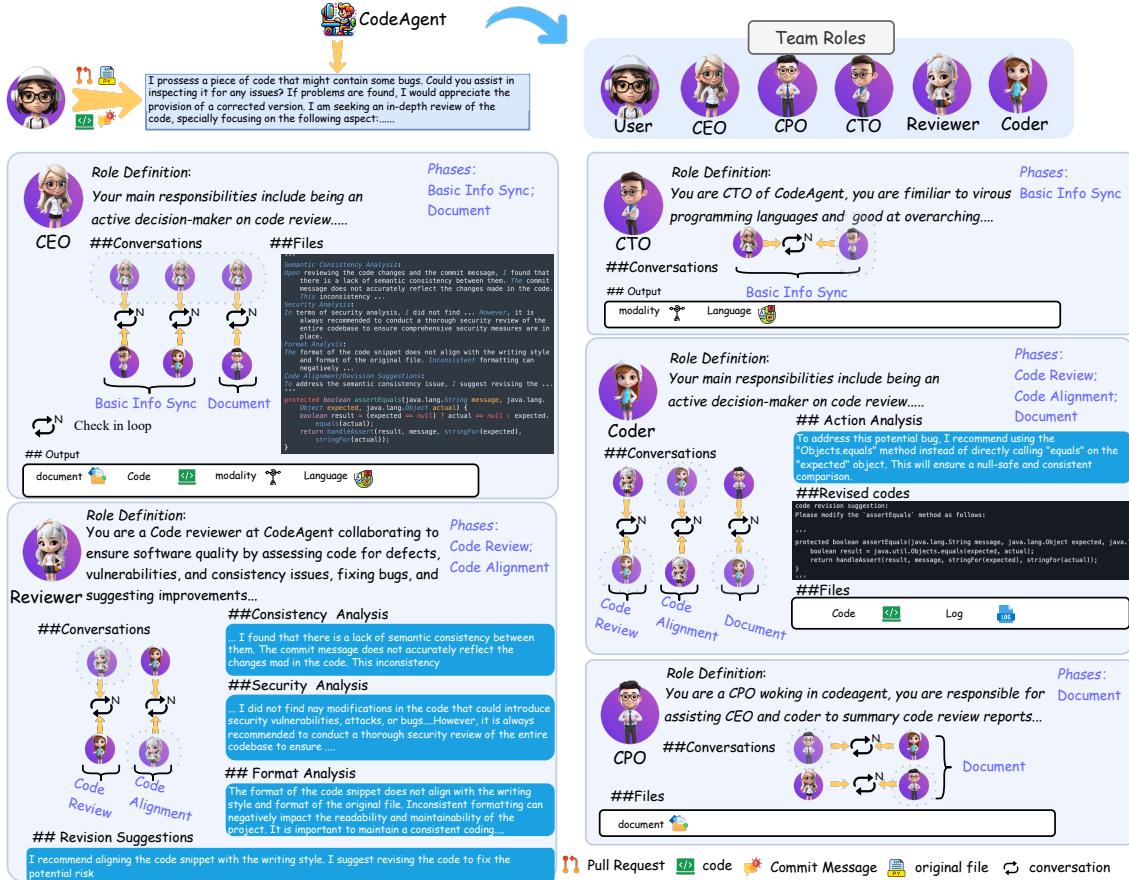
We summarize our contributions as follows:

- To the best of our knowledge, we are the first to propose an autonomous agent-based system for practical code review in the field of software maintenance.
- We build a new dataset comprising 3 545 real-world code changes and commit messages. This dataset, which includes all relevant files and details in a self-contained format, is valuable for evaluating advanced code review tasks such as vulnerability detection, code style detection, and code revision suggestions.
- We demonstrate the effectiveness of the QA-Checker. This agent monitors the conversation flow to ensure alignment with the original intent, effectively addressing the common prompt drifting issues in multi-agent systems.

Experimental evaluation highlights the performance of CodeAgent: In vulnerability detection, CodeAgent outperforms GPT-4 and CodeBERT by 3 to 7 percentage points in terms of the number of vulnerabilities detected. For format alignment, CodeAgent outperforms ReAct by approximately 14% in recall for inconsistency detection. On the code revision task, CodeAgent surpasses the state of the art in software engineering literature, achieving an average performance improvement of about 30% in the Edit Progress metric Zhou et al. [2023b].

## 5.2 CodeAgent

This section details the methodology behind our CodeAgent framework. We discuss tasks and definition in Sec 5.2.1, pipeline in Section 5.2.2, defined role cards in Section 5.2.3, and the design of the QA-Checker in Sec 5.2.4.



**Figure 5.1: A Schematic diagram of role data cards of simulated code review team and their conversations within CodeAgent.** We have six characters in CodeAgent across four phases, including “Basic Info Sync”, “Code Review”, “Code Alignment”, and “Document”. Code review is a kind of collaboration work, where we design conversations between every two roles for every step to complete the task.

## 5.2.1 Tasks

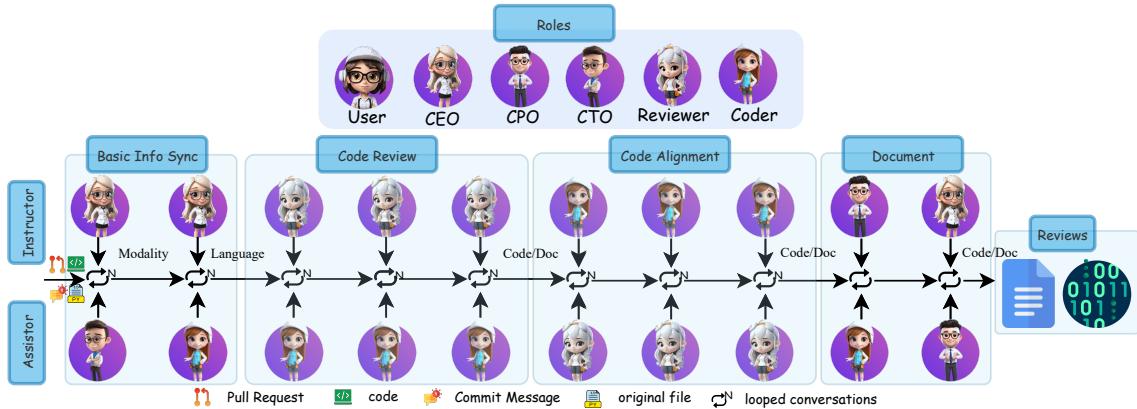
We define  $CA$ ,  $VA$ ,  $FA$ , and  $CR$  in as following:

**CA** Zhang et al. [2022]: Consistency analysis between code change and commit message; the task is to detect cases where the commit message accurately describes (in natural language) the intent of code changes (in programming language).

VA Braz et al. [2022]: Vulnerability analysis; the task is to identify cases where the code change introduces a vulnerability in the code.

**FA** Han et al. [2020]: Format consistency analysis between commit and original files; the task is to validate that the code change formatting style is not aligned with the target code.

**CR** Zhou et al. [2023b]: Code revisions; this task attempts to automatically suggest rewrites of the code change to address any issue discovered.



**Figure 5.2:** CodeAgent’s pipeline/scenario of a full conversation during the code review process among different roles. “Basic Info Sync” demonstrates the basic information confirmation by the CEO, CTO, and Coder; “Code Review” shows the actual code review process; “Code Alignment” illustrates the potential code revision; and “Document” represents the summarizing and writing conclusion for all the stakeholders. All the conversations are being ensured by the Quality Assurance checker until they reach the maximum dialogue turns or meet all the requirements.

## 5.2.2 Pipeline

We defined six characters and four phases for the framework. The roles of the characters are illustrated in Figure 5.1. Each phase contains multiple conversations, and each conversation happens between agents. The four phases consist of 1) Basic Info Sync, containing the roles of chief executive officer (*CEO*), chief technology officer (*CTO*), and Coder to conduct modality and language analysis; 2) Code Review, asking the *Coder* and *Reviewer* for actual code review (i.e., target sub-tasks); 3) Code Alignment, supporting the *Coder* and *Reviewer* to correct the commit through code revision and suggestions to the author; and 4) Document, finalizing by synthesizing the opinions of the *CEO*, *CPO* (Chief Product Officer), *Coder*, and *Reviewer* to provide the final comments. In addition to six defined roles, the proposed architecture of CodeAgent consists of phase-level and conversation-level components. The waterfall model breaks the code review process at the phase level into four sequential phases. At the conversation level, each phase is divided into atomic conversations. These atomic conversations involve task-oriented role-playing between two agents, promoting collaborative communication. One agent works as an instructor and the other as an assistant. Communication follows an instruction-following style, where agents interact to accomplish a specific subtask within each conversation, and each conversation is supervised by QA-Checker. QA-Checker is used to align the consistency of questions and answers between the instructor and the assistant in a conversation to avoid digression. QA-Checker will be introduced in Section 5.2.4.

Figure 5.2 shows an illustrative example of the CodeAgent pipeline. CodeAgent receives the request to do the code review with the submitted commit, commit message, and original files. In the first phase, *CEO*, *CTO*, and *Coder* will cooperate to recognize the modality of input (e.g., document, code) and language (e.g., Python, Java and Go). In the second phase, with the help of *Coder*, *Reviewer* will write an analysis report on consistency analysis, vulnerability analysis, format analysis and suggestions for code revision. In the third phase, based on analysis reports, *Coder* will align or revise the code if any incorrect snippets are identified with assistance from *Reviewer*. *Coder* cooperates with *CPO* and *CEO* to summarize the document and codes about the whole code review in the final phase.

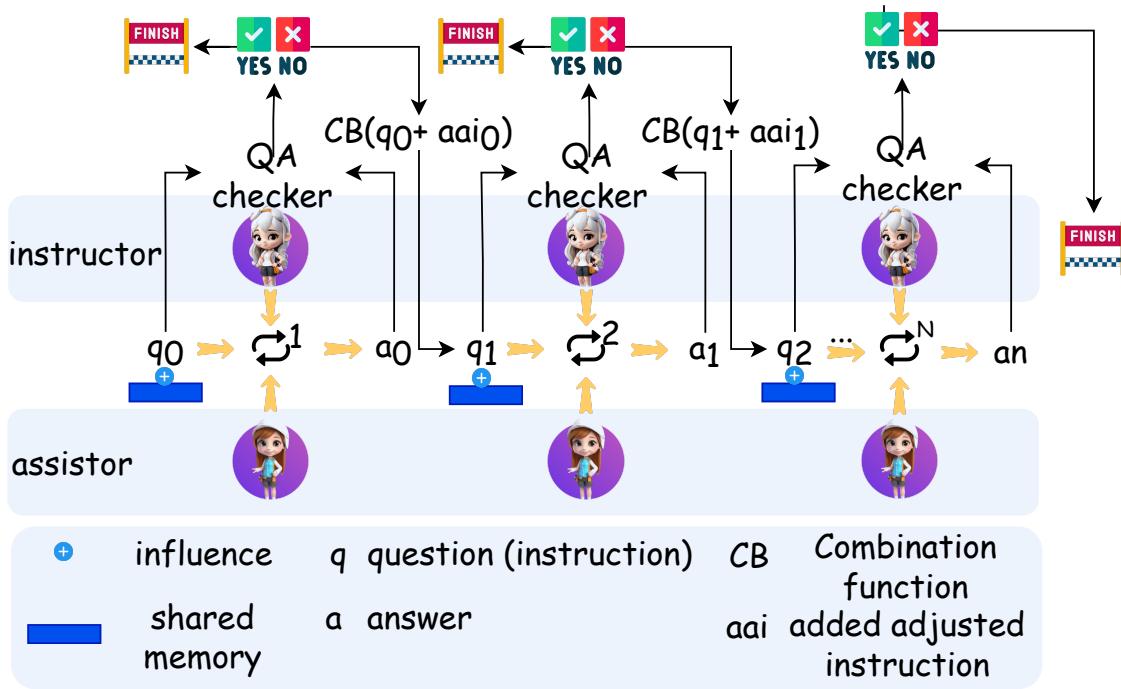
### 5.2.3 Role Card Definition

As shown in Figure 5.1, we define six characters in our simulation system (CodeAgent), including *User*, *CEO*, *CPO*, *CTO*, *Reviewer*, *Coder*, and they are defined for different specific tasks.

All tasks are processed by the collaborative work of two agents in their multi-round conversations. For example, as a role *Reviewer*, her responsibility is to do the code review for given codes and files in three aspects (tasks *CA*, *VA*, and *FA* in Sec 5.2.1) and provide a detailed description of observation. *Reviewer*'s code review activity is under the assistance with *Coder* as shown in Figure 5.2. Meanwhile, with the *Reviewer*'s assistance, *Coder* can process the code revision as shown in the 'Revised codes' part in the *Coder* card in Figure 5.1. Apart from *Reviewer*, *Coder* also cooperates with *CTO* and *CEO* in the simulated team.

Each role and conversation, input and output of each conversation is designed in Figure 5.1. Further information about role definition details is provided in our Appendix-Section 5.8.1.

### 5.2.4 Self-Improving CoT with QA Checker



**Figure 5.3:** This diagram shows the architecture of our designed Chain-of-Thought (CoT): Question-Answer Checker (QA-Checker).

QA-Checker is an instruct-driven agent, designed to fine-tune the question inside a conversation to drive the generated answer related to the question. As shown in Figure 5.3, the initial question (task instruction) is represented as  $q_0$ , and the first answer of the conversation between *Reviewer* and *Coder* is represented as  $a_0$ . If QA-Checker identifies that  $a_0$  is inappropriate for  $q_0$ , it generates additional instructions attached to the original question (task instruction) and combines them to ask agents to further generate a different answer. The combination in Figure 5.3 is defined as  $q_1 = CB(q_0 + aai_0)$ , where  $aai_0$  is the additional instruction attached. The conversation between two agents is held until the generated answer is judged as appropriate by QA-Checker or it reaches the maximum number of dialogue turns.

**5.2.4.0.1 Theoretical Analysis of QA-Checker in Dialogue Refinement** The QA-Checker is an instruction-driven agent, crucial in refining questions and answers within a

conversation to ensure relevance and precision. Its operation can be understood through the following lemma and proof in Appendix 5.6.

## 5.3 Experimental Setup

We evaluate the performance of CodeAgent through various qualitative and quantitative experiments across nine programming languages, using four distinct metrics. In this section, we will discuss experimental settings, including datasets, metrics, and baselines. For more information, please see Appendix 5.8.

### 5.3.1 Datasets

To conduct a fair and reliable comparison for the code revision task, we employ the same datasets (i.e.,  $\text{Trans-Review}_{data}$ ,  $\text{AutoTransform}_{data}$ , and  $\text{T5-Review}_{data}$ ) as the state-of-the-art study Zhou et al. [2023b]. Furthermore, we collect and curate an additional dataset targeting the advanced tasks. Table 5.1 shows our new dataset which includes over 3,545 commits and 2,933 pull requests from more than 180 projects, spanning nine programming languages: Python, Java, Go, C++, JavaScript, C, C#, PHP, and Ruby. It focuses on consistency and format detection, featuring both positive and negative samples segmented by the merged and closed status of pull requests across various languages. The detailed information about the dataset can be seen in Appendix-Section 5.11.

**Table 5.1:** Comparison of Positive and Negative Samples in CA and FA (CA and FA are defined in Section 5.2.1).

Samples	CA		FA	
	Merged	Closed	Merged	Closed
Positive (consistency)	2,089	820	2,238	861
Negative (inconsistency)	501	135	352	94

### 5.3.2 Metrics

- **F1-Score and Recall.** We utilized the F1-Score and recall to evaluate our method’s effectiveness on tasks *CA* and *FA*. The F1-Score, a balance between precision and recall, is crucial for distinguishing between false positives and negatives. Recall measures the proportion of actual positives correctly identified Hossin and Sulaiman [2015].
- **Edit Progress (EP).** EP evaluates the improvement in code transitioning from erroneous to correct by measuring the reduction in edit distance between the original code and the prediction on task *CR*. A higher EP indicates better efficiency in code generation Dibia et al. [2022], Elgohary et al. [2021], Zhou et al. [2023b].
- **Hit Rate (Rate)** We also use hit rate to evaluate the rate of confirmed vulnerability issues out of the found issues by approaches on task *VA*.

### 5.3.3 State-of-the-Art Tools and Models

Our study evaluates various tools and models for code revision and modeling. **Trans-Review** Tufano et al. [2021] employs src2abs for code abstraction, effectively reducing vocabulary size. **AutoTransform** Thongtanunam et al. [2022] uses Byte-Pair Encoding for efficient vocabulary management in pre-review code revision. **T5-Review** Tufano et al. [2022] leverages the T5 architecture, emphasizing improvement in code review through pre-training on code and text data. In handling both natural and programming languages, **CodeBERT** Feng et al. [2020a] adopts a bimodal approach, while **GraphCodeBERT** Guo et al. [2021b] incorporates code structure into its modeling. **CodeT5** Wang et al. [2021d], based on the T5 framework, is optimized for identifier type awareness, aiding in generation-based tasks. Additionally, we compare these tools with **GPT** OpenAI [2022] by OpenAI, notable for its human-like text generation capabilities in natural language processing. Finally,

**Table 5.2:** The number of vulnerabilities found by CodeAgent and other approaches. As described in Appendix-Section 5.11, we have 3,545 items to evaluate. Rate<sub>cr</sub> represents the confirmed number divided by the number of findings while Rate<sub>ca</sub> is the confirmed number divided by the total evaluated number. CodeAgent<sub>w/o</sub> indicates the version without QA-Checker.

	CodeBERT	GPT-3.5	GPT-4.0	COT	ReAct	CodeAgent	CodeAgent <sub>w/o</sub>
Find	1,063	864	671	752	693	483	564
Confirm	212	317	345	371	359	449	413
Rate <sub>cr</sub>	19.94%	36.69%	51.42%	49.34%	51.80%	92.96%	73.23%
Rate <sub>ca</sub>	5.98%	8.94%	9.73%	10.46%	10.13%	12.67%	11.65%

The values in gray (nn.nn) denote the greatest values for the confirmed number of vulnerabilities and the rates.

we involve **COT** Wei et al. [2022] and **ReAct** Yao et al. [2022], of which **COT** is a method where language models are guided to solve complex problems by generating and following a series of intermediate reasoning steps and **ReAct** synergistically enhances language models by interleaving reasoning and action generation, improving task performance and interpretability across various decision-making and language tasks.

## 5.4 Experimental Result Analysis

This section discusses the performance of CodeAgent in the four tasks considered for our experiments. In Appendix Section 5.10, we provide further analyses: we discuss the difference in the execution time of CodeAgent in different languages and perform a capability analysis between CodeAgent and recent approaches.

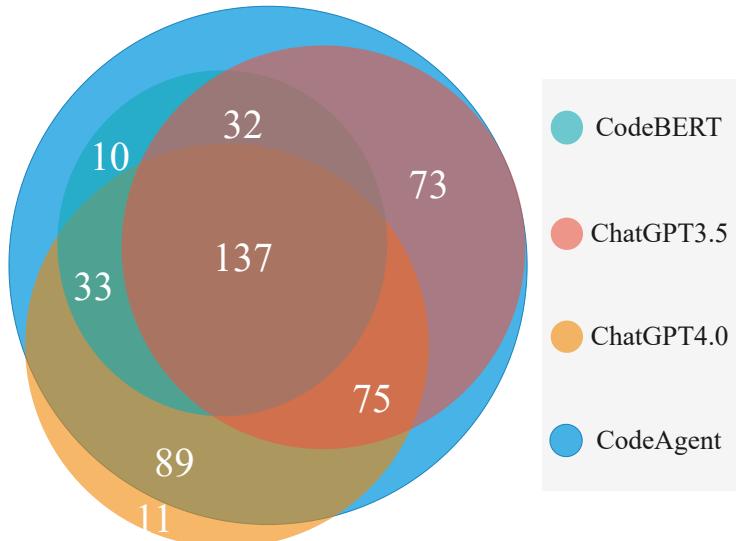
### 5.4.1 Vulnerability Analysis

Compared to *CA* and *FA*, *VA* is a more complex code review subtask, covering more than 25 different aspects (please see the Appendix-Section 5.12), including buffer overflows, sensitive data exposure, configuration errors, data leakage, etc. Vulnerability analysis being a costly, time-consuming, resource-intensive and sensitive activity, only a low proportion of commits are labeled. We therefore propose a proactive method for data annotation: we execute CodeAgent on the 3,545 samples (covering nine languages) and manual verify the identified cases to build a ground truth. Then, we applied CodeBERT Feng et al. [2020a] and GPT on the dataset with the task of vulnerability binary prediction.

**5.4.1.0.1 Comparison** As shown in Table 5.2, CodeAgent successfully identified 483 potential vulnerabilities within a data set of 3,545 samples, with an impressive 449 of these finally confirmed as high-risk vulnerabilities<sup>1</sup>. CodeBERT, a key pre-trained model for code-related tasks, with its parameters frozen for this experiment, initially identified 1,063 items as vulnerable, yet only 212 passed the stringent verification criteria. Similar trends were observed with GPT-3.5 and GPT-4.0, which confirmed 317 and 345 vulnerabilities out of 864 and 671 identified items, respectively. These outcomes are further quantified by the confirmation rates ( $\text{Rate}_{cr}$ ) of 19.94% for CodeBERT, 36.69% for GPT-3.5, and 51.42% for GPT-4.0, while CodeAgent demonstrated a remarkable  $\text{Rate}_{cr}$  of 92.96%. Additionally, the analysis of confirmed vulnerabilities against all analyzed items ( $\text{Rate}_{ca}$ ) yielded 5.98%, 8.94%, 9.73%, and 12.67% for CodeBERT, GPT-3.5, GPT-4.0, and CodeAgent, respectively. Evidently, Table 5.2 not only highlights CodeAgent’s high precision in identifying vulnerable commits but also reveals the progressive improvement from GPT-3.5 to GPT-4.0, likely due to the latter’s capacity to handle longer input sequences, with token limits of 4,096 and 32,768, respectively. The integration of sophisticated algorithms like CoT and QA-Checker in CodeAgent has significantly enhanced its capabilities in vulnerability detection, surpassing the individual input-output efficiencies of GPT and CodeBERT. Appendix-Sections 5.9 and 5.18 highlight further details regarding the importance of the QA-checker. Moreover, more experimental results in 9 languages are accessible in Appendix-Section 5.15.

In addition, the analysis of vulnerabilities identified by various models reveals interesting overlaps among the models. CodeBERT confirmed 212 vulnerabilities, whereas GPT-3.5, GPT-4.0, and CodeAgent confirmed 317, 345, and 449 vulnerabilities, respectively. Notably, the intersection of vulnerabilities confirmed by CodeBERT and GPT-3.5 is 169, indicating a substantial overlap in their findings. Similarly, the intersection between CodeBERT and GPT-4.0 is 170, while a larger overlap of 212 vulnerabilities is observed between GPT-3.5 and GPT-4.0. The combined intersection among CodeBERT, GPT-3.5, and GPT-4.0 is 137, underscoring the commonalities in vulnerabilities detected across these models. Further, the intersections of vulnerabilities confirmed by CodeBERT, GPT-3.5, and GPT-4.0 with CodeAgent are 212, 317, and 334, respectively, highlighting the comprehensive coverage and detection capabilities of CodeAgent.

<sup>1</sup>The verification process involved a rigorous manual examination, extending beyond 120 working hours. Each sample being validated by at least 2 people: a researcher and an engineer



**Figure 5.4:** Overlap of vulnerability detection by CodeBERT, GPT-3.5, GPT-4.0, and CodeAgent.

**5.4.1.0.2 Ablation Study.** As shown in Table 5.2, we conducted an ablation study to evaluate the effectiveness of the QA-Checker in CodeAgent. Specifically, we created a version of our tool without the QA-Checker, referred to as  $\text{CodeAgent}_{w/o}$ . We then compared this version to the full version of CodeAgent that includes the QA-Checker. The results demonstrate that  $\text{CodeAgent}_{w/o}$  is substantially less effective in identifying vulnerable issues, yielding lower hit rates ( $\text{Rate}_{cr}$  and  $\text{Rate}_{ca}$ ). This reduction in performance highlights the critical role of the QA-Checker in enhancing CodeAgent’s overall effectiveness. More detailed information about the ablation study can be found in Appendix-Section 5.18.

## 5.4.2 Consistency and Format Detection

In this section, we will discuss the performance of CodeAgent and baselines on metrics like the F1-Score and recall score of task *CA* and *FA*. For *CA* and *FA*, the dataset we have is shown in Table 5.1 and more detailed data information is shown in Figure 5.7 in Appendix.

**5.4.2.0.1 Code Change and Commit Message Consistency Detection.** As illustrated in Table 5.3, we assess the efficacy of CodeAgent in detecting the consistency between code changes and commit messages, contrasting its performance with other prevalent methods like CodeBERT, GPT-3.5, and GPT-4.0. This evaluation specifically focuses on merged and closed commits in nine languages. In particular, CodeAgent exhibits remarkable performance, outperforming other methods in both merged and closed scenarios. In terms of Recall, CodeAgent achieved an impressive 90.11% for merged commits and 87.15% for closed ones, marking a considerable average improvement of 5.62% over the other models. Similarly, the F1-Score of CodeAgent stands at 93.89% for merged and 92.40% for closed commits, surpassing its counterparts with an average improvement of 3.79%. More comparable details in different languages are shown in Appendix-Section. 5.16.

**5.4.2.0.2 Format Consistency Detection.** In our detailed evaluation of format consistency between commits and original files, CodeAgent’s performance was benchmarked against established models like CodeBERT and GPT variants across nine different languages. This comparative analysis, presented in Table 5.4, was centered around pivotal metrics such as Recall and F1-Score. CodeAgent demonstrated a significant edge over the state-of-the-art, particularly in the merged category, with an impressive Recall of 89.34% and an F1-Score of 94.01%. These figures represent an average improvement of 10.81% in Recall and 6.94%

**Table 5.3:** Comparison of CodeAgent with other methods on merged and closed commits across 9 languages on **CA task**. ‘Imp’ represents the improvement.

Merged	CodeBERT	GPT-3.5	GPT-4.0	COT	ReAct	CodeAgent	Imp (pp)
Recall	63.64	80.08	84.27	80.73	82.04	90.11	5.84
F1	75.00	87.20	90.12	87.62	88.93	93.89	3.77
Closed	CodeBERT	GPT-3.5	GPT-4.0	COT	ReAct	CodeAgent	Imp (pp)
Recall	64.80	79.05	81.75	81.77	83.42	87.15	5.21
F1	77.20	87.35	89.61	89.30	89.81	92.40	3.35
Average	CodeBERT	GPT-3.5	GPT-4.0	COT	ReAct	CodeAgent	Imp (pp)
Recall	64.22	79.57	83.01	81.25	82.73	88.63	5.62
F1	76.01	87.28	89.61	88.46	89.37	93.16	3.79

in F1-Score over other models. In the closed category, CodeAgent continued to outperform, achieving a Recall of 89.57% and an F1-Score of 94.13%, surpassing its counterparts with an improvement of 15.56% in Recall and 9.94% in F1-Score. The overall average performance of CodeAgent further accentuates its superiority, with a Recall of 89.46% and an F1-Score of 94.07%, marking an average improvement of 13.39% in Recall and 10.45% in F1-Score. These results underscore CodeAgent’s exceptional capability in accurately detecting format consistency between commits and their original files.

**Table 5.4:** Comparison of CodeAgent with other methods on merged and closed commits across the 9 languages on **FA task**. ‘Imp’ represents the improvement.

Merged	CodeBERT	GPT-3.5	GPT-4.0	COT	ReAct	CodeAgent	Imp (pp)
Recall	60.59	60.72	78.53	70.39	71.21	89.34	10.81
F1	74.14	74.88	87.07	80.69	82.18	94.01	6.94
Closed	CodeBERT	GPT-3.5	GPT-4.0	COT	ReAct	CodeAgent	Imp (pp)
Recall	69.95	73.61	68.46	73.39	74.01	89.57	15.56
F1	80.49	84.19	80.16	83.65	83.90	94.13	9.94
Average	CodeBERT	GPT-3.5	GPT-4.0	COT	ReAct	CodeAgent	Imp (pp)
Recall	65.27	67.17	73.50	71.89	72.61	89.46	15.96
F1	77.32	79.54	83.62	82.17	83.04	94.07	10.45

### 5.4.3 Code Revision

We evaluate the effectiveness of CodeAgent in revision suggestion (i.e., bug fixing) based on Edit Progress (EP) metric. We consider Trans-Review, AutoTransform, T5-Review, CodeBERT, GraphCodeBERT, CodeT5 as comparable state of the art. As detailed in Table 5.5, these approaches exhibit a varied performance across different datasets. In particular, CodeAgent shows remarkable performance in the T5-Review dataset, achieving the highest EP of 37.6%. This is a significant improvement over other methods, which underlines the effectiveness of CodeAgent in handling complex code revision tasks. Furthermore, with an average EP of 31.6%, CodeAgent consistently outperforms its counterparts, positioning itself as a leading solution in automated code revision. Its ability to excel in the T5-Review, a challenging benchmark data, indicates a strong capability to address complex bugs. In addition, its overall average performance surpasses other state-of-the-art models, highlighting its robustness and reliability.

**Table 5.5:** Experimental Results for the Code Revision (**CR task**) of CodeAgent and the state-of-the-art works. Bold indicates the best performers.

Approach	Trans-Review <sub>data</sub>	AutoTransform <sub>data</sub>	T5-Review <sub>data</sub>	Average
	EP	EP	EP	EP
<b>Trans-Review</b>	-1.1%	-16.6%	-151.2%	-56.3%
<b>AutoTransform</b>	49.7%	<b>29.9%</b>	9.7%	29.8%
<b>T5-Review</b>	-14.9%	-71.5%	13.8%	-24.2%
<b>CodeBERT</b>	49.8%	-75.3%	22.3%	-1.1%
<b>GraphCodeBERT</b>	<b>50.6%</b>	-80.9%	22.6%	-2.6%
<b>CodeT5</b>	41.8%	-67.8%	25.6%	-0.1%
<b>CodeAgent</b>	42.7%	14.4%	<b>37.6%</b>	<b>31.6%</b>

## 5.5 Conclusion

In this paper, we introduced CodeAgent, a novel multi-agent framework that automates code reviews. CodeAgent leverages its novel QA-Checker system to maintain focus on the review’s objectives and ensure alignment. Our experiments demonstrate CodeAgent’s effectiveness in detecting vulnerabilities, enforcing code-message consistency, and promoting uniform code style. Furthermore, CodeAgent outperforms existing state-of-the-art solutions in code revision suggestions. By incorporating human-like conversational elements and considering the specific characteristics of code review, CodeAgent significantly improves both efficiency and accuracy. We believe this work opens exciting new avenues for research and collaboration practices in software development.

## 5.6 Details of QA-Checker Algorithm

**Lemma 1.** Let  $\mathcal{Q}(Q_i, A_i)$  denote the quality assessment function of the QA-Checker for the question-answer pair  $(Q_i, A_i)$  in a conversation at the  $i$ -th iteration. Assume  $\mathcal{Q}$  is twice differentiable and its Hessian matrix  $H(\mathcal{Q})$  is positive definite. If the QA-Checker modifies the question  $Q_i$  to  $Q_{i+1}$  by attaching an additional instruction  $aai_i$ , and this leads to a refined answer  $A_{i+1}$ , then the sequence  $\{(Q_i, A_i)\}$  converges to an optimal question-answer pair  $(Q^*, A^*)$ , under specific regularity conditions.

*Proof.* The QA-Checker refines the question and answers using the rule:

$$\begin{aligned} Q_{i+1} &= Q_i + aai_i, \\ A_{i+1} &= A_i - \alpha H(\mathcal{Q}(Q_i, A_i))^{-1} \nabla \mathcal{Q}(Q_i, A_i), \end{aligned}$$

where  $\alpha$  is the learning rate. To analyze convergence, we consider the Taylor expansion of  $\mathcal{Q}$  around  $(Q_i, A_i)$ :

$$\begin{aligned} \mathcal{Q}(Q_{i+1}, A_{i+1}) &\approx \mathcal{Q}(Q_i, A_i) + \nabla \mathcal{Q}(Q_i, A_i) \\ &\quad \cdot (Q_{i+1} - Q_i, A_{i+1} - A_i) \\ &\quad + \frac{1}{2} (Q_{i+1} - Q_i, A_{i+1} - A_i)^T \\ &\quad H(\mathcal{Q}(Q_i, A_i)) (Q_{i+1} - Q_i, A_{i+1} - A_i). \end{aligned}$$

Substituting the update rule and rearranging, we get:

$$\begin{aligned} \mathcal{Q}(Q_{i+1}, A_{i+1}) &\approx \mathcal{Q}(Q_i, A_i) \\ &\quad - \alpha \nabla \mathcal{Q}(Q_i, A_i)^T H(\mathcal{Q}(Q_i, A_i))^{-1} \\ &\quad \nabla \mathcal{Q}(Q_i, A_i) \\ &\quad + \frac{\alpha^2}{2} \nabla \mathcal{Q}(Q_i, A_i)^T H(\mathcal{Q}(Q_i, A_i))^{-1} \\ &\quad \nabla \mathcal{Q}(Q_i, A_i). \end{aligned}$$

For sufficiently small  $\alpha$ , this model suggests an increase in  $\mathcal{Q}$ , implying convergence to an optimal question-answer pair  $(Q^*, A^*)$  as  $i \rightarrow \infty$ . The convergence relies on the positive definiteness of  $H(\mathcal{Q})$  and the appropriate choice of  $\alpha$ , ensuring each iteration moves towards an improved quality of the question-answer pair.  $\square$

In practical terms, this lemma and its proof underpin the QA-Checker's ability to refine answers iteratively. The QA-Checker assesses the quality of each answer concerning the posed question, employing advanced optimization techniques that are modeled by the modified Newton-Raphson method to enhance answer quality. This framework ensures that, with each iteration, the system moves closer to the optimal answer, leveraging both first and second-order derivatives for efficient and effective learning.

**5.6.0.0.1 Further Discussion** The QA-Checker computes  $\mathcal{Q}(Q_i, A_i)$  at each iteration  $i$  and compares it to a predefined quality threshold  $\tau$ . If  $\mathcal{Q}(Q_i, A_i) < \tau$ , the QA-Checker generates an additional instruction  $aai_i$  to refine the question to  $Q_{i+1} = Q_i + aai_i$ , prompting the agents to generate an improved answer  $A_{i+1}$ .

First, we assume that the quality assessment function  $\mathcal{Q}(Q_i, A_i)$  is twice differentiable with respect to the question  $Q_i$ . This assumption is reasonable given the smooth nature of the component functions (relevance, specificity, and coherence) and the use of continuous

word embeddings. Next, we apply the second-order Taylor approximation to  $\mathcal{Q}(Q_{i+1}, A_{i+1})$  around the point  $(Q_i, A_i)$ :

$$\begin{aligned}\mathcal{Q}(Q_{i+1}, A_{i+1}) &\approx \mathcal{Q}(Q_i, A_i) + \nabla \mathcal{Q}(Q_i, A_i)^T \Delta Q_i \\ &\quad + \frac{1}{2} \Delta Q_i^T H(\mathcal{Q}(Q_i, A_i)) \Delta Q_i + R_2(\Delta Q_i)\end{aligned}$$

where  $\Delta Q_i = Q_{i+1} - Q_i$ ,  $H(\mathcal{Q}(Q_i, A_i))$  is the Hessian matrix of  $\mathcal{Q}$  evaluated at  $(Q_i, A_i)$ , and  $R_2(\Delta Q_i)$  is the remainder term.

Assuming that the remainder term  $R_2(\Delta Q_i)$  is negligible and that the Hessian matrix is positive definite, we can approximate the optimal step  $\Delta Q_i^*$  as:

$$\Delta Q_i^* \approx -H(\mathcal{Q}(Q_i, A_i))^{-1} \nabla \mathcal{Q}(Q_i, A_i).$$

Substituting this approximation into the Taylor expansion and using the fact that  $Q_{i+1} = Q_i + \alpha \Delta Q_i^*$  (where  $\alpha$  is the learning rate), we obtain:

$$\begin{aligned}\mathcal{Q}(Q_{i+1}, A_{i+1}) &\approx \mathcal{Q}(Q_i, A_i) - \alpha \nabla \mathcal{Q}(Q_i, A_i)^T \\ &\quad \cdot H(\mathcal{Q}(Q_i, A_i))^{-1} \nabla \mathcal{Q}(Q_i, A_i) \\ &\quad + \frac{\alpha^2}{2} \nabla \mathcal{Q}(Q_i, A_i)^T H(\mathcal{Q}(Q_i, A_i))^{-1} \\ &\quad \cdot \nabla \mathcal{Q}(Q_i, A_i).\end{aligned}$$

The assumptions of twice differentiability, negligible remainder term, and positive definite Hessian matrix provide a more solid foundation for the approximation in Lemma 3.1. For sufficiently small  $\alpha$ , this approximation suggests an increase in  $\mathcal{Q}$ , implying convergence to an optimal question-answer pair  $(Q^*, A^*)$  as  $i \rightarrow \infty$ . The convergence relies on the positive definiteness of  $H(\mathcal{Q})$  and the appropriate choice of  $\alpha$ , ensuring each iteration moves towards an improved quality of the question-answer pair.

The quality assessment function  $\mathcal{Q}$  used by the QA-Checker is defined as:

$$\begin{aligned}\mathcal{Q}(Q_i, A_i) &= \alpha \cdot \text{Relevance}(Q_i, A_i) \\ &\quad + \beta \cdot \text{Specificity}(A_i) \\ &\quad + \gamma \cdot \text{Coherence}(A_i)\end{aligned}$$

where:

- $Q_i$  and  $A_i$  represent the question and answer at the  $i$ -th iteration of the conversation.
- $\text{Relevance}(Q_i, A_i)$  measures how well the answer  $A_i$  addresses the key points and intent of the question  $Q_i$ , computed as:

$$\text{Relevance}(Q_i, A_i) = \frac{\vec{Q}_i \cdot \vec{A}_i}{|\vec{Q}_i| |\vec{A}_i|}$$

where  $\vec{Q}_i$  and  $\vec{A}_i$  are vector representations of  $Q_i$  and  $A_i$ .

- $\text{Specificity}(A_i)$  assesses how specific and detailed the answer  $A_i$  is, calculated as:

$$A_i = \frac{\sum_{t \in \text{ContentWords}(A_i)} \text{TechnicalityScore}(t)}{\text{Length}(A_i)}$$

where  $\text{ContentWords}(A_i)$  is the set of substantive content words in  $A_i$ ,  $\text{TechnicalityScore}(t)$  is a measure of how technical or domain-specific the term  $t$  is, and  $\text{Length}(A_i)$  is the total number of words in  $A_i$ .

- $\text{Coherence}(A_i)$  evaluates the logical flow and structural coherence of the answer  $A_i$ , computed as:

$$\begin{aligned}\text{Coherence}(A_i) = & \alpha \cdot \text{DiscourseConnectives}(A_i) \\ & + \beta \cdot \text{CoreferenceConsistency}(A_i) \\ & + \gamma \cdot \text{AnswerPatternAdherence}(A_i)\end{aligned}$$

where  $\text{Discourse Connectives}(A_i)$  is the density of discourse connectives in  $A_i$ ,  $\text{CoreferenceConsistency}(A_i)$  measures the consistency of coreference chains in  $A_i$ , and  $\text{AnswerPatternAdherence}(A_i)$  assesses how well  $A_i$  follows the expected structural patterns for the given question type.

$\alpha$ ,  $\beta$ , and  $\gamma$  are non-negative weights that sum to 1, with  $\alpha = \beta = \gamma$ .

## 5.7 Complete Related Work

**Automating Code Review Activities** Our focus included detecting source code vulnerabilities, ensuring style alignment, and maintaining commit message and code consistency. Other studies explore various aspects of code review. Hellendoorn et al. Hellendoorn et al. [2021] addressed the challenge of anticipating code change positions. Siow et al. Siow et al. [2020] introduced CORE, employing multi-level embeddings for code modification semantics and retrieval-based review suggestions. Hong et al. Hong et al. [2022] proposed COMMENTFINDER, a retrieval-based method for suggesting comments during code reviews. Tufano et al. Tufano et al. [2021] designed T5CR with SentencePiece, enabling work with raw source code without abstraction. Li et al. Li et al. [2022b] developed CodeReviewer, focusing on code diff quality, review comment generation, and code refinement using the T5 model. Recently, large language models have been incorporated; Lu et al. Lu et al. [2023] fine-tuned LLama with prefix tuning for LLaMA-Reviewer, using parameter-efficient fine-tuning and instruction tuning in a code-centric domain.

**Collaborative AI** Collaborative AI refers to artificial intelligent systems designed to achieve shared goals with humans or other AI systems. Previous research extensively explores the use of multiple LLMs in collaborative settings, as demonstrated by Talebirad et al. Talebirad and Nadiri [2023] and Qian et al. Qian et al. [2023]. These approaches rely on the idea that inter-agent interactions enable LLMs to collectively enhance their capabilities, leading to improved overall performance. The research covers various aspects of multi-agent scenarios, including collective thinking, conversation dataset curation, sociological phenomenon exploration, and collaboration for efficiency. Collective thinking aims to boost problem-solving abilities by orchestrating discussions among multiple agents. Researchers like Wei et al. Wei et al. [2023a] and Li et al. Li et al. [2023a] have created conversational datasets through role-playing methodologies. Sociological phenomenon investigations, such as Park et al. Park et al. [2023]’s work, involve creating virtual communities with rudimentary language interactions and limited cooperative endeavors. In contrast, Akata et al. Akata et al. [2023] scrutinized LLM cooperation through orchestrated repeated games. Collaboration for efficiency, proposed by Cai et al. Cai et al. [2023], introduces a model for cost reduction through large models as tool-makers and small models as tool-users. Zhang et al. Zhang et al. [2023b] established a framework for verbal communication and collaboration, enhancing overall efficiency. However, Li et al. Li et al. [2023a] and Qian et al. Qian et al. [2023], presenting a multi-agent framework for software development, primarily relied on natural language conversations, not standardized software engineering documentation, and lacked advanced human process management expertise. Challenges in multi-agent cooperation include maintaining coherence, avoiding unproductive loops,

and fostering beneficial interactions. Our approach emphasizes integrating advanced human processes, like code review in software maintenance, within multi-agent systems.

## 5.8 Experimental Details

In our work, the maximum number of conversation rounds is set at 10.

### 5.8.1 Role Definition

Six roles are defined as shown in Figure 5.5.

Role Specialization	
	My primary responsibilities involve the integration of commit content, crafting commit messages, managing original files, and supplying necessary input information like commit details and code.
User	
	I'm Chief Executive Officer. Now, we are both working at CodeAgent and we share a common interest in collaborating to successfully complete the code review for commits or code. My main responsibilities include being a decision-maker in policy and strategy, a leader managing teams, and an effective communicator with management and employees. I also specialize in summarizing complex code reviews.
CEO	
	I am the Chief Product Officer at CodeAgent, collaborating closely with my team to complete code reviews successfully. I am responsible for assisting CEO and coder to summary code review reports
CPO	
	I am the CTO of CodeAgent, familiar with various programming languages and skilled in overarching technology strategies. My role involves collaborating on new customer tasks, making high-level IT decisions that align with our organization's goals, and working closely with IT staff in everyday operations.
CTO	
	I am a Code reviewer at CodeAgent collaborating to ensure software quality by assessing code for defects, vulnerabilities, and consistency issues, fixing bugs, and suggesting improvements. I also collaborate with other agents to complete the code revision and summary of code review
Reviewer	
	I am a Coder at CodeAgent who actively reviews and revises code. I make decisions about code changes and ensure code quality by evaluating code for defects and suggesting improvements. I am proficient in various programming languages and platforms, including Python, Java, Go, C++, JavaScript, C, C#, PHP, and Ruby, etc.
Coder	

**Figure 5.5:** Specialization of six main characters in CodeAgent.

Apart from that, for the QA-checker in CodeAgent, we define an initial prompt for it, which is shown as follows:



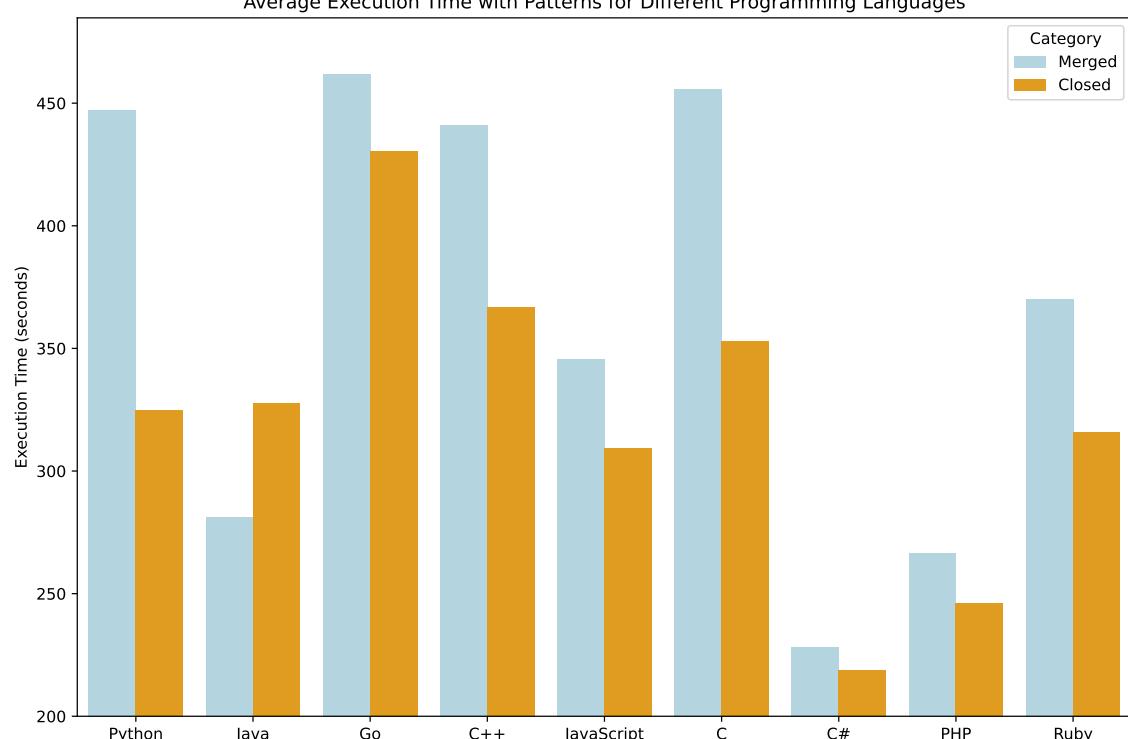
I'm the QA-Checker, an AI-driven agent specializing in ensuring quality and coherence in conversational dynamics, particularly in code review discussions at CodeAgent. My primary role involves analyzing and aligning conversations to maintain topic relevance, ensuring that all discussions about code commits and reviews stay focused and on track. As a sophisticated component of the AI system, advanced algorithms are applied, including chain-of-thought reasoning and optimization techniques, to evaluate and guide conversational flow. I am adept at identifying and correcting topic drifts and ensuring that every conversation adheres to its intended purpose. My capabilities extend to facilitating clear and effective communication between team members, making me an essential asset in streamlining code review processes and enhancing overall team collaboration and decision making.

### 5.8.2 Execute Time Across Languages

As depicted in the data, we observe a significant trend in the average execution time for code reviews in CodeAgent across various programming languages. The analysis includes nine languages: Python, Java, Go, C++, JavaScript, C, C#, PHP, and Ruby. For each language, the average execution time of code reviews for both merged and closed pull requests (PRs) is measured. The results, presented in Figure 5.6, indicate that, on average, the execution time for merged PRs is longer than that for closed PRs by approximately 44.92 seconds. This considerable time difference can be attributed to several potential reasons. One primary explanation is that merged PRs likely undergo a more rigorous and detailed review process. They are intended to be integrated into the main codebase, and as such, contributors might be requested to update their commits in the PRs more frequently to adhere to the project's high-quality standards. On the other hand, closed PRs, which are not meant for merging, might not require such extensive review processes, leading to shorter review times on average, which may also be the reason they are not merged into main projects.

**Table 5.6:** Comparative Overview of QA-Checker AI System and Recursive Self-Improvement Systems

Feature/System	QA-Checker AI System	Recursive Self-Improvement System
Application Focus	Specialized for QA tasks with precise task execution	Broad scope, covering various dimensions like software development and learning algorithms
Learning Mechanism	Advanced optimization techniques for iterative improvement in QA	Multi-level learning: learning, meta-learning, and recursive self-improvement
Scope of Improvement	Focused on individual capability in specific QA tasks	Enhances the entire system, including multi-agent interactions and communication protocols
Experience Integration	Based on mathematical models to optimize answer quality	Utilizes experiences from past projects to improve overall performance


**Figure 5.6:** Execution time with CodeAgent across different language (count unit: second).

## 5.9 Comparative Analysis of QA-Checker AI System and Recursive Self-Improvement Systems

In this section, we will delve into the differences between QA-Checker and self-improvement systems Hong et al. [2023], and underscore the importance of the QA-Checker in role conversations.

### 5.9.1 Comparison Table

We begin with a comparative overview presented in Table 5.6.

### 5.9.2 Differences and Implications

The key differences between these systems lie in their application scope, learning mechanisms, and improvement scopes. The QA-Checker is highly specialized, focusing on QA tasks with efficiency and precision. In contrast, recursive self-improvement systems boast a broader application range and adaptability, integrating experiences from diverse projects for systemic improvements.

### 5.9.3 Importance of QA-Checker in Role Conversations

In the context of role conversations, the QA-Checker plays a pivotal role. Its specialized nature makes it exceptionally adept at handling specific conversational aspects, such as accuracy, relevance, and clarity in responses. This specialization is crucial in domains where the quality of information is paramount, ensuring that responses are not only correct but also contextually appropriate and informative.

Furthermore, the efficiency of the QA-Checker in refining responses based on advanced optimization techniques makes it an invaluable tool in dynamic conversational environments. It can quickly adapt to the nuances of a conversation, providing high-quality responses that are aligned with the evolving nature of dialogue.

### 5.9.4 Conclusion

While recursive self-improvement systems offer broad adaptability and systemic learning, the QA-Checker stands out in its specialized role in QA tasks, particularly in role conversations. Its focused approach to improving answer quality and its efficiency in handling conversational nuances make it an essential component in AI-driven communication systems.

## 5.10 Capabilities Analysis between CodeAgent and Other Methods

Compared to open-source baseline methods such as AutoGPT and autonomous agents such as ChatDev and MetaGPT, CodeAgent offers functions for code review tasks: consistency analysis, vulnerability analysis, and format analysis. As shown in Table 5.7, our CodeAgent encompasses a wide range of abilities to handle complex code review tasks efficiently. Incorporating the QA-Checker self-improved module can significantly improve the conversation generation between agents and contribute to the improvement of code review. Compared to COT, the difference and the advantages of CodeAgent with QA-Checker are shown in Section 5.9.

**Table 5.7:** Comparison of capabilities for CodeAgent and other approaches. ‘✓’ indicates the presence of a specific feature in the corresponding framework, ‘✗’ is absence. ChatDev and MetaGPT are two representative multi-agent frameworks, GPT is a kind of single-agent framework, and CodeBert is a representative pre-trained model.

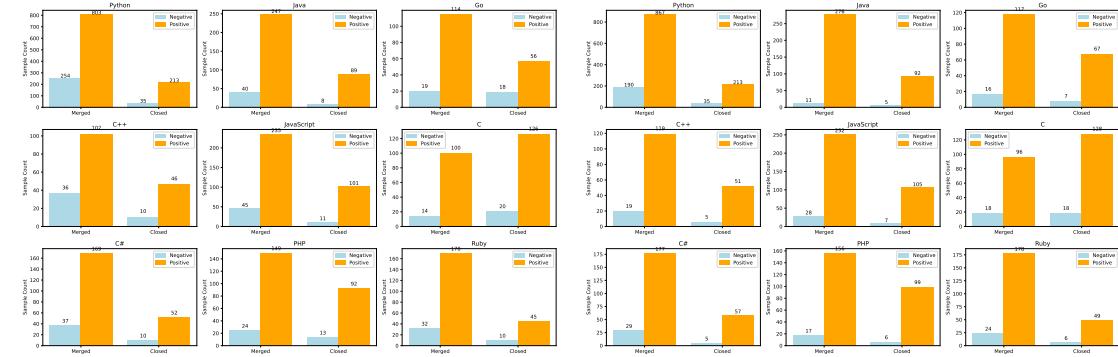
Approaches	Consistency Analysis	Vulnerability Analysis	Format Analysis	Code Revision	COT	QA-Checker
ChatDev Qian et al. [2023]	✗	✗	✗	✗	✓	✗
MetaGPT Hong et al. [2023]	✗	✗	✗	✗	✓	✗
GPT OpenAI [2022]	✓	✓	✓	✓	✗	✗
CodeBert Feng et al. [2020a]	✓	✓	✓	✓	✗	✗
CodeAgent	✓	✓	✓	✓	✓	✓

## 5.11 Dataset

**5.11.0.0.1 Previous Dataset** As shown in Zhou et al. [2023b], our study incorporates three distinct datasets for evaluating the performance of CodeAgent: Trans-Review<sub>data</sub>, AutoTransform<sub>data</sub>, and T5-Review<sub>data</sub>. Trans-Review<sub>data</sub>, compiled by Tufano et al. Tufano et al. [2021], derives from Gerrit and GitHub projects, excluding noisy or overly lengthy comments and review data with new tokens in revised code not present in the initial submission. AutoTransform<sub>data</sub>, collected by Thongtanunam et al. Thongtanunam et al. [2022] from three Gerrit repositories, comprises only submitted and revised codes without review comments. Lastly, T5-Review<sub>data</sub>, gathered by Tufano et al. Tufano et al. [2022] from Java

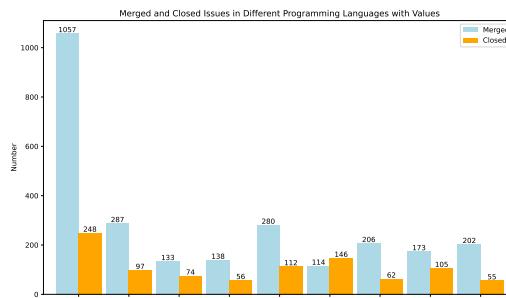
projects on GitHub, filters out noisy, non-English, and duplicate comments. These datasets are employed for Code Revision Before Review (CRB) and Code Revision After Review (CRA) tasks, with the exception of AutoTransform<sub>data</sub> for CRA and Review Comment Generation (RCG) due to its lack of review comments.

**5.11.0.0.2 New Dataset Design and Collection** To enhance our model evaluation and avoid data leakage, we curated a new dataset, exclusively collecting data from repositories created after April 2023. This approach ensures the evaluation of our CodeAgent model on contemporary and relevant data, free from historical biases. The new dataset is extensive, covering a broad spectrum of software projects across nine programming languages.



(a) Positive and negative data of both merged and closed commits across 9 languages on CA task (Sec 5.2.1). (b) Positive and negative data of both merged and closed commits across 9 languages on FA task (Sec 5.2.1).

**Figure 5.7:** Distribution of positive, negative of both merged and closed data across 9 languages, including ‘python’, ‘java’, ‘go’, ‘c++’, ‘javascript’, ‘c’, ‘c#’, ‘php’, ‘ruby’.



**Figure 5.8:** Comparative Visualization of Merged and Closed Commit Counts Across Various Programming Languages

**5.11.0.0.3 Dataset Description** Our dataset, illustrated in Fig. 5.8, encapsulates a detailed analysis of consistency and format detection in software development, spanning various programming languages. It includes CA (consistency between commit and commit message (See Sec 5.2.1)) and FA (format consistency between commit and original (See Sec 5.2.1)) data, segmented into positive and negative samples based on the merged and closed status of pull requests. For example, in Python, the dataset comprises 254 merged and 35 closed negative CA samples, alongside 803 merged and 213 closed positive CA samples, with corresponding distributions for other languages like Java, Go, C++, and more. Similarly, the FA data follows this pattern of positive and negative samples across languages. Figure 5.7 graphically represents this data, highlighting the distribution and comparison of merged versus closed samples in both CA and FA categories for each language. This comprehensive dataset, covering over 3,545 commits and nearly 2,933 pull requests from more than 180 projects, was meticulously compiled using a custom crawler designed for

GitHub API interactions, targeting post-April 2023 repositories to ensure up-to-date and diverse data for an in-depth analysis of current software development trends.

**Table 5.8:** Statistics of Studied Datasets.

Dataset Statistics	#Train	#Valid	#Test
<b>Trans-Review</b>	13,756	1,719	1,719
<b>AutoTransform</b>	118,039	14,750	14,750
<b>T5-Review</b>	134,239	16,780	16,780

## 5.12 Key Factors Leading to Vulnerabilities

The following table outlines various key factors that can lead to vulnerabilities in software systems, along with their descriptions. These factors should be carefully considered and addressed to enhance the security of the system.

## 5.13 Data Leakage Statement

As the new dataset introduced in Section 5.11, the time of the collected dataset is after April 2023, avoiding data leakage while we evaluate CodeAgent on **codeData** dataset.

## 5.14 Algorithmic Description of CodeAgent Pipeline with QA-Checker

This algorithm demonstrates the integration of QA-Checker within the CodeAgent pipeline, employing mathematical equations to describe the QA-Checker's iterative refinement process.

No.	Vulnerability Factor	Description
1	Insufficient Input Validation	Check for vulnerabilities like SQL injection, Cross-Site Scripting (XSS), and command injection in new or modified code, especially where user input is processed.
2	Buffer Overflows	Particularly in lower-level languages, ensure that memory management is handled securely to prevent overflows.
3	Authentication and Authorization Flaws	Evaluate any changes in authentication and authorization logic for potential weaknesses that could allow unauthorized access or privilege escalation.
4	Sensitive Data Exposure	Assess handling and storage of sensitive information like passwords, private keys, or personal data to prevent exposure.
5	Improper Error and Exception Handling	Ensure that errors and exceptions are handled appropriately without revealing sensitive information or causing service disruption.
6	Vulnerabilities in Dependency Libraries or Components	Review updates or changes in third-party libraries or components for known vulnerabilities.
7	Cross-Site Request Forgery (CSRF)	Verify that adequate protection mechanisms are in place against CSRF attacks.
8	Unsafe Use of APIs	Check for the use of insecure encryption algorithms or other risky API practices.
9	Code Injection	Look for vulnerabilities related to dynamic code execution.
10	Configuration Errors	Ensure that no insecure configurations or settings like open debug ports or default passwords have been introduced.
11	Race Conditions	Analyze for potential data corruption or security issues arising from race conditions.
12	Memory Leaks	Identify any changes that could potentially lead to memory leaks and resource exhaustion.
13	Improper Resource Management	Check resource management, such as proper closure of file handles or database connections.
14	Inadequate Security Configurations	Assess for any insecure default settings or unencrypted communications.
15	Path Traversal and File Inclusion Vulnerabilities	Examine for risks that could allow unauthorized file access or execution.
16	Unsafe Deserialization	Look for issues that could allow the execution of malicious code or tampering with application logic.
17	XML External Entity (XXE) Attacks	Check if XML processing is secure against XXE attacks.
18	Inconsistent Error Handling	Review error messages to ensure they do not leak sensitive system details.
19	Server-Side Request Forgery (SSRF)	Analyze for vulnerabilities that could be exploited to attack internal systems.
20	Unsafe Redirects and Forwards	Check for vulnerabilities leading to phishing or redirection attacks.
21	Use of Deprecated or Unsafe Functions and Commands	Identify usage of any such functions and commands in the code.
22	Code Leakages and Hardcoded Sensitive Information	Look for hardcoded passwords, keys, or other sensitive data in the code.
23	Unencrypted Communications	Verify that data transmissions are securely encrypted to prevent interception and tampering.
24	Mobile Code Security Issues	For mobile applications, ensure proper handling of permission requests and secure data storage.
25	Cloud Service Configuration Errors	Review any cloud-based configurations for potential data leaks or unauthorized access.

---

**Algorithm 1:** Integrated Workflow of CodeAgent with QA-Checker

---

**Input:** Code submission, commit message, original files

**Output:** Refined code review document

Initialize phase  $p=1$  **while**  $p \leq 4$  **do**

**Switch:** Phase  $p$

**Case 1: Basic Info Sync**

            Conduct initial information analysis

            Update:  $p=2$

**Case 2: Code Review**

            Perform code review with Coder and Reviewer

            Update:  $p=3$

**Case 3: Code Alignment**

            Apply code revisions based on feedback

            Update:  $p=4$

**Case 4: Document**

            Finalize review document

            Update:  $p=5$  (End)

**QA-Checker Refinement** (Applies in Cases 2 and 3)

        Let  $Q_i$  be the current question and  $A_i$  the current answer

        Evaluate response quality:  $qScore = Q(Q_i, A_i)$  **if**  $qScore$  below threshold **then**

            Generate additional instruction  $aai$

            Update question:  $Q_{i+1} = Q_i + aai$

            Request new response:  $A_{i+1}$

**Return:** Refined code review document

---

In this algorithm,  $Q(Q_i, A_i)$  represents the quality assessment function of the QA-Checker, which evaluates the relevance and accuracy of the answer  $A_i$  to the question  $Q_i$ . If the quality score  $qScore$  is below a predefined threshold, the QA-Checker intervenes by generating an additional instruction  $aai$  to refine the question, prompting a more accurate response in the next iteration.

## 5.15 Detailed Performance of CodeAgent in Various Languages on VA task

In our comprehensive analysis using CodeAgent, as detailed in Table 5.9, we observe a diverse landscape of confirmed vulnerabilities across different programming languages. The table categorizes these vulnerabilities into ‘merged’ and ‘closed’ statuses for languages such as Python, Java, Go, C++, JavaScript, C, C#, PHP, and Ruby. A significant finding is a markedly high number of ‘merged’ vulnerabilities in Python, potentially reflective of its extensive application or intrinsic complexities leading to security gaps. Conversely, languages like Go, Ruby, and C exhibit notably lower counts in both categories, perhaps indicating lesser engagement in complex applications or more robust security protocols. Table 5.9 shows that the ‘closed’ category consistently presents lower vulnerabilities than ‘merged’ across most languages, signifying effective resolution mechanisms. However, an exception is noted in C, where ‘closed’ counts surpass those of ‘merged’, possibly indicating either delayed vulnerability identification or efficient mitigation strategies. Remarkably, the Rate<sub>close</sub> is generally observed to be higher than Rate<sub>merge</sub> across the languages, exemplifying a significant reduction in vulnerabilities post-resolution. For example, Python demonstrates a Rate<sub>merge</sub> of 14.00% against a

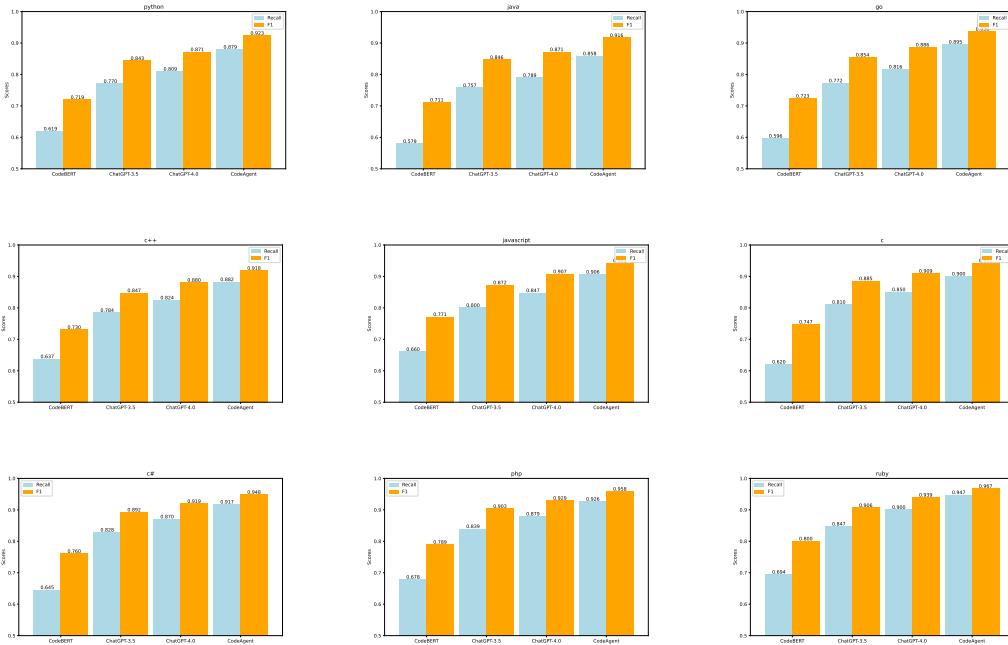
higher Rate<sub>close</sub> of 18.16%. This trend is consistent in most languages, emphasizing the importance of proactive vulnerability management. The Rate<sub>avg</sub>, representing the proportion of confirmed vulnerabilities against the total of both merged and closed items, further elucidates this point, with C++ showing the highest Rate<sub>avg</sub> at 16.49%. These insights not only underline the diverse vulnerability landscape across programming languages but also highlight the adeptness of CodeAgent in pinpointing and verifying vulnerabilities in these varied contexts.

**Table 5.9:** Vulnerable problems (#) found by CodeAgent. Rate<sub>merge</sub> means the value of confirmed divided by the total number in the merged and Rate<sub>close</sub> is the value of confirmed divided by the total number in the closed. Rate<sub>avg</sub> is the value of the confirmed number divided by the total number of the merged and closed.

CodeAgent	Python	Java	Go	C++	JavaScript	C	C#	PHP	Ruby
merged (total#)	1,057	287	133	138	280	114	206	173	202
merged (confirmed#)	148	17	11	19	34	9	21	28	20
Rate <sub>merge</sub>	14.00%	5.92%	8.27%	13.77%	12.14%	7.89%	10.19%	16.18%	9.90%
closed (total#)	248	97	74	56	112	146	62	105	55
closed (confirmed#)	45	10	5	13	16	26	7	15	5
Rate <sub>close</sub>	18.16%	10.31%	6.76%	23.2%	14.29%	17.81%	11.29%	14.29%	9.09%
Total number (#)	1,305	384	207	194	392	260	268	278	257
Total confirmed (#)	193	27	16	32	50	35	28	43	25
Rate <sub>avg</sub>	14.79%	7.03%	7.73%	16.49%	12.76%	13.46%	10.45%	14.47%	9.73%

## 5.16 More detailed experimental results on CA and FA tasks

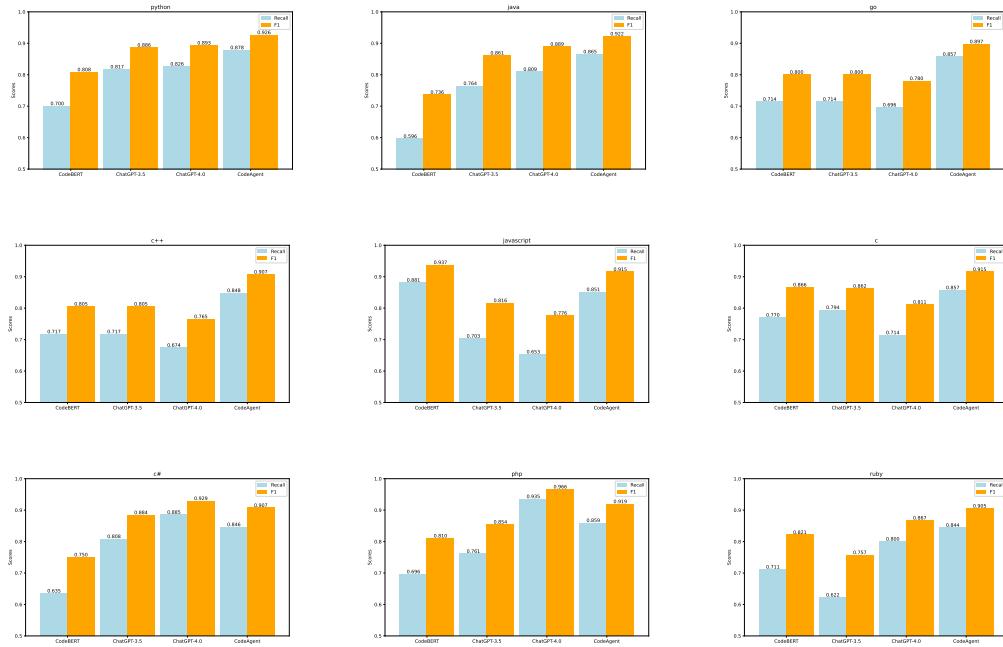
Detailed experimental results of CA are shown in Figure 5.9 and Figure 5.10. Detailed experimental results of FA are shown in Figure 5.11 and Figure 5.12.



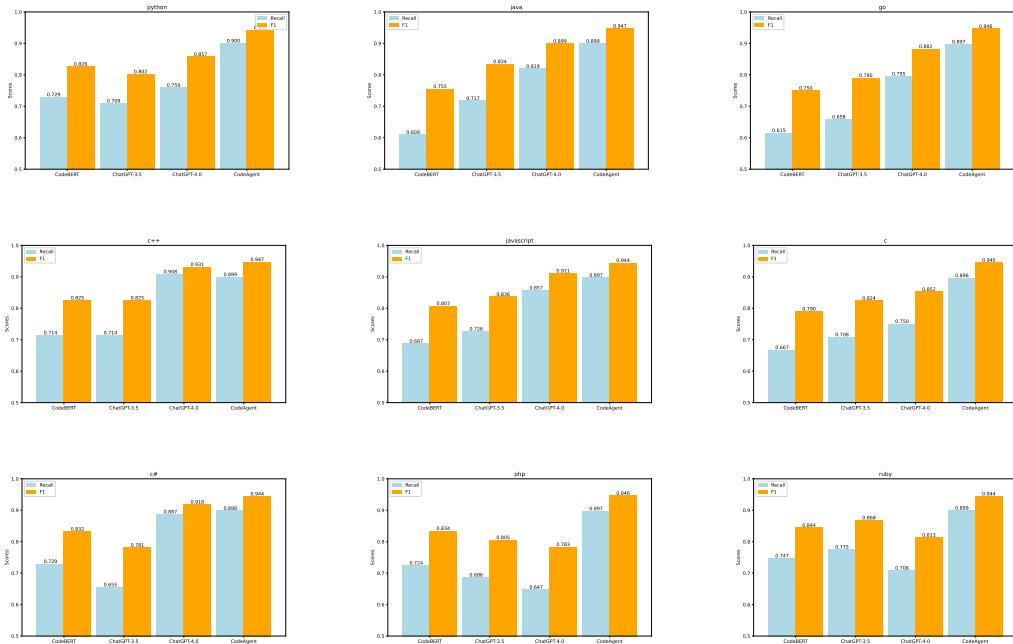
**Figure 5.9:** Comparison of models on the merged data across 9 languages on CA task.

## 5.17 Case Study

As shown in Table 5.10, we can easily localize the figure numbers of case studies for specific programming languages.



**Figure 5.10:** Comparison of models on the **closed** data across 9 languages on **CA** task.



**Figure 5.11:** Comparison of models on the **merged** data across 9 languages on **FA** task.

### 5.17.1 Performance on 9 languages

**Table 5.10:** Correlation Table between specific programming language and case study.

Programming Language	Figure No.
Python	5.13
Java	5.14
Go	5.15
C++	5.16
JavaScript	5.17
C	5.18
C#	5.19
php	5.20
Ruby	5.21

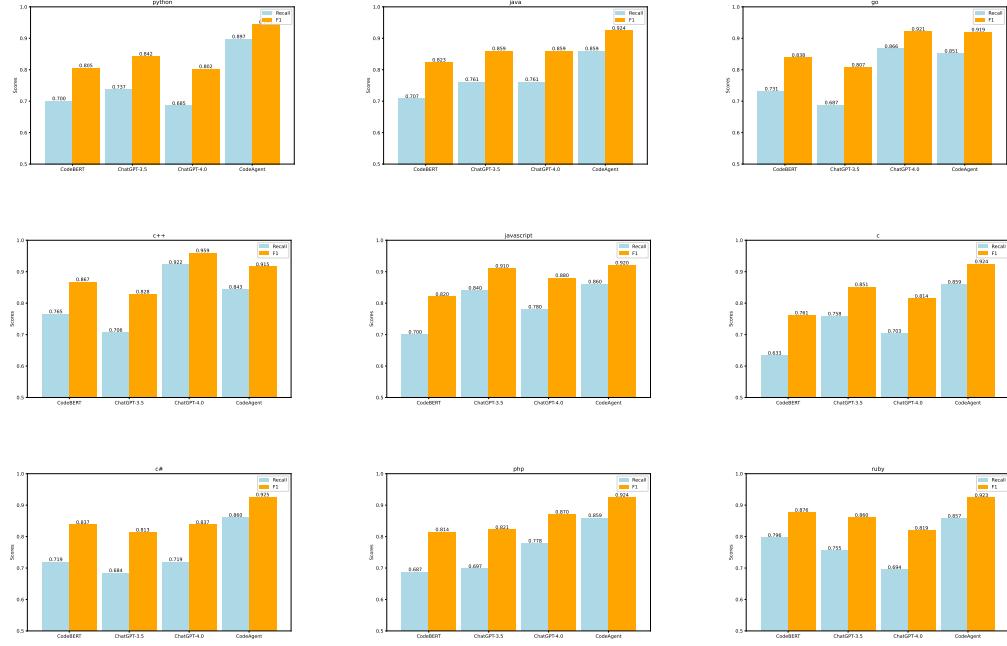


Figure 5.12: Comparison of models on the **closed** data across 9 languages on **FA** task.

### 5.17.2 Difference of CodeAgent-3.5 and CodeAgent-4.0

CodeAgent-3.5 and CodeAgent-4.0 in this paper has no difference in general code review, however, CodeAgent-4.0 is more powerful in processing long input sequences and logic reasoning. As shown in Figure 5.22, we take one example of consistency detection between commit and commit message and find that CodeAgent-4.0 differs from CodeAgent-3.5 in the detailed explanation. CodeAgent-3.5 outputs a report with 15k lines while CodeAgent-4.0 outputs a report with more than 17.7k lines. Detailed data is shown in <https://zenodo.org/records/10607925>.

## 5.18 Ablation study

In this section, we evaluate the performance of different parts in CodeAgent in vulnerability analysis. CodeAgent is based on chain-of-thought (COT) and large language model (a.k.a. GPT). As shown in Section 5.4.1, CodeAgent outperforms baselines (a.k.a. CodeBERT, GPT-3.5, GPT-4.0) across 9 different languages. The performance mainly comes from the combination of COT and QA-Checker. Thus, we design an additional version called CodeAgent<sub>w/o</sub>, which means CodeAgent without QA-Checker. Then, we use CodeAgent<sub>w/o</sub> to do vulnerability analysis and compare with CodeAgent. We first discuss about the result of CodeAgent<sub>w/o</sub> and then discuss about comparison between CodeAgent and CodeAgent<sub>w/o</sub>.

**5.18.0.0.1 Overview of Vulnerabilities in CodeAgent<sub>w/o</sub>** Table 5.11 presents the findings of CodeAgent<sub>w/o</sub>, a variant of the original CodeAgent, in identifying vulnerabilities across different programming languages. The table showcases the number of ‘merged’ and ‘closed’ vulnerabilities in languages such as Python, Java, Go, C++, JavaScript, C, C#, PHP, and Ruby. Notably, Python leads in the ‘merged’ category with a total of 1,057 cases, of which 140 are confirmed, yielding a Rate<sub>merge</sub> of 13.25%. In contrast, languages like Go and Ruby show lower vulnerability counts in both ‘merged’ and ‘closed’ categories. The table also includes Rate<sub>close</sub> and Rate<sub>avg</sub>, providing insights into the effectiveness of vulnerability management across these languages.

**Python**

**Commit**

**Commit Message**

**Original File**

```

@@ -365,8 +365,8 @@ def handle_connect(self, params):
    # Welcome to Centrifuge dear Connection!
    self.is_authenticated = True
    self.default_info = {
-     'user_id': self.user,
-     'client_id': self.uid,
+     'user': self.user,
+     'client': self.uid,
        'default_info': info,
        'channel_info': None
    }
@@ -421,9 +421,6 @@ def handle_refresh(self, params):
    """
    Handle request with refreshed connection timestamp
    """
-    if not self.is_authenticated:
-        raise Return((None, self.application.UNAUTHORIZED))
-
    project_name = params["project"]
    user = params["user"]
    timestamp = params["timestamp"]

```

rename client\_id and user\_id into client and user in connection info

```

...
class Client(object):
    """
    This class describes a single connection of
    client.
    """
    application = None

    def __init__(self, sock, info):
        self.sock = sock
        self.info = info
        self.uid = uuid.uuid4().hex
        self.is_authenticated = False
        self.user = None
        self.timestamp = None
        self.channel_info = {}
        self.default_info = {}
        self.project_name = None
        self.channels = None
        self.presence_ping_task = None
        self.expire_timeout = None
...

```

**CodeAgent**

**Summary**

**Semantic Consistency Analysis**

The semantic consistency between the code changes and the commit message is generally good. However, there are a few inconsistencies that should be addressed. In the 'handle\_connect' method, the 'default\_info' dictionary is being updated to use the keys 'user' and 'client' instead of 'user\_id' and 'client\_id' as mentioned in the commit message. It would be better to update the commit message to reflect this change or update the code to use the original keys. Additionally, in the 'handle\_refresh' method, the check for 'is\_authenticated' has been removed, which is inconsistent with the commit message. It would be helpful to clarify the reason for this change in the commit message.

**Security Analysis**

The provided code appears to have some potential security vulnerabilities that should be addressed. Firstly, there is no input validation or sanitization for user input, which can lead to SQL injection, XSS, and command injection risks... The code should also be reviewed for proper authentication and authorization processes, as well as the handling of sensitive data to prevent unauthorized access and data breaches. Proper error handling and exception management should be implemented to avoid leaking sensitive information and causing service interruptions...

**Format Analysis**

The format of the code is generally consistent with the writing style and format of the original file. However, there are a few inconsistencies that could impact readability and maintainability. In the 'handle\_connect' method, the indentation of the 'self.default\_info' dictionary is inconsistent. It would be better to align the indentation to improve code readability. Additionally, in the 'handle\_refresh' method, the comment at the end of the line should be moved to a separate line for better code formatting. These minor formatting issues can be easily addressed to improve the overall quality of the code.

**Suggestion**

Based on the analysis, the following suggestions are provided for code alignment and revisions: 1). In the 'handle\_connect' method, update the commit message to reflect the change in keys from 'user\_id' and 'client\_id' to 'user' and 'client'. Alternatively, update the code to use the original keys mentioned in the commit message. 2). In the 'handle\_refresh' method, clarify the reason for removing the 'is\_authenticated' check in the commit message. If it was intentional, provide an explanation for the change. 3). Improve input validation and sanitization techniques to prevent SQL injection, XSS, and command injection risks. Implement parameterized queries for database operations and escape user-generated content displayed in HTML. 4). Ensure robust memory management practices to avoid buffer overflows in lower-level languages. 5). Review and enhance authentication and authorization processes, as well as the handling of sensitive data, to prevent unauthorized access and data breaches...

**Figure 5.13:** Example in Python project with sha value: “0d3b94bbd7...”

**5.18.0.0.2 Detailed Comparison between CodeAgent and CodeAgent<sub>w/o</sub>** Comparing the findings in Table 5.11 with those in Table 5.9, we observe some notable differences in vulnerability detection by CodeAgent and CodeAgent<sub>w/o</sub>. While the overall trend of higher ‘merged’ vulnerabilities in Python and lower counts in Go and Ruby remains consistent, Table 5.11 shows a slight reduction in the Rate<sub>merge</sub> for most languages, suggesting a more conservative confirmation approach in CodeAgent<sub>w/o</sub>. Similarly, Rate<sub>close</sub> and Rate<sub>avg</sub> values in Table 5.11 generally indicate a lower proportion of confirmed vulnerabilities compared to Table 5.9, reflecting potentially different criteria or efficacy in vulnerability

 Java
 Commit
 Commit Message
 Original File

```

@@ -403,7 +403,7 @@ public boolean canDismiss(Object token) {
    @Override
    public void onDismiss(View view, Object token) {
        if (view != null) {
-            finish();
+            dismiss(false);
        }
    }
@@ -512,6 +512,10 @@ private void startTimer(long duration) {
    public void dismiss() {
+        dismiss(mAnimated);
+    }
+    private void dismiss(boolean animate) {
        if (mIsDismissing) {
            return;
        }
@@ -522,7 +526,7 @@ public void dismiss() {
        mEventListener.onDismiss(Snackbar.this);
    }
-    if (!mAnimated) {
+    if (!animate) {
        finish();
        return;
    }

```

Fixed issue where onDismiss() did not get called if Snackbar is swiped.

```

...
public static Snackbar with(Context context) {
    return new Snackbar(context);
}
/**
 * Sets the type of {@link Snackbar} to be displayed.
 */
* @param type the {@link SnackbarType} of this instance
* @return
public Snackbar type(SnackbarType type) {
    mType = type;
    return this;
}
/**
 * Sets the text to be displayed in this {@link Snackbar}
...

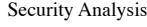
```

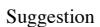
 +  +  Java

Summary

**Semantic Consistency Analysis**

The code changes in the commit do not accurately reflect the description provided in the commit message. The commit message mentions fixing an issue where onDismiss() did not get called if Snackbar is swiped, but the code changes do not address this issue. Instead, the code changes seem to be related to dismissing the Snackbar.





No security analysis can be performed as the code is not provided.

The format of the code aligns with the writing style and format of the original file. There are no formatting inconsistencies that impact the overall readability and maintainability of the project.

The highest priority comment is regarding the semantic consistency between the code changes and the commit message. The code changes should address the issue mentioned in the commit message, i.e., fixing the issue where onDismiss() did not get called if Snackbar is swiped. To fix this issue, the following changes can be made:

1. In the 'Snackbar' class, modify the `dismiss()` method as follows:

```

public void dismiss() {
    dismiss(mAnimated);
}
private void dismiss(boolean animate) {
    if (mIsDismissing) {
        return;
    }
    mIsDismissing = true;
    if (mEventListener != null) {
        mEventListener.onDismiss(Snackbar.this);
    }
    if (!animate) {
        finish();
        return;
    }
}

```

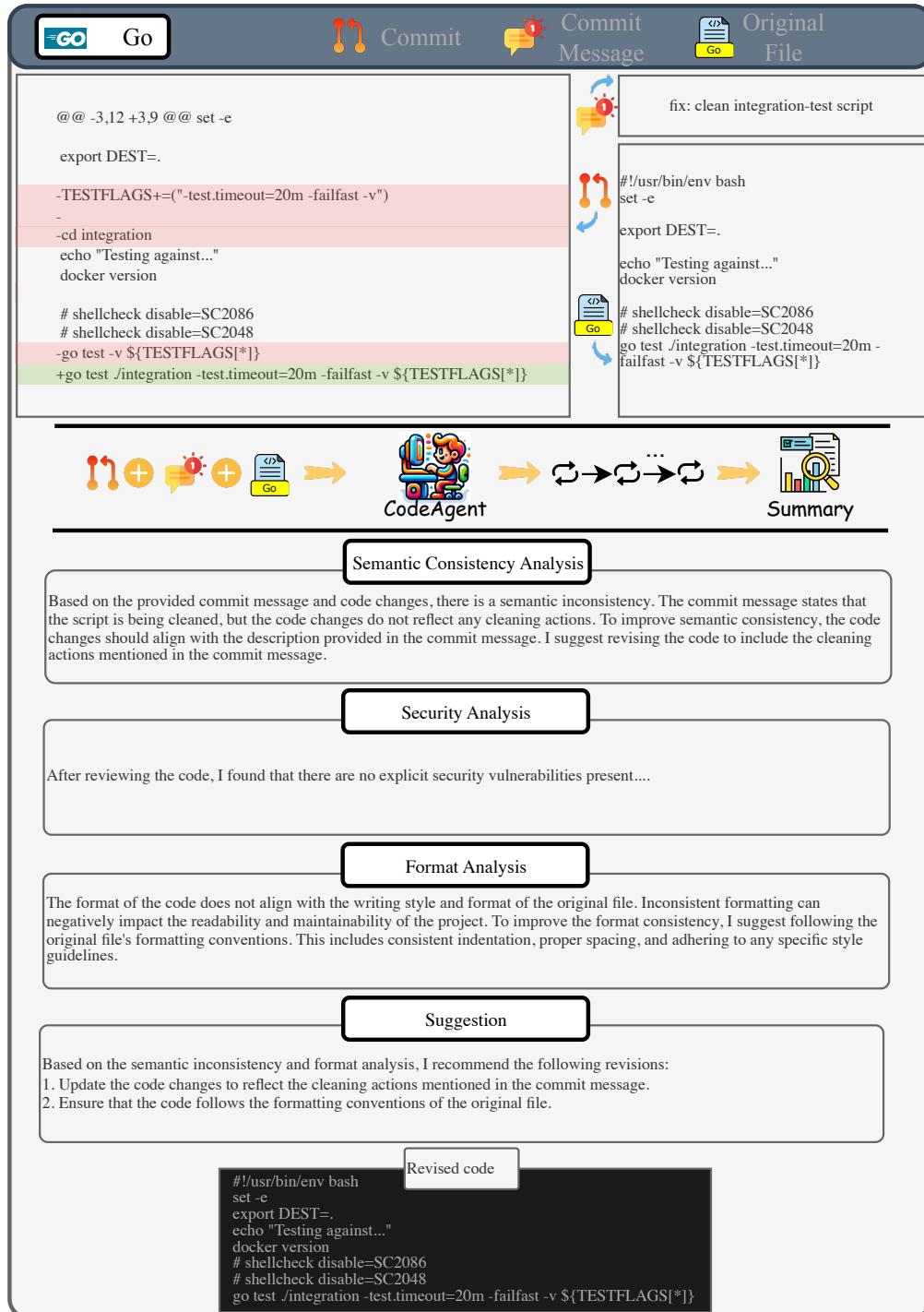
2. Please make the above changes to ensure that the code aligns with the description provided in the commit message.

```

...
private void startTimer(long duration) {
    // implementation
}
public void dismiss() {
    dismiss(mAnimated);
}
private void dismiss(boolean animate) {
    if (mIsDismissing) {
        return;
    }
    mIsDismissing = true;
    if (mEventListener != null) {
        mEventListener.onDismiss(Snackbar.this);
    }
}

```

**Figure 5.14:** Example in Java project with sha value: “6e3c6d17d9...”



**Figure 5.15:** Example in Go project with sha value: “a6b00b09...” assessment. These variations highlight the impact of QA-Checker in CodeAgent.

## 5.19 Cost statement

As shown in Table 5.12, CodeAgent-4 has a higher query time and cost compared to CodeAgent-3.5 due to its increased complexity and capabilities. We acknowledge that the integration of AI models and multi-agent systems may introduce complexity and require specialized knowledge for implementation.

The screenshot shows the CodeAgent interface for a C++ project. The main area displays a code editor with a diff view, showing changes made to a file named `master_server_routes.cpp`. The commit message is visible above the code, and the original file is shown below. The interface includes a navigation bar with icons for commit, message, file, and code analysis. Below the editor are sections for Semantic Consistency Analysis, Security Analysis, Format Analysis, and Suggestion. A sidebar on the right shows a flowchart of the analysis process.

**Semantic Consistency Analysis**

The code changes in the provided code are not consistent with the description provided in the commit message. The commit message mentions the removal of the `"/analytics/events/:name"` endpoint, but this change is not reflected in the code. It is important to ensure that the code accurately reflects the changes described in the commit message to avoid confusion and potential issues.

**Security Analysis**

The provided code appears to have some potential security vulnerabilities that should be addressed. Firstly, there is no input validation or sanitization for user input, which can lead to SQL injection, XSS, and command injection risks... The code should also be reviewed for proper authentication and authorization processes, as well as the handling of sensitive data to prevent unauthorized access and data breaches. Proper error handling and exception management should be implemented to avoid leaking sensitive information and causing service interruptions...

**Format Analysis**

The format of the code does not align with the writing style and format of the original file. There are inconsistencies in indentation and spacing, which can impact the overall readability and maintainability of the project. It is recommended to follow a consistent coding style throughout the codebase to improve readability and make it easier for developers to understand and maintain the code.

**Suggestion**

1. Remove the `"/analytics/events/:name"` endpoint as mentioned in the commit message.
2. Ensure consistent indentation and spacing throughout the code to improve readability.
3. Consider organizing the code into logical sections or modules to enhance maintainability.

**Figure 5.16:** Example in C++ project with sha value: “09e88c7c2...”

**Table 5.11:** Vulnerable problems (#) found by CodeAgent<sub>w/o</sub>

CodeAgent	Python	Java	Go	C++	JavaScript	C	C#	PHP	Ruby
merged (total#)	1,057	287	133	138	280	114	206	173	202
merged (confirmed#)	140	17	10	12	28	9	21	28	17
Rate <sub>merge</sub>	13.25%	5.92%	7.52%	8.70%	10.00%	7.89%	10.19%	16.18%	8.42%
closed (total#)	248	97	74	56	112	146	62	105	55
closed (confirmed#)	36	9	5	12	16	26	7	15	5
Rate <sub>close</sub>	14.52%	9.28%	6.76%	21.43%	14.29%	17.81%	11.29%	14.29%	9.09%
Total number (#)	1,305	384	207	194	392	260	268	278	257
Total confirmed (#)	176	26	15	24	44	35	28	43	22
Rate <sub>avg</sub>	13.49%	6.77%	7.25%	12.37%	11.22%	13.46%	10.45%	15.47%	8.56%

**Javascript** Commit Commit Message Original JS File

```
@@ -225,6 +225,22 @@ window['$'] = window['quip'] = (function(){
  }
  return this;
};

+ p['replaceWith'] = function(val){
+   if (isF(val)) {
+     return this['each']().function(i) {
+       var self = $(this), old = self.html();
+       self.replaceWith( val.call(this, i, old) );
+     };
+   } else if (isS(val)) {
+     val = $(val).detach();
+   }
+   return this['each']().function() {
+     var next = this.nextSibling;
+     parent = this.parentNode;
+     parent.removeChild(this);
+     (next ? $(next).before(val) : $(parent).append(val));
+   };
+ };
p['hide'] = function(){
  return this['each']().function(){
    cache(this, "display", this.style.display);
  }
};

+ p['after'] = function(){
  if (this[0] && this[0].parentNode){
    return this.dm(arguments, false,
      function(el){
        this.parentNode.insertBefore(el,
        this.nextSibling);
      });
  }
  return this;
};

p['replaceWith'] = function(val){
  if (isF(val)) {
    return this['each']().function(i) {
      var self = $(this), old = self.html();
      self.replaceWith( val.call(this, i, old) );
    };
  } else if (isS(val)) {
    val = $(val).detach();
  }
...

```

**CodeAgent** Summary

**Semantic Consistency Analysis**

The semantic consistency between the code changes and the commit message is generally good. The changes accurately reflect the description provided in the commit message. However, there is one inconsistency in the code. In the added 'replaceWith' function, there is a missing check for the 'val' parameter being a function. Currently, the code assumes that 'val' will always be a function or a string. It would be better to add a check for the function type and handle it accordingly. This will ensure that the code is more robust and can handle different types of input.

**Security Analysis**

The security analysis of the provided code reveals some potential vulnerabilities. The code does not validate user input, which can lead to SQL injection, XSS, and command injection risks. It is crucial to implement input validation and sanitization techniques to prevent these vulnerabilities....

**Format Analysis**

The format of the code aligns with the writing style and format of the original file. There are no formatting inconsistencies that impact the overall readability and maintainability of the project. The code follows a consistent indentation style and uses appropriate naming conventions. However, it is recommended to add comments to explain the purpose and functionality of complex code sections to improve code understandability.

**Suggestion**

1. In the 'replaceWith' function, add a check for the 'val' parameter being a function and handle it accordingly.
2. Implement input validation and sanitization techniques to prevent SQL injection, XSS, and command injection risks.
3. Improve error handling mechanisms to avoid sensitive information leakage and service interruptions.
4. Review the code for any deprecated functions, hardcoded sensitive data, or code leakages.
5. Add comments to explain the purpose and functionality of complex code sections.

**Revised Code**

```
window['$'] = window['quip'] = (function(){
  var win = window,
    queryShimCdn = "http://cdnjs.cloudflare.com/ajax/libs/sizzle/1.4.4/sizzle.min.js",
    queryEngines = function(){ return win["Sizzle"] || win["qryery"]; },
    doc = document, docEl = doc.documentElement,
    scriptFns = [], load =[], $l = $,
    runtil = /Until$/.multiselector = /./,
    rparentsprev = /(?:parents|prevUntil|prevAll)/,
    rtagname = /<(\w+)>/.
```

**Figure 5.17:** Example in JavaScript project with sha value: “6e9a80d93...”

The screenshot shows the CodeAgent interface. At the top, there are tabs for 'C' (selected), 'Commit', 'Commit Message', and 'Original File'. The 'Commit' tab has a 'Commit' icon. The 'Commit Message' tab has a 'Commit' icon with a red exclamation mark. The 'Original File' tab has a 'File' icon with a yellow 'C'.

**Code Diff:** Shows a diff between two versions of a file. The left side shows the code with changes highlighted in red and green. The right side shows the commit message and the corresponding code changes.

**Commit Message:** The commit message is "remove ventoy.net in grub UI". The code changes are:

```
@@ -757,9 +757,9 @@ <h4>
<footer class="main-footer">
  <div class="pull-right hidden-xs">
-   <b id="plugsong_build_date">20231014 18:52:12</b>
+   <b id="plugsong_build_date">20231014 18:52:12</b>
  </div>
-   <strong><a href="https://www.ventoy.net">https://www.ventoy.net</a></strong>
+   </strong>
</footer>
<!-- Control Sidebar -->
```

**Flow Diagram:** Shows the workflow: Commit → CodeAgent → Analysis (Semantic Consistency Analysis, Security Analysis, Format Analysis) → Summary.

**Semantic Consistency Analysis:** A box stating: "Based on the provided code and commit message, the semantic consistency analysis reveals that the changes in the code accurately reflect the description provided in the commit message. There are no inconsistencies or potential hidden malicious code identified."

**Security Analysis:** A box stating: "The security analysis of the provided code reveals several potential vulnerabilities. Firstly, there is no validation of user input, which can lead to SQL injection, XSS, and command injection risks. It is recommended to implement proper input validation and sanitization techniques to prevent these vulnerabilities. Additionally, the code does not handle errors and exceptions properly, which can result in sensitive information leakage and service interruptions..."

**Format Analysis:** A box stating: "The format analysis reveals that the code formatting is inconsistent with the writing style and format of the original file. This inconsistency can impact the overall readability and maintainability of the project. It is recommended to align the code formatting with the original file's style and format to improve code quality."

**Suggestion:** A box with a numbered list of recommendations:

1. Implement input validation and sanitization techniques to prevent SQL injection, XSS, and command injection risks.
2. Improve error handling mechanisms to avoid sensitive information leakage and service interruptions.
3. Review the code for any deprecated functions, hardcoded sensitive data, or code leakages.
4. Conduct a thorough review of all dependencies, APIs, and configurations, including third-party libraries
5. Align the code formatting with the writing style and format of the original file to improve code quality and maintainability.

**Figure 5.18:** Example in C project with sha value: “5f93cf43d...”

**Table 5.12:** Summarizes the average query time and cost per code review for CodeAgent-3.5 and CodeAgent-4.

Model	Query Time(min)	Cost in USD
CodeAgent-3.5	3	0.017
CodeAgent-4	5	0.122

## 5.20 Tool

We develop a website for CodeAgent, which is shown in Figure 5.23, and it is also accessible by visiting following link:

The screenshot shows the CodeAgent interface for a C# project. At the top, there are tabs for 'C#' (selected), 'Commit', 'Commit Message', and 'Original File'. The 'Commit' tab shows a diff with a red minus sign and a green plus sign, indicating a change from `ILoggerFactory` to `XunitLogger<Kernel>` in the `InitializeKernel` method. The 'Commit Message' tab shows the message: 'change logger to concrete type'. The 'Original File' tab shows the full code. Below these are several analysis steps: 'Semantic Consistency Analysis' (warning about inconsistency in the commit message), 'Security Analysis' (warning about SQL injection risks), 'Format Analysis' (warning about formatting), and 'Suggestion' (warning to update the logger type in the code and commit message). The interface has a central 'CodeAgent' icon and a 'Summary' icon at the bottom.

@@ -148,7 +148,7 @@ private Kernel InitializeKernel()

```

private readonly RedirectOutput _testOutputHelper;
private readonly IConfigurationRoot _configuration;
- private readonly ILoggerFactory _logger;
+ private readonly XunitLogger<Kernel> _logger;

public void Dispose()
{

```

change logger to concrete type

```

...
builder.AddOpenAIChatCompletion(
    modelId:
    openAIConfiguration.ModelId,
    apiKey:
    openAIConfiguration.ApiKey);

var kernel = builder.Build();

return kernel;
}

private readonly RedirectOutput
testOutputHelper;
private readonly IConfigurationRoot
configuration;
private readonly XunitLogger<Kernel>
logger;
...

```

**Semantic Consistency Analysis**

The semantic consistency between the code changes and the commit message is generally good. The code changes accurately reflect the description provided in the commit message. However, there is one inconsistency in the code. In the `InitializeKernel` method, the `\_logger` field is changed from `ILoggerFactory` to `XunitLogger<Kernel>`. The commit message does not mention this change, which could lead to confusion for other developers. It is recommended to update the commit message to include this change.

**Security Analysis**

The security analysis of the provided code reveals several potential vulnerabilities. Firstly, there is no validation of user input to prevent SQL injection, XSS, and command injection risks. ..

**Format Analysis**

The format of the code aligns well with the writing style and format of the original file. There are no significant formatting inconsistencies that impact the overall readability and maintainability of the project.

**Suggestion**

In the `InitializeKernel` method, the `\_logger` field should be updated to use the concrete type `XunitLogger<Kernel>` instead of the abstract type `ILoggerFactory`. This change should be reflected in the commit message as well.

**Figure 5.19:** Example in C# project with sha value: “0e231c7a...”

<https://code-agent-new.vercel.app/index.html>

**PHP**

```

...function get_posts( $args = null ) {
@@ -2425,6 +2425,7 @@ function get_posts( $args = null ) {
    'meta_value'      => '',
    'post_type'       => 'post',
    'suppress_filters' => true,
+   'get_post_meta' => false
);
$parsed_args = wp_parse_args( $args, $defaults );
@@ -2449,7 +2450,20 @@ function get_posts( $args = null ) {
    $parsed_args['no_found_rows'] = true;

    $get_posts = new WP_Query();
-   return $get_posts->query( $parsed_args );
+   $get_posts = $get_posts->query( $parsed_args );
+
+   if ( !empty( $parsed_args['get_post_meta'] ) ) {
+       foreach ( $get_posts as $post ) {
+           $post_meta = [];
+           foreach ( get_post_meta( $post->ID ) as $key => $value ) {
+               $post_meta[ $key ] = $value[0];
+               $post->meta_data = $post_meta;
+               $get_posts[] = $post;
+           }
+       };
+   }
+   return $get_posts;
}

```

**Commit**

**Commit Message**

**Original File**

**CodeAgent**

**Summary**

**Semantic Consistency Analysis**

The semantic consistency between the code changes and the commit message seems to be accurate. The changes made in the code align with the description provided in the commit message. There are no inconsistencies or potential hidden malicious code identified in this aspect.

**Security Analysis**

The provided code requires a comprehensive security review to identify potential vulnerabilities. The analysis should focus on the following areas:.....Based on the provided code, it is difficult to perform a comprehensive security analysis. However, it is recommended to conduct a thorough security review considering the above points to identify any potential vulnerabilities.

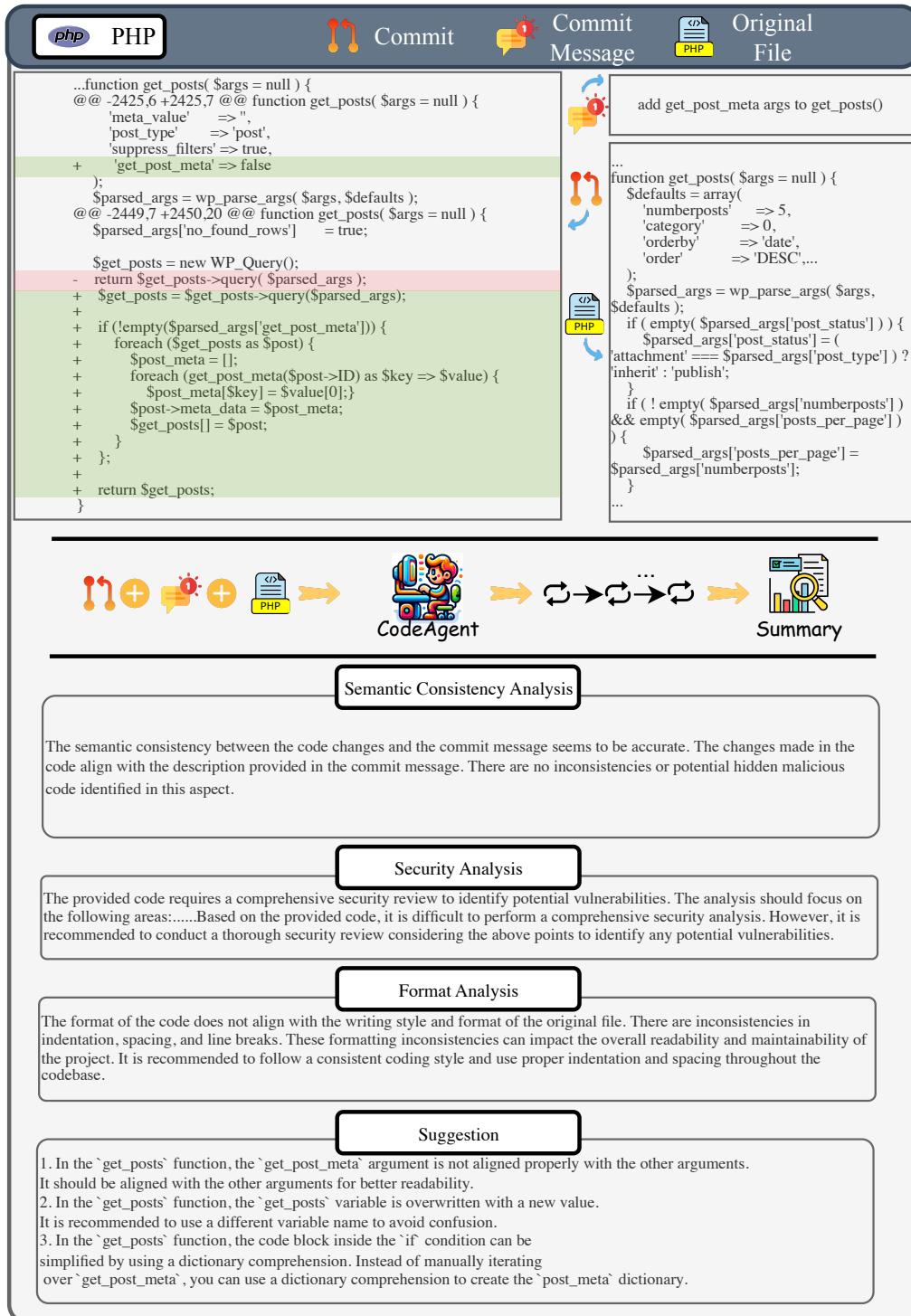
**Format Analysis**

The format of the code does not align with the writing style and format of the original file. There are inconsistencies in indentation, spacing, and line breaks. These formatting inconsistencies can impact the overall readability and maintainability of the project. It is recommended to follow a consistent coding style and use proper indentation and spacing throughout the codebase.

**Suggestion**

- In the `get\_posts` function, the `get\_post\_meta` argument is not aligned properly with the other arguments. It should be aligned with the other arguments for better readability.
- In the `get\_posts` function, the `get\_posts` variable is overwritten with a new value. It is recommended to use a different variable name to avoid confusion.
- In the `get\_posts` function, the code block inside the `if` condition can be simplified by using a dictionary comprehension. Instead of manually iterating over `get\_post\_meta`, you can use a dictionary comprehension to create the `post\_meta` dictionary.

**Figure 5.20:** Example in PHP project with sha value: “6679f059b...”

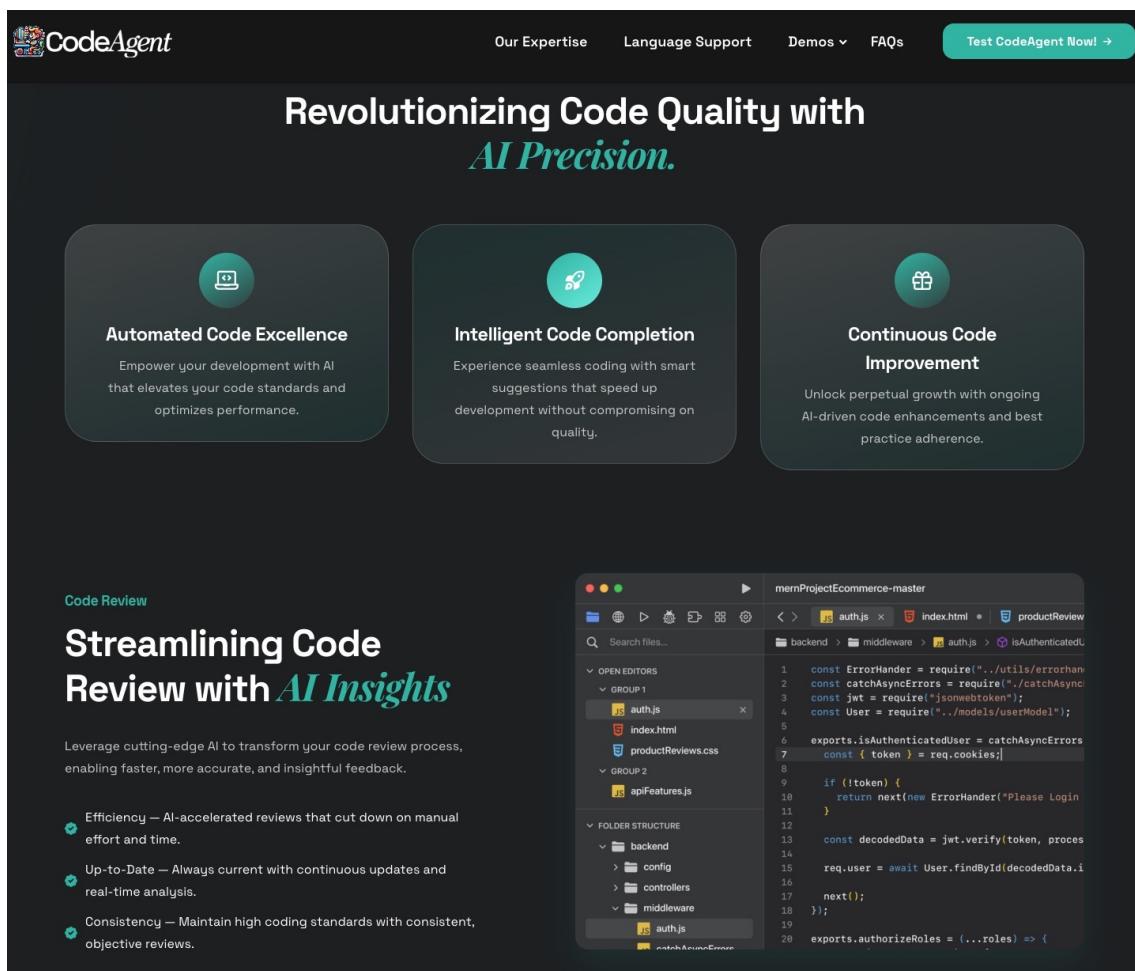


**Figure 5.21:** Example in Ruby project with sha value: “584f72e7f4...”

The screenshot shows a Java code review interface with the following components:

- Java**: The code editor area displays a diff of a Java file. The diff highlights changes in the `onDismiss` and `dismiss` methods. A commit message at the top right states: "Fixed issue where onDismiss() did not get called if Snackbar is swiped." The file has 655 lines.
- Commit**: Shows the commit history with a single commit message.
- Commit Message**: The commit message is "Fixed issue where onDismiss() did not get called if Snackbar is swiped."
- Original File**: A preview of the original Java file content.
- CodeAgent-3.5** and **CodeAgent-4.0**: Two sections comparing the analysis results of the two agents.
- Semantic Consistency Analysis**:
  - CodeAgent-3.5**: States that the code changes do not accurately reflect the commit message, which mentioned fixing an issue where `onDismiss` did not get called if Snackbar is swiped. The changes seem to be related to dismissing the Snackbar.
  - CodeAgent-4.0**: States that the code changes reflect the commit message accurately, fixing the issue where `onDismiss` did not get called if Snackbar is swiped. The changes include calling `dismiss(false)` and adding a new `dismiss` method with a boolean parameter.
- Security Analysis**:
  - CodeAgent-3.5**: No security analysis can be performed as the code is not provided.
  - CodeAgent-4.0**: The code does not seem to have any security vulnerabilities, as it does not handle user input directly.
- Format Analysis**:
  - CodeAgent-3.5**: The format aligns with the writing style and format of the original file.
  - CodeAgent-4.0**: The format aligns with the writing style and format of the original file, with no formatting inconsistencies.
- Suggestion**:
  - CodeAgent-3.5**: The highest priority comment is regarding semantic consistency between code changes and the commit message, suggesting to fix the issue where `onDismiss` did not get called if Snackbar is swiped.
  - CodeAgent-4.0**: The code changes are well-aligned with the rest of the codebase, following naming conventions and style.

**Figure 5.22:** Comparison of code review between CodeAgent-3.5 and CodeAgent-4.0 on one Java project with sha value: “6e3c6d17...”



The image shows the homepage of CodeAgent and a code editor interface. The homepage features a dark header with the CodeAgent logo, navigation links for 'Our Expertise', 'Language Support', 'Demos', 'FAQs', and a 'Test CodeAgent Now!' button. The main title is 'Revolutionizing Code Quality with AI Precision.' Below the title are three rounded cards: 'Automated Code Excellence' (with a gear icon), 'Intelligent Code Completion' (with a rocket icon), and 'Continuous Code Improvement' (with a gift icon). The code editor interface on the right shows a file structure for 'memnProjectEcommerce-master' with 'backend' and 'middleware' folders. The 'auth.js' file is open, displaying the following code:

```

memnProjectEcommerce-master
backend > middleware > auth.js > isAuthenticated.js

1  const ErrorHander = require("./utils/errorhan
2  const catchAsyncErrors = require("./catchAsync
3  const jwt = require("jsonwebtoken");
4  const User = require("../models/userModel");
5
6  exports.isAuthenticatedUser = catchAsyncErrors
7  const { token } = req.cookies;
8
9  if (!token) {
10    return next(new ErrorHander("Please Login
11  })
12
13  const decodedData = jwt.verify(token, proces
14
15  req.user = await User.findById(decodedData.i
16
17  next();
18  });
19
20  exports.authorizeRoles = (...roles) => {

```

Figure 5.23: website of CodeAgent



# 6 SynFix: Synchronous Repair for Codebase via CallGraph

*In this chapter, we propose to investigate the challenges of repository-level debugging where existing methods often fail to capture the broader context of dependencies between files, classes, and functions. Our approach, SynFix, addresses these limitations through a novel CallGraph-driven framework that systematically models structural relationships within a codebase, ensuring that fixes propagate correctly across interconnected components. By shifting from iterative agent-based exploration to a structured, deterministic process with four key phases—CallGraph modeling, hierarchical localization, synchronous repair, and paired patch validation—SynFix achieves comprehensive and consistent fixes at the repository level. Our extensive evaluation on SWE-bench benchmarks demonstrates that SynFix significantly outperforms existing agent-based approaches, resolving 52.33% of issues in SWE-bench-lite, 55.8% in SWE-bench-verified, and 29.86% in SWE-bench-full. These results highlight the effectiveness of our structured approach in maintaining repository-wide consistency during program repair while eliminating the inefficiencies and cascading errors common in multi-turn decision-making systems.*

The content presented herein derives from scholarly investigation previously documented in the academic publication referenced below:

- **Xunzhu Tang**, Jiechao Gao, Jin Xu, Tiezhu Sun, Yewei Song, Saad Ezzini, Wendûuni C. Ouédraogo, Jacques Klein, and Tegawendé F. Bissyandé. SynFix: Synchronous Repair for Codebase via RelationGraph. In *Findings of the Association for Computational Linguistics (ACL 2025)*, 2025. (Accepted)

## Contents

---

6.1	Overview	137
6.2	Methodology	139
6.2.1	CallGraph Building	139
6.2.2	Localization	140
6.2.3	Synchrony	141
6.2.4	Paired Patches Validation	142
6.2.5	Datasets	144
6.2.6	Implementation	144

6.2.7	Baselines . . . . .	144
6.2.8	Research Questions . . . . .	144
<b>6.3</b>	<b>Experimental Results . . . . .</b>	<b>145</b>
6.3.1	RQ1: Overall Performance . . . . .	145
6.3.2	RQ2: Ablation Study . . . . .	146
<b>6.4</b>	<b>Conclusion . . . . .</b>	<b>148</b>
<b>6.5</b>	<b>Efficiency Considerations with GPT-4o . . . . .</b>	<b>149</b>
<b>6.6</b>	<b>Matching Prompt for Using Chatgpt 3.5 . . . . .</b>	<b>149</b>
<b>6.7</b>	<b>Analysis Prompt for Using Chatgpt 4o . . . . .</b>	<b>149</b>
<b>6.8</b>	<b>Results on SWE-bench Full . . . . .</b>	<b>149</b>
<b>6.9</b>	<b>Results on SWE-bench Verified . . . . .</b>	<b>150</b>
6.9.1	External Experiment Across Foundation Models . . . . .	150
<b>6.10</b>	<b>CallGraph Driven Error Detection and Synchronous Repair . . . . .</b>	<b>151</b>

---

## 6.1 Overview

Large language models (LLMs) have transformed software engineering by automating tasks such as bug localization, program repair, and test generation [Chen et al., 2021b, Austin et al., 2021, Li et al., 2023d, Wei et al., 2023b]. However, applying LLMs to repository-level debugging still presents significant challenges. While existing methods do well at resolving isolated issues, they often fail to capture the broader context of repository-wide dependencies between files, classes, and functions [Zhou et al., 2012, Austin et al., 2021]. Repository-level tasks are inherently difficult, involving complex interactions across components that require a holistic understanding of the codebase. Traditional approaches struggle to address these challenges, often leading to partial or suboptimal fixes. Consequently, there is a pressing need for solutions that can effectively model and address the interconnected nature of large-scale codebases.

To bridge this gap, benchmarks like SWE-bench and its refined subset, SWE-bench Lite Jimenez et al. [2024a], swe [2024], provide rigorous evaluations of real-world software development tasks. SWE-bench encapsulates a diverse range of issues from open-source repositories, challenging tools to address complex problems such as multi-file dependency handling, parameter synchronization, and regression testing. The SWE-bench Lite subset focuses specifically on bug-fixing tasks, which offers a controlled environment to test advanced repair methods. These benchmarks not only highlight the limitations of current approaches but also provide a foundation for developing tools that can handle repository-wide debugging with precision and scalability. By simulating real-world scenarios, SWE-bench ensures that solutions are applicable to practical software engineering problems, making it a critical resource for evaluating modern debugging tools.

Agent-based systems have been proposed to address these repository-level challenges by equipping LLMs with toolsets for iterative decision-making [Zhang et al., 2024a, Yang et al., 2024c]. These systems enable LLMs to autonomously decide on actions, perform operations like file edits or test executions, and iteratively refine their solutions based on feedback. While promising in theory, agent-based approaches face key limitations in practice, as the complexity of tool usage often introduces abstraction layers that are prone to errors, especially when mapping actions to APIs Liu et al. [2024b]. Moreover, the lack of robust planning mechanisms means that agents frequently make suboptimal decisions, leading to inefficiencies and unnecessary costs. The iterative, multi-turn nature of these methods exacerbates these issues, as incorrect decisions in early stages can cascade into larger problems, significantly hindering performance Olausson et al. [2023], Shi et al. [2023]. Furthermore, their limited ability to self-reflect and filter irrelevant or incorrect feedback amplifies these challenges, making agent-based systems less reliable for large-scale debugging tasks.

To address these limitations, we propose **SynFix**, a CallGraph-driven approach that avoids the complexities of autonomous multi-turn decision-making. SynFix shifts the focus from iterative exploration to structured, deterministic processes that explicitly model the structural relationships within a codebase. Central to SynFix is the **CallGraph**, which captures dependencies between files, functions, and variables, enabling a comprehensive understanding of the codebase. This representation ensures that fixes in one part of the codebase are propagated correctly across related components, maintaining repository-wide consistency. By leveraging the CallGraph, SynFix systematically localizes issues, identifies affected contexts, and synchronizes repairs to address interdependencies effectively. Unlike agent-based systems, SynFix avoids uncontrolled exploration, ensuring that each repair step is precise and well-informed.

SynFix follows a straightforward four-phase pipeline: CallGraph, Localization, Synchrony, and Paired Patch Validation. In the CallGraph phase, SynFix models the structural dependencies in the codebase. During the Localization phase, it uses this framework to identify suspicious files, classes, and specific lines of code. Synchrony ensures that fixes propagate correctly across interconnected components, while Paired Patch Validation rigorously tests the patches using regression suites to ensure correctness and compatibility with existing functionality. This deterministic and interpretable design eliminates the inefficiencies of agent-based systems, offering a cost-effective solution for real-world software debugging. SynFix has demonstrated exceptional performance on SWE-bench benchmarks [swe, 2024], achieving state-of-the-art results while maintaining scalability and reliability. Its success highlights the potential of structured, agentless approaches to set new standards for efficient, repository-level program repair.

## Contributions

- **CallGraph-Driven Framework:** SynFix introduces a novel CallGraph-based approach that systematically models the structural and dependency relationships within a repository. This enables precise localization and coordinated synchronization of fixes, ensuring repository-wide consistency.
- **Enhanced Localization and Synchronization:** SynFix employs a hierarchical localization strategy to pinpoint issues with high precision and ensures that fixes propagate correctly across interconnected components through its Synchrony phase.
- **Efficient Validation Mechanism:** Through Paired Patch Validation, SynFix rigorously tests proposed patches using regression suites, ensuring correctness and compatibility while minimizing the risk of introducing new errors.
- **State-of-the-Art Performance:** SynFix achieves superior results on SWE-bench benchmarks, surpassing agent-based baselines in resolution rates, scalability, and cost-efficiency.

## 6.2 Methodology

Figure 6.2 presents an overview of PANTHER, which operates in four main phases: **call-graph** (1), **localization** (2), **synchrony** (3), and **paired patches validation** (4, 5, 6, 7, and 8). Given a problem statement and an existing codebase as inputs, PANTHER proceeds as follows. First, PANTHER constructs a CallGraph from the input codebase, capturing its hierarchical structure, variants, classes, functions, and their calling relationships (1). Next, using both the CallGraph and the problem statement, PANTHER prompts a low-cost LLM (e.g., GPT-3.5) to identify and rank the top  $N$  most suspicious nodes that may require edits to address the issue. These nodes are presented with their surrounding context—such as the corresponding function, class, or code variant line (2). To account for synchronous changes, PANTHER then identifies the one-hop neighbors of the suspicious nodes, extracting these neighbors along with their corresponding lines, functions, and classes (3). These extracted contexts serve two purposes: (1) as additional input for the LLM, they help evaluate and refine the understanding of issues in the target suspicious nodes, ultimately facilitating more accurate modifications; and (2) after addressing the primary target nodes, the LLM evaluates whether corresponding updates are necessary for these one-hop neighbors, especially in cases where changes to parameters or related structures in the primary nodes propagate to their neighbors. The first purpose is discussed in detail in the main text, while the second is elaborated upon in the Appendix. With this preliminary set of suspicious nodes identified, PANTHER provides their full code contexts to the LLM. The model refines and narrows down the potential edit locations (e.g., specific lines) for both dynamic and static analyses. This step ensures that the final chosen edit points are as accurate as possible (4). In the repair phase, PANTHER presents the problem statement and the identified edit locations to the LLM. The model is then prompted to propose patches aimed at resolving the identified issues (5). Once patches are generated, PANTHER attempts to apply them to the original codebase. If a patch cannot be cleanly applied, the process returns to the previous step (4), repeating for a preconfigured number of iterations (6). If the patch application is successful, PANTHER validates the updated codebase by running regression tests. If these tests pass, the patch is considered successful (7). Finally, PANTHER updates the CallGraph to reflect the changes in the codebase, ensuring that subsequent iterations start from an accurate and up-to-date state (8). The following sections provide a more detailed explanation of each phase within PANTHER.

### 6.2.1 CallGraph Building

The CallGraph is a directed graph where each node represents a code entity, and edges capture hierarchical or dependency relationships between these entities.

**Nodes:** Each node in the CallGraph corresponds to a distinct code entity within the project, such as: folder name (with relative path), file name (with relative path), variable name, class name, function name. By including nodes at multiple abstraction levels, the CallGraph allows for a holistic understanding of the codebase—from its highest-level directory structure down to the granular level of individual methods.

**Edges:** Edges in the CallGraph represent structural, import-based, and functional dependencies. For example: – Folders and their subfolders, as well as folders and the files within them, are connected by edges to reflect hierarchical relationships. – Files that import other files or modules are linked, indicating a dependency between these code units. – If a variable, class, or method defined in one file is referenced or invoked by another file or entity, an edge is drawn to represent that relationship (e.g., a method A calling method B).

**Subgraph Construction and Merging:** The construction of the full CallGraph may start by building smaller subgraphs derived from individual directories, modules, or sets of

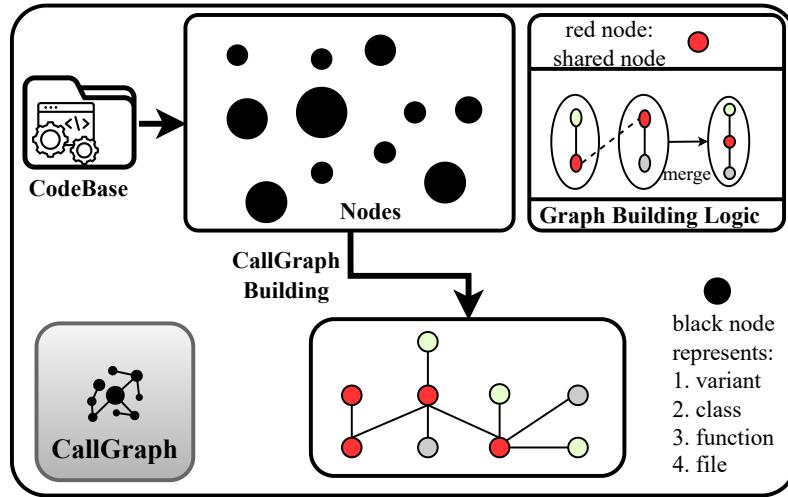


Figure 6.1: Pipeline of CallGraph building.

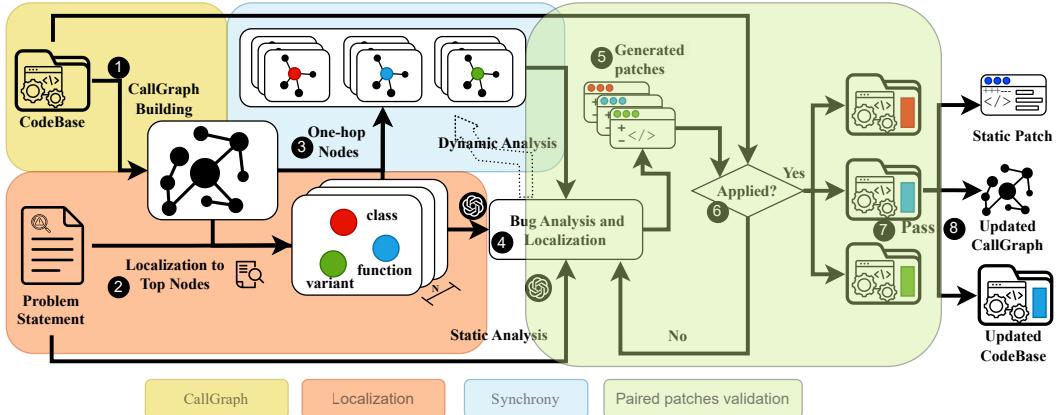


Figure 6.2: Pipeline of PANTHER

interrelated files. Since these local subgraphs can be constructed in parallel across different subfolders, the overall process becomes more efficient. Once each subgraph is completed, it is integrated into the larger CallGraph. In this way, parallelization allows rapid scaling of the graph-building process.

These subgraphs are then iteratively merged into a single, comprehensive CallGraph. When merging, if two subgraphs share an identical node (e.g., the same file, method, or class), these overlapping nodes are merged into a single unified node. Figure 6.1 illustrates an example of such a merging process. Red nodes represent shared nodes in two subgraphs; these are collapsed into a single node, retaining all incoming and outgoing edges from both subgraphs.

## 6.2.2 Localization

Localization is a critical phase in PANTHER, designed to systematically identify the minimal set of code regions requiring modification. The central challenge lies in navigating extensive repositories containing hundreds of files and thousands of lines of code to isolate the few elements pertinent to the problem statement. To address this, PANTHER employs a three-step hierarchical process: suspicious node identification, element refinement, and precise edit localization. By integrating advanced reasoning capabilities of large language models (LLMs) like GPT-3.5 and embedding-based retrieval techniques, PANTHER achieves both accuracy and efficiency in its localization process.

The first step is the identification of suspicious nodes in the CallGraph. Let the CallGraph  $G = (V, E)$ , where  $V$  represents the nodes (e.g., functions, classes, files) and  $E$  denotes the

edges capturing hierarchical or dependency relationships. For a given problem statement  $P$ , PANTHER aims to compute a ranking  $R \subseteq V$ , representing the top  $N$  most suspicious nodes. Using a low-cost LLM such as GPT-3.5, PANTHER constructs a tailored prompt that integrates the CallGraph structure and problem statement to guide the model in ranking nodes by their relevance:

$$R = \operatorname{argmax}_{v \in V} \operatorname{Relevance}(v, P), \quad (6.1)$$

where  $\operatorname{Relevance}(v, P)$  is a function that evaluates the relevance of node  $v$  to the problem statement  $P$  based on its structural, textual, and contextual attributes. To ensure efficiency, PANTHER complements this LLM-based reasoning with an embedding-based retrieval mechanism. Each node  $v \in V$  is embedded into a latent space  $\phi(v)$  using a pre-trained transformer model, and its cosine similarity with the problem statement embedding  $\phi(P)$  is computed:  $\frac{\phi(v) \cdot \phi(P)}{\|\phi(v)\| \|\phi(P)\|}$ .

The top candidates from both methods are combined into a unified ranking  $R$ , ensuring a robust selection of suspicious nodes. To leverage GPT-3.5 for identifying suspicious nodes efficiently and cost-effectively, PANTHER uses the following carefully designed prompt in Appendix Sec 6.6.

After identifying suspicious nodes, PANTHER refines these candidates by analyzing their internal elements, such as specific functions, classes, or variables. A compact representation, called the **skeleton format**, is constructed for each node. This format summarizes critical components, such as function headers, class declarations, and high-level docstrings. For example, a node representing a file might include declarations like:

```
class ClassName (Base) :  
    def method_name(self, arg) :      # Method description. (6.2)
```

By using GPT-3.5, PANTHER enables the model to efficiently identify the most relevant elements within each suspicious node while significantly reducing computational overhead.

Finally, PANTHER pinpoints precise edit locations within the refined elements. Given a function  $f$  or class  $c$  identified in the previous step, the LLM is prompted with its full content to determine specific lines requiring modification. Static analysis methods, such as control flow and data dependency analysis, are employed to identify potential structural inconsistencies. In parallel, dynamic analysis evaluates runtime behaviors such as parameter changes and function calls. These analyses are integrated into a unified decision-making process, ensuring the selected edit locations are accurate and comprehensive. This hierarchical approach not only minimizes the search space but also optimizes the repair process in subsequent stages.

### 6.2.3 Synchrony

In modern software systems, changes to a single code entity can have cascading effects on its dependent functions, classes, or files. Ensuring consistency across these interdependent entities is crucial to maintaining the correctness and coherence of the codebase. The synchrony phase in PANTHER addresses this challenge by identifying and validating one-hop neighbors in the CallGraph, ensuring that modifications made to a suspicious node propagate correctly to its related entities.

For a suspicious node  $v_s \in V$ , its immediate neighbors are defined as  $N(v_s) = \{v \in V \mid (v_s, v) \in E \vee (v, v_s) \in E\}$ . Given a modification  $\Delta(v_s)$  to the suspicious node, the goal is to determine whether any entity in  $N(v_s)$  requires synchronous updates to maintain consistency. The consistency condition can be expressed as:  $\operatorname{Consistent}(\mathcal{X}(v_s), \{\mathcal{X}(u) : u \in N(v_s)\})$ ,

where  $\mathcal{X}(v)$  denotes the local context of a node  $v$ . If this condition is violated, synchronous updates are applied to the affected neighbors.

The process of synchronous repair can be formulated as a constrained optimization problem. Let  $x_v \in \{0,1\}$  denote whether node  $v$  is modified, and let  $c(v)$  represent the cost of modifying  $v$ . The objective is to minimize the total modification cost while ensuring consistency constraints are met:

$$\min \sum_{v \in N(v_s)} c(v)x_v, \quad (6.3)$$

subject to  $\text{Consistent}(\mathcal{X}(v), \{\mathcal{X}(u) : u \in N(v)\})$ .

To implement this process, we outline the synchrony phase in Algorithm 2. The algorithm begins by analyzing the suspicious node  $v_s$  and extracting its neighbors. A consistency check is performed on each neighbor, and if violations are detected, synchronous updates are applied within predefined depth and modification constraints. The algorithm ensures efficient repair by leveraging the CallGraph structure and minimizing the total cost of modifications.

---

**Algorithm 2:** Synchronous Repair Process

**Input** :  $G = (V, E)$ : CallGraph,  $v_s$ : suspicious node,  $\Delta(v_s)$ : modification to  $v_s$   
**Output** : Updated codebase with consistent synchronous modifications

Extract one-hop neighbors:  $N(v_s) \leftarrow \{v \in V \mid (v_s, v) \in E \vee (v, v_s) \in E\}$ .

**foreach**  $v \in N(v_s)$  **do**

**if**  $\text{ConsistencyCheck}(v, \Delta(v_s)) = \text{False}$  **then**  
 style="padding-left: 40px;">**Apply** modification  $\Delta(v)$  **to**  $v$ .

**return** *Updated codebase.*

**Function**  $\text{ConsistencyCheck}(v, \Delta(v_s))$ :

**Input** :  $v$ : neighbor node,  $\Delta(v_s)$ : modification to suspicious node

**Output** : True if consistent, False otherwise

Evaluate consistency:

$\text{Consistent} \leftarrow \text{True}$ .

**if** Changes in  $v_s$  affect dependencies of  $v$  **then**

$\text{Consistent} \leftarrow \text{False}$ .

**return** *Consistent*.

---

The synchrony phase leverages lightweight LLMs like GPT-4o for analyzing consistency and generating modifications, making the process cost-effective. By integrating this phase into PANTHER, we ensure that the repair process accounts for interdependencies within the CallGraph, thereby maintaining the integrity and correctness of the entire codebase.

### 6.2.4 Paired Patches Validation

Validation is a critical phase of PANTHER to ensure that generated patches not only address the identified issue but also preserve the overall functionality of the codebase. Paired patches validation takes this process a step further by leveraging a systematic comparison between the modified code and expected outputs using a curated test suite, referred to as *test\_patches*. This approach ensures that the applied changes are both syntactically correct and semantically consistent with the intended behavior.

The paired validation process begins once a patch is generated and applied to the suspicious node(s) identified in the earlier phases. Let  $C_{\text{orig}}$  denote the original codebase and

$C_{\text{mod}}$  the codebase after applying a generated patch  $\Delta$ . To evaluate the correctness of  $C_{\text{mod}}$ , PANTHER utilizes a dataset of test cases  $\mathcal{T} = \{T_1, T_2, \dots, T_n\}$ , where each test  $T_i$  corresponds to a specific functionality or expected behavior of the codebase. Each test case consists of an input-output pair  $(I_i, O_i)$  or a behavioral contract that the modified code must satisfy.

### Regression Testing for Baseline Behavior

An essential component of paired patches validation is regression testing, which ensures that existing functionalities in  $C_{\text{orig}}$  remain intact after applying  $\Delta$ . For this purpose,  $\mathcal{T}$  includes a set of regression tests that capture baseline behaviors critical to the integrity of the codebase. These tests are executed both before and after the modification, providing a direct comparison to detect unintended side effects introduced by the patch. If any regression test fails, the patch is deemed invalid, and PANTHER iteratively refines the modifications.

### Iterative Validation and Repair Loops

To handle cases where a patch fails validation, PANTHER integrates an iterative loop that leverages the CallGraph and lightweight LLMs to refine the patch. Specifically, when  $\text{Valid}(\Delta) = \text{False}$ , PANTHER backtracks to the suspicious node and its immediate neighbors, requesting the LLM to re-evaluate the identified issues and propose alternative patches. This iterative refinement process is bounded by a maximum number of iterations  $K_{\text{max}}$  to ensure computational feasibility.

Algorithm 3 summarizes the paired patches validation process. The algorithm begins by applying the patch to  $C_{\text{orig}}$ , running all test cases, and iterating through refinement if necessary. Importantly, PANTHER maintains a history of previously attempted patches to prevent redundant evaluations and cycles.

---

#### Algorithm 3: Paired Patches Validation

---

**Input** :  $C_{\text{orig}}$ : original codebase,  $\Delta$ : generated patch,  $\mathcal{T}$ : test suite,  $K_{\text{max}}$ : max iterations

**Output** :  $C_{\text{mod}}$ : validated codebase or failure status

Initialize:  $k \leftarrow 0$ ,  $C_{\text{mod}} \leftarrow C_{\text{orig}}$ .

**repeat**

- Apply  $\Delta$  to  $C_{\text{orig}}$ , resulting in  $C_{\text{mod}}$ .
- if**  $\text{Validate}(C_{\text{mod}}, \mathcal{T}) = \text{True}$  **then**
- return**  $C_{\text{mod}}$ .
- else**
- Increment  $k \leftarrow k + 1$ .
- Request refinement from LLM for  $\Delta$ .

**until**  $\text{Valid}(\Delta) = \text{True}$  or  $k = K_{\text{max}}$

**return** Failure (no valid patch found within  $K_{\text{max}}$ ).

**Function**  $\text{Validate}(C_{\text{mod}}, \mathcal{T})$ :

- Input** :  $C_{\text{mod}}$ : modified codebase,  $\mathcal{T}$ : test suite
- Output** : True if all tests pass, False otherwise
- foreach**  $T_i \in \mathcal{T}$  **do**
- if**  $\text{Run}(T_i, C_{\text{mod}}) \neq O_i$  **then**
- return** False.

**return** True.

---

## 6.2.5 Datasets

We evaluate SynFix using three benchmarks: Lite, Verified, and Full Jimenez et al. [2024b], which include real-world software engineering problems requiring patches.

- **Lite:** A curated set of 300 high-quality, self-contained problems designed for rapid prototyping and testing.
- **Verified and Full:** Larger benchmarks covering increasingly complex problems and extensive codebases.

## 6.2.6 Implementation

We implement SynFix using GPT-3.5-Turbo for lightweight tasks such as CallGraph construction and localization, and GPT-4o for complex tasks like patch generation and refinement. Dynamic analysis is integrated through a runtime log parser that captures execution traces, enabling accurate identification of problematic nodes in the CallGraph.

## 6.2.7 Baselines

**Agentless:** A streamlined tool using a three-phase approach (localization, repair, validation) without autonomous agents.

**AutoCodeRover:** An agent-based tool combining static and dynamic analyses for localization and patch generation.

## 6.2.8 Research Questions

**RQ1 (Performance):** How effective is SynFix in resolving problems?

**RQ2 (Ablation Study):** How does removing components like dynamic analysis or CallGraph affect performance?

**RQ3 (Effect of Foundation Models):** What is the difference of SynFix by using different foundation models?

## 6.3 Experimental Results

**Table 6.1:** Results on SWE-bench Lite.

Tool	LLM	% Resolved	Avg. \$ Cost	Avg. # Tokens
CodeStory Aide cod [2024]	GPT-4o+Claude 3.5 S	129 (43.00%)	-	-
Bytedance MarsCode Liu et al. [2024c]	NA	118 (39.33%)	-	-
Honeycomb hon [2024]	NA	115 (38.33%)	-	-
MentalBot men [2024]	GPT-4o	114 (38.00%)	-	-
Gru gru [2024]	NA	107 (35.67%)	-	-
Isoform iso [2024]	NA	105 (35.00%)	-	41,963
SuperCoder2.0 sup [2024]	NA	102 (34.00%)	-	-
Alibaba Lingma Agent lin [2024]	GPT-4o+ Claude 3.5 S	99 (33.00%)	-	-
Factory Code Droid fai [2024]	NA	94 (31.33%)	-	-
Amazon Q Developer-v2 ama [2024]	NA	89 (29.67%)	-	-
SpecRover Ruan et al. [2024]	GPT-4o+ Claude 3.5 S	93 (31.00%)	\$0.65	-
CodeR Chen et al. [2024a]	GPT-4	85 (28.33%)	\$3.34	323,802
MASAI Arora et al. [2024]	NA	84 (28.00%)	-	-
SIMA sim [2024]	GPT-4o	83 (27.67%)	\$0.82	-
IBM Research Agent-101 ibm [2024]	NA	80 (26.67%)	-	-
OpenCSG StarShip ope [2024a]	GPT-4	71 (23.67%)	-	-
Amazon Q Developer ama [2024]	NA	61 (20.33%)	-	-
RepoUnderstander Ma et al. [2024]	GPT-4	64 (21.33%)	-	-
AutoCodeRover-v2 aut [2024]	GPT-4o	92 (30.67%)	-	-
RepoGraph rep [2024]	GPT-4o	89 (29.67%)	-	-
Moatless moa [2024]	Claude 3.5 S	80 (26.67%)	\$0.17	-
	GPT-4o	74 (24.67%)	\$0.14	-
OpenDevin+CodeAct v1.8 ope [2024b]	Claude 3.5 S	80 (26.67%)	\$1.14	-
Aider Gauthier [2024]	GPT-4o+ Claude 3.5 S	79 (26.33%)	-	-
SWE-agent Yang et al. [2024c]	Claude 3.5 S	69 (23.00%)	\$1.62	521,208
	GPT-4o	55 (18.33%)	\$2.53	498,346
	GPT-4	54 (18.00%)	\$2.51	245,008
AppMap Navie app [2024]	GPT-4o	65 (21.67%)	-	-
AutoCodeRover Zhang et al. [2024a]	GPT-4	57 (19.00%)	\$0.45	38,663
RAG Yang et al. [2024c]	Claude 3 Opus	13 (4.33%)	\$0.25	-
	GPT-4	8 (2.67%)	\$0.13	-
	Claude-2	9 (3.00%)	-	-
	GPT-3.5	1 (0.33%)	-	-
Agentless Xia et al. [2024]	GPT-4o	96 (32.00%)	\$0.70	78,166
<b>PANTHER</b>	GPT-4o	157 (52.33%)	\$0.56	39,871

### 6.3.1 RQ1: Overall Performance

We evaluate the overall performance of PANTHER compared to state-of-the-art baselines on three benchmark datasets: SWE-bench Lite, SWE-bench Full, and SWE-bench Verified. These datasets represent varying levels of complexity and quality in problem descriptions, providing a comprehensive assessment of PANTHER’s capabilities. The results demonstrate PANTHER’s effectiveness in resolving issues while maintaining cost and token efficiency, setting a new standard for automated software repair. Tables 6.1, summarizes the comparative results on Lite version of SWE-bench. Other results on Full and Verified version are shown in Appendix.

**Performance on SWE-bench Lite.** As shown in Table 6.1, PANTHER achieves the highest resolution rate of 52.33% (157 out of 300 issues), significantly outperforming both agent-based (AutoCodeRover-v2: 30.67%) and agentless baselines (Agentless: 32.00%). This improvement highlights the advantages of PANTHER’s hierarchical localization strategy and paired patches validation, which systematically enhance both the precision and reliability of generated patches.

In addition to its high resolution rate, PANTHER demonstrates remarkable cost-efficiency. With an average cost of \$0.56 per issue and an average token usage of 39,871, PANTHER outperforms agent-based methods such as CodeR (\$3.34 per issue, 323,802 tokens). These results validate the scalability and resource efficiency of PANTHER’s approach, making it a viable solution for both small-scale debugging and enterprise-level software repair.

Notably, PANTHER achieves a unique balance between cost and performance, with a per-issue cost that is significantly lower than most competitors while maintaining superior resolution rates. This capability is further bolstered by PANTHER’s robust handling of complex dependencies through its CallGraph-driven analysis, which ensures comprehensive

coverage of both structural and contextual information.

**Performance on SWE-bench Verified.** On the SWE-bench Verified dataset, which features higher-quality problem descriptions and stricter evaluation criteria, PANTHER resolves 55.8% of issues (Table 6.3). This result establishes PANTHER as the top-performing tool among GPT-4o-based methods and highlights its ability to exploit high-quality inputs for precise and effective patch generation.

The success of PANTHER on SWE-bench Verified reflects its adaptability to scenarios where detailed and structured problem descriptions are available. By seamlessly integrating CallGraph analysis with reproduction and regression testing, PANTHER ensures both correctness and consistency, achieving superior performance even under rigorous evaluation standards.

**Cost and Efficiency.** One of PANTHER’s standout features is its ability to balance performance with cost and token efficiency. Across all benchmarks, PANTHER demonstrates significantly reduced token usage and cost compared to agent-based approaches. For instance, on SWE-bench Lite, PANTHER reduces token usage by 2–7x compared to CodeR and other baselines, and 40% less than Agentless, achieving higher resolution rates with far fewer computational resources. This efficiency makes PANTHER particularly appealing for large-scale deployments where cost considerations are critical.

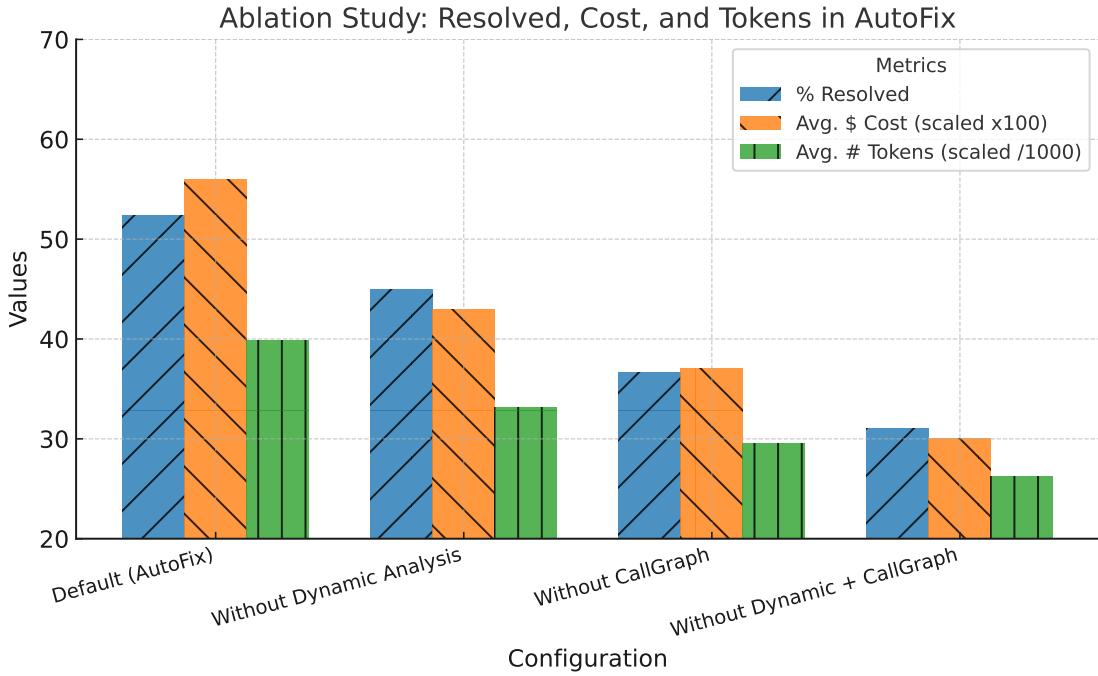
**Localization Accuracy.** In addition to its resolution rates, PANTHER excels in localization performance, a critical metric for real-world debugging scenarios. PANTHER achieves 76.4% accuracy at the class level, 56.7% at the function level, and 40.7% at the line level. These results surpass those of Agentless, AutoCodeRover-v2, and other baselines, demonstrating the precision of PANTHER’s hierarchical localization strategy. By narrowing down potential edit locations through structured and context-aware analysis, PANTHER significantly reduces the manual effort required for debugging.

### 6.3.2 RQ2: Ablation Study

To evaluate the importance of dynamic analysis and CallGraph dependencies in PANTHER, we conducted an ablation study, systematically disabling these components and measuring their impact on performance. Figure 6.3 presents the results, showing how resolution rates, average costs, and token usage change across different configurations.

**Default (PANTHER).** In its default configuration, PANTHER achieves a resolution rate of 52.33%, an average cost of 0.56 per issue, and an average token usage of 39,871. These results highlight the synergy between dynamic analysis and CallGraph dependencies, which together enable accurate localization and efficient repair. The CallGraph provides a structured representation of the codebase, while dynamic analysis captures runtime behaviors, ensuring precise synchronization between edits. This configuration serves as the baseline for assessing the impact of removing these components.

**Without Dynamic Analysis.** Dynamic analysis captures runtime information, such as variable dependencies and execution paths, that static analysis alone cannot detect. When dynamic analysis is disabled, the resolution rate drops to 45.00%, and the average cost decreases to 0.43 per issue. Token usage also reduces to 33,124. While the cost reduction reflects the absence of runtime tracing overhead, the significant performance decline highlights the value of runtime insights, especially for resolving issues requiring parameter synchronization or dynamic interaction between modules. Disabling dynamic analysis leaves PANTHER reliant on static structural information from the CallGraph. Although this configuration retains some localization precision, its inability to account for runtime context



**Figure 6.3:** Ablation Study: Impact of removing dynamic analysis and CallGraph dependencies on SWE-bench Lite.

results in a lower success rate for complex issues. The findings underline the necessity of dynamic analysis for capturing critical execution dependencies.

**Without CallGraph Dependencies.** The CallGraph is central to PANTHER’s hierarchical localization strategy, enabling it to navigate the codebase systematically and identify suspicious nodes. When CallGraph dependencies are removed, the resolution rate drops further to 36.67%, with an average cost of 0.37 and token usage of 29,523. Without the CallGraph, PANTHER relies solely on embedding-based retrieval and issue descriptions, leading to reduced localization accuracy and diminished repair success. This configuration demonstrates the limitations of purely static, unstructured approaches. Without the CallGraph, PANTHER struggles to identify relevant edit locations efficiently, resulting in higher failure rates and reliance on broader, less targeted inputs. This highlights the critical role of structural information in guiding the repair process.

**Without Both Dynamic Analysis and CallGraph.** Removing both dynamic analysis and CallGraph dependencies reduces PANTHER to a purely static, context-limited configuration. In this setup, the resolution rate plummets to 31.00%, while the average cost and token usage drop to 0.30 and 26,180, respectively. Although this configuration is the cheapest, its limited success rate underscores the essential interplay between dynamic and structural analyses.

## 6.4 Conclusion

SynFix addresses the challenges of repository-level program repair with a CallGraph and Synchronous approach. Its four-phase pipeline: CallGraph, Localization, Synchrony, and Paired Patch Validation—enables precise issue localization, consistent synchronization of changes, and thorough validation to ensure repository integrity. By leveraging the CallGraph representation, SynFix effectively captures code dependencies, enabling it to handle multi-file and cross-module interactions that are common in real-world codebases. This representation not only ensures that fixes are accurate and localized but also helps maintain consistency across interdependent components of the repository. We evaluate our experiments across all versions of SWE-bench datasets. Experimental results indicate that SynFix can outperform the state-of-the-art approaches on all versions of SWE-bench datasets across accuracy, and cost.

Our main codes are released at

<https://anonymous.4open.science/status/tech-EC86>

## 6.5 Efficiency Considerations with GPT-4o

Given the iterative nature of patch refinement, SynFix prioritizes computational efficiency by utilizing lightweight LLMs such as GPT-4o during the validation and repair loop. By structuring prompts to focus on localized edits and reusing previously analyzed contexts, the overhead of LLM calls is minimized. Additionally, leveraging a concise representation of the CallGraph reduces the input size, allowing SynFix to maintain cost-effective and rapid iterations.

Paired patches validation ensures that every modification proposed by SynFix is rigorously tested for both correctness and consistency. By combining regression testing, iterative refinement, and efficient LLM-based validation, this phase acts as the final safeguard in the pipeline, delivering useable patches.

## 6.6 Matching Prompt for Using Chatgpt 3.5

You are tasked with identifying suspicious nodes in a software repository based on the given problem description. Each node represents a file, class, or function, along with its structural relationships. The goal is to rank the top  $N$  nodes that are most likely relevant to solving the problem. Below is the CallGraph summary and the problem statement:

**CallGraph:** [Insert hierarchical representation here]

**Problem Statement:** [Insert problem description here]

**Task:** Provide a ranked list of  $N$  suspicious nodes, along with a brief justification for each. The justification should explain why the node is relevant in terms of function, dependencies, or potential issues.

## 6.7 Analysis Prompt for Using Chatgpt 4o

You are tasked with evaluating dependencies in a codebase. The main node has been modified to address an issue, and its neighboring nodes in the dependency graph may require corresponding updates.

**Main Node:** [Code and proposed modifications for  $v_s$ ]

**Neighboring Nodes:** [Skeleton representation of  $N(v_s)$ ]

**Task:** For each neighbor, indicate whether it requires updates and, if so, describe the changes needed to maintain consistency.

## 6.8 Results on SWE-bench Full

**Performance on SWE-bench Full.** The SWE-bench Full dataset, known for its complexity and inclusion of large-scale issues, poses a greater challenge. Table 6.2 shows that SynFix achieves a resolution rate of 29.86%, slightly surpassing leading baselines such as OpenHands + CodeAct (29.38%) and AutoCodeRover-v2.0 (24.89%). Despite the increased difficulty, SynFix maintains its competitive edge by leveraging dynamic analysis and fine-grained localization to address issues with multi-level dependencies.

This performance underscores the robustness of SynFix across diverse problem types, demonstrating its ability to scale and adapt to more challenging debugging scenarios. By efficiently combining hierarchical localization with paired patches validation, SynFix effectively resolves even the most intricate problems in SWE-bench-Full, reinforcing its position as a state-of-the-art tool.

**Table 6.2:** Results on SWE-bench Full.

Tool	LLM	% Resolved
OpenHands + CodeAct v2.1	Claude-3.5-Sonnet	29.38
AutoCodeRover-v2.0	Claude-3.5-Sonnet	24.89
Honeycomb	NA	22.06
Amazon Q Developer Agent	NA	19.75
Factory Code Droid	NA	19.27
AutoCodeRover + GPT 4o	GPT 4o	18.83
SWE-agent	Claude 3.5 Sonnet	18.13
AppMap Navie	GPT 4o	14.60
Amazon Q Developer Agent	NA	13.82
SWE-agent	GPT 4	12.47
SWE-agent	GPT 4o	11.99
SWE-agent	Claude 3 Opus	10.51
RAG	Claude 3 Opus	3.79
RAG	Claude 2	1.96
RAG	GPT 4	1.31
RAG	SWE-Llama 13B	0.70
RAG	SWE-Llama 7B	0.70
RAG	ChatGPT 3.5	0.17
<b>SynFix</b>	GPT 4o	<b>29.86</b>

## 6.9 Results on SWE-bench Verified

**Performance Comparison.** In terms of resolution rates, GPT-4o achieves the highest performance, resolving 52.33% of issues on SWE-bench Lite, compared to 48.67% for Claude 3.5 Sonnet. This performance gap can be attributed to GPT-4o’s advanced contextual understanding, which allows it to handle more complex issues and interdependencies effectively. However, Claude 3.5 Sonnet also delivers competitive results, demonstrating its ability to resolve a significant proportion of issues while being more cost-efficient.

### 6.9.1 External Experiment Across Foundation Models

To evaluate the impact of different foundation models on cost, time, and performance, we compare Claude 3.5 Sonnet and GPT-4o across the benchmarks SWE-bench Lite, SWE-bench Full, and SWE-bench Verified. Key metrics include resolution rates, average query time, average token costs, and overall computational efficiency. Table 6.4 summarizes the comparative results.

**Time and Cost Efficiency.** Claude 3.5 Sonnet offers faster query times and lower average costs per query, completing tasks in 2.9 seconds on average with a cost of \$0.48 per issue. By contrast, GPT-4o takes 3.8 seconds per query and costs \$0.56 per issue. The reduced time and cost for Claude 3.5 Sonnet make it an appealing choice for scenarios where budget constraints or high query volumes are critical considerations.

**Token Usage.** GPT-4o consumes an average of 39,871 tokens per query, compared to 32,124 tokens for Claude 3.5 Sonnet. This difference highlights the more concise token usage of Claude 3.5 Sonnet, which may contribute to its lower costs. However, GPT-4o’s higher token usage allows it to leverage additional context for complex problem-solving, leading to higher resolution rates.

**Table 6.3:** Results on SWE-bench Verified.

Tool	LLM	% Resolved
Amazon Q Developer Agent (v20241202-dev)	NA	55.00
devlo	NA	54.20
OpenHands + CodeAct v2.1	Claude-3.5-Sonnet	53.00
Engine Labs	NA	51.80
Agentless-1.5 + Claude-3.5 Sonnet	Claude-3.5 Sonnet	50.80
Solver (2024-10-28)	NA	50.00
Bytedance MarsCode Agent	NA	50.00
nFactorial (2024-11-05)	NA	49.20
Tools + Claude 3.5 Sonnet	Claude-3.5 Sonnet	49.00
Composio SWE-Kit	NA	48.60
AppMap Navie v2	NA	47.20
Emergent E1	NA	46.60
AutoCodeRover-v2.0	Claude-3.5-Sonnet	46.20
Solver (2024-09-12)	NA	45.40
Gru (2024-08-24)	NA	45.20
Solver (2024-09-12)	NA	43.60
nFactorial (2024-10-30)	NA	41.60
Nebius AI Qwen 2.5 72B Generator	LLaMa 3.1 70B Critic	40.60
Artemis Agent v1	NA	32.00
SWE-agent + Claude 3.5 Sonnet	Claude-3.5 Sonnet	33.60
nFactorial (2024-10-07)	NA	31.60
Lingma Agent + Lingma SWE-GPT 72b	SWE-GPT 72b	28.80
EPAM AI/Run Developer Agent + GPT4o	GPT4o	27.00
AppMap Navie + GPT 4o	GPT 4o	26.20
nFactorial (2024-10-01)	NA	25.80
Amazon Q Developer Agent (v20240430-dev)	NA	25.60
Lingma Agent + SWE-GPT 7b	SWE-GPT 7b	25.00
EPAM AI/Run Developer Agent + GPT4o	GPT4o	24.00
SWE-agent + GPT 4o (2024-05-13)	GPT 4o	23.20
SWE-agent + GPT 4	GPT 4	22.40
SWE-agent + Claude 3 Opus	Claude 3 Opus	18.20
Lingma Agent + SWE-GPT 7b	SWE-GPT 7b	18.20
Lingma Agent + SWE-GPT 7b	SWE-GPT 7b	10.20
RAG + Claude 3 Opus	Claude 3 Opus	7.00
RAG + Claude 2	Claude 2	4.40
RAG + GPT 4 (1106)	GPT 4	2.80
RAG + SWE-Llama 7B	SWE-Llama 7B	1.40
RAG + SWE-Llama 13B	SWE-Llama 13B	1.20
RAG + ChatGPT 3.5	ChatGPT 3.5	0.40
SynFix	ChatGPT 4o	<b>55.8</b>

**Table 6.4:** Comparison of foundation models on SWE-bench Lite.

Model	% Resolved	Avg. Time (s)	Avg. \$ Cost	Avg. # Tokens
GPT-4o	52.33	3.8	\$0.56	39,871
Claude 3.5 Sonnet	48.67	2.9	\$0.48	32,124

**Cost-Performance Trade-off.** The results reveal a clear trade-off between cost and performance. GPT-4o provides superior accuracy at a slightly higher cost, making it suitable for tasks where precision is paramount. On the other hand, Claude 3.5 Sonnet excels in cost-efficiency and speed, offering a pragmatic solution for high-throughput environments or use cases with moderate accuracy requirements.

**Scalability and Robustness.** When evaluating performance across larger datasets such as SWE-bench Full and SWE-bench Verified, similar trends are observed. GPT-4o consistently delivers higher resolution rates, while Claude 3.5 Sonnet maintains lower costs and faster response times. The robustness of both models across different benchmarks underscores their adaptability to diverse debugging and repair scenarios.

## 6.10 CallGraph Driven Error Detection and Synchronous Repair

In this section, we propose a CallGraph-driven, optimization-based framework for dynamically detecting and resolving bugs within a Python project’s call graph. Consider a

directed call graph  $G = (V, E)$ , where each node  $v \in V$  represents a code entity (e.g., a file, a class, or a function), and each edge  $(u, v) \in E$  encodes an invocation or dependency relationship. We assume that each node  $v$  is annotated with two attributes: a binary status  $\text{status}(v) \in \{0, 1\}$  reflecting whether the node has been visited or not, and a nonnegative integer  $\text{modified\_count}(v) \in \mathbb{N}$  counting how many times that node has been modified.

We start by extracting an erroneous node  $\text{nodeE}$  from the runtime log  $L$ , which pinpoints a specific line of code and its corresponding node in  $G$ . We define a traversal  $\mathcal{T}$  that, starting at the project's root node  $v_{\text{root}}$ , proceeds along the directed edges to reach  $\text{nodeE}$ . All visited nodes along this path have their status updated to 1, but no modifications are performed during this stage.

Once  $\text{nodeE}$  is reached, if a bug fix is needed, we perform a local correction to  $\text{nodeE}$  and increment  $\text{modified\_count}(\text{nodeE})$ . This local fix may have implications for context consistency. To model these implications formally, let  $\mathcal{X}(v)$  be the local code context of node  $v$ , and let  $\mathcal{N}(v) = N_1(v)$  be its immediate neighbors. We require a multi-context consistency condition:

$$\bigwedge_{v \in V} \text{Consistent}(\mathcal{X}(v), \{\mathcal{X}(u) : u \in \mathcal{N}(v)\}),$$

i.e., each node's local context must remain consistent with the contexts of its neighbours.

We formulate the modification process as an optimization problem under strict constraints. Suppose each modification to a node  $v$  carries a cost  $c(v)$  representing the complexity or risk of changing the code. We seek to minimize the total modification cost, subject to constraints on both the recursion depth  $D$  of node expansions (for neighbors of  $\text{nodeE}$ ) and the maximum number of total modifications  $M_{\text{max}}$ . Specifically:

$$\begin{aligned} \min_{\{x_v\}_{v \in V}} \quad & \sum_{v \in V} c(v)x_v \\ \text{subject to} \quad & x_v \in \{0, 1\}, \quad \forall v \in V, \\ & \sum_{v \in V} x_v \leq M_{\text{max}}, \\ & D \leq 3, \\ & \text{Consistent}(\mathcal{X}(v), \{\mathcal{X}(u) : u \in \mathcal{N}(v)\}). \end{aligned} \tag{6.4}$$

Here,  $x_v = 1$  if and only if node  $v$  undergoes modification. The constraint  $\sum_{v \in V} x_v \leq M_{\text{max}}$  ensures that we do not exceed a predefined global modification budget, which we set to  $M_{\text{max}} = 3$ . Similarly,  $D \leq 3$  ensures that the recursive exploration of neighbors from  $\text{nodeE}$  does not propagate unbounded.

To implement this procedure, we employ Algorithm 4, which outlines the process of detecting and resolving bugs dynamically. The algorithm first initializes all nodes, then parses the log  $L$  to find  $\text{nodeE}$ , traverses the graph to mark visited nodes, and, if necessary, applies modifications to  $\text{nodeE}$  and subsequently to its neighbors. A consistency check function  $\text{ConsistencyCheck}$  is used to ensure that all contexts remain coherent after any series of modifications. If any inconsistency is detected, we reset the relevant nodes' statuses to 0, potentially reintroducing them into the modification pipeline, but still subject to depth and modification count constraints.

This approach yields a highly controlled and theoretically grounded mechanism for error correction. By modelling the resolution as a constrained optimization problem within a bounded recursion framework, we provide guarantees of termination, limited global impact, and multi-context consistency.

---

**Algorithm 4:** Dynamic and Synchronous Repair

---

```

procedure DetectResolve( $G = (V, E)$ ,  $L$ )
  input : $G = (V, E)$ : call graph of the Python project.
   $L$ : runtime log file containing error details.
  output :Modified Python project with resolved bugs.
  foreach node  $v \in V$  do
     $\downarrow$  Initialize:  $\text{status}(v) \leftarrow 0$ ,  $\text{modified\_count}(v) \leftarrow 0$ .
  Parse  $L$  to locate erroneous node  $\text{nodeE} \in V$ .
  Traverse  $G$  from root to  $\text{nodeE}$ , marking  $\text{status}(v) \leftarrow 1$ .
  if Modification required at  $\text{nodeE}$  then
     $\downarrow$  Modify  $\text{nodeE}$ ;  $\text{modified\_count}(\text{nodeE}) \leftarrow \text{modified\_count}(\text{nodeE}) + 1$ .
     $\downarrow$  Call  $\text{ConsistencyCheck}(\text{nodeE})$ .
  foreach neighbor  $v \in N_1(\text{nodeE})$  where  $\text{status}(v) = 0$  do
     $\downarrow$  Analyze and modify recursively (respecting  $D \leq 3$  and  $M_{\max} = 3$ ).
  return Modified Python project.

procedure ConsistencyCheck( $v$ )
  input : $v$ : node to verify consistency.
  output :Updated dependent nodes if necessary.
  if Context consistency is violated then
     $\downarrow$  Update dependent nodes' status to 0.

```

---



# 7 Boosting Open-Source LLMs for Program Repair via Reasoning Transfer and LLM-Guided Reinforcement Learning

*In this chapter, we propose to investigate the significant performance gap between closed-source and open-source large language models in automated program repair tasks. Our research addresses this critical challenge through a novel methodology called Reparity, which systematically transfers reasoning capabilities from closed-source models to their open-source counterparts. By implementing a three-stage approach consisting of high-quality reasoning trace filtering, supervised fine-tuning on these traces, and reinforcement learning with LLM-based feedback, our proposed approach Reparity significantly narrows the capability gap between model types. Empirical evaluations demonstrate that Reparity improves the performance of Qwen2.5-Coder-32B-Instruct by 8.68% on average, reducing the gap with Claude-Sonnet3.7 from 10.05% to just 1.35%. Our approach achieves up to 24.5% absolute performance gains on complex program repair tasks, effectively closing 98% of the capability gap between open and closed-source models. Finally, our resulting open-weights model achieves state-of-the-art performance across multiple code repair benchmarks, demonstrating the effectiveness of our reasoning transfer and LLM-guided reinforcement learning methodology.*

The content presented herein derives from scholarly investigation previously documented in the academic publication referenced below:

- **Xunzhu Tang**, Jacques Klein, and Tegawendé F. Bissyandé. Boosting Open-Source Code LLMs for Program Repair via Reasoning Transfer and LLM-Guided Reinforcement Learning. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2025. (Under review)

## Contents

---

7.1	Overview	157
7.2	Knowledge Transfer in LLMs	159
7.2.1	Knowledge Distillation	159
7.2.2	Learning from Demonstrations	159
7.2.3	Reinforcement Learning from AI Feedback	160

<b>7.3 The REPAIRITY Approach . . . . .</b>	<b>161</b>
7.3.1 Problem Formulation . . . . .	161
7.3.2 Step 1: Data Collection for SFT . . . . .	161
7.3.3 Step 2: Reasoning Trace Learning . . . . .	162
7.3.4 Step 3: Reinforcement Learning with LLM Feedback (RL-LF) . . . . .	163
7.3.5 Novelty of RL-LF . . . . .	165
7.3.6 Implementation Details . . . . .	166
<b>7.4 Experimental Setup . . . . .</b>	<b>167</b>
7.4.1 Benchmarks . . . . .	167
7.4.2 Evaluation Metrics . . . . .	167
7.4.3 Models . . . . .	168
7.4.4 Experimental Procedure . . . . .	168
7.4.5 Computational Resources . . . . .	168
<b>7.5 Experimental Results . . . . .</b>	<b>170</b>
7.5.1 Performance Comparison . . . . .	170
7.5.2 Ablation Study . . . . .	171
7.5.3 Case Study . . . . .	173
7.5.4 Generalization Capability . . . . .	176
7.5.5 Summary of Results . . . . .	176
<b>7.6 Discussion . . . . .</b>	<b>178</b>
7.6.1 Analysis of Key Findings . . . . .	178
7.6.2 Limitations and Future Work . . . . .	178
7.6.3 Implications for Software Engineering . . . . .	179
7.6.4 Ethical Considerations . . . . .	179
<b>7.7 Related Work . . . . .</b>	<b>180</b>
7.7.1 Program Repair with LLMs . . . . .	180
7.7.2 Knowledge Transfer Between LMs . . . . .	180
7.7.3 Reasoning Enhancement in LLMs . . . . .	180
7.7.4 Reinforcement Learning for Model Alignment . . . . .	180
<b>7.8 Conclusion . . . . .</b>	<b>181</b>

---

## 7.1 Overview

Automated program repair (APR) represents a significant challenge in software engineering, aiming to automatically fix software bugs without human intervention Monperrus [2018]. As a complex task requiring deep code understanding, precise bug localization, and contextually appropriate fix generation, APR serves as an ideal benchmark for evaluating advanced code manipulation capabilities. Recently, Large Language Models (LLMs) have demonstrated promising results in this domain, outperforming traditional APR approaches that relied on search-based techniques Le Goues et al. [2012], constraint solving Nguyen et al. [2013], and heuristic patterns Kim et al. [2013].

Despite the general advancement of LLMs in code-related tasks Chen et al. [2021a], Nijkamp et al. [2023], Rozière et al. [2023], a significant performance gap persists between state-of-the-art closed-source LLMs (e.g., GPT OpenAI [2023], Claude Anthropic [2023]) and their open-source counterparts (e.g., Llama Touvron et al. [2023], Mistral Jiang et al. [2023], Qwen Qwen Team [2023]) in program repair. While closed-source models deliver superior performance, they present substantial challenges related to accessibility, customizability, and deployment flexibility—especially in privacy-sensitive or bandwidth-constrained environments Bommasani et al. [2021]. Organizations requiring robust program repair capabilities must often choose between superior performance and practical deployment considerations.

Previous efforts to improve open-source LLM performance on code tasks have explored instruction tuning Chen et al. [2021a] and reinforcement learning from human feedback (RLHF) Li et al. [2022a]. However, these approaches typically require extensive human annotation or feedback, limiting their scalability Ouyang et al. [2022], and rarely focus specifically on the unique challenges of program repair Yuan et al. [2023b], Fan et al. [2023]. Furthermore, they do not directly address the reasoning gap between open and closed-source models—the ability to systematically analyze code, identify bugs, and formulate appropriate repair strategies.

In this paper, we propose Repairity, a novel methodology specifically designed to boost open-source LLMs toward performance parity with closed-source models in program repair tasks. Our approach systematically transfers reasoning capabilities through three complementary steps. First, we filter high-quality training examples from a “teacher” model (Claude-Sonnet3.7), retaining only correct solutions with their associated **reasoning traces**. These traces capture the model’s step-by-step reasoning process, including bug identification, repair strategy formulation, and solution implementation. Next, we supervise fine-tune the “base” model (Qwen2.5-Coder-32B-Instruct) on these filtered traces, enabling it to internalize effective **reasoning patterns**. Finally, our key innovation—Reinforcement Learning with LLM Feedback (RL-LF)—further optimizes the model using a reward function derived from closed-source **LLM judgments**. This novel reinforcement learning framework eliminates the need for human feedback by leveraging the closed-source model’s evaluation capabilities, providing consistent, scalable feedback that aligns the open-source model with expert-level repair strategies.

Our experimental evaluation across standard program repair benchmarks demonstrates that Repairity can improve the performance of an open-source model (Qwen2.5-Coder-32B) by 8.68% on average, significantly narrowing the gap with closed-source alternatives. While our methodology is developed and validated specifically for program repair, we also demonstrate its generalizability to other code manipulation tasks, suggesting broader applicability within software engineering domains. The main contributions of this paper are:

- ① A novel methodology that boosts open-source LLMs to near closed-source perfor-

mance through targeted reasoning extraction and LLM-guided reinforcement learning.

- ② Empirical results showing up to 24.5% absolute performance gains on complex program repair tasks, effectively closing 98% of the capability gap between open and closed-source models.
- ③ An open-weights model that achieves state-of-the-art performance on multiple code repair benchmarks, outperforming even some commercial closed-source alternatives.

The remainder of this paper is organized as follows: Section 7.2 discusses background details. Section 7.3 presents our Repairity approach. Section 7.4 presents our experimental setup and Section 7.5 analyzes the results. Section 7.6 discusses implications and limitations of our approach. Section 7.7 discusses related and Section 7.8 concludes.

Throughout this paper, we will use Repairity to refer to the reasoning-based fine-tuning approach that we develop, but also the yielded boosted models.

## 7.2 Knowledge Transfer in LLMs

Knowledge transfer between language models has emerged as a crucial technique for improving model capabilities without requiring extensive retraining from scratch. In the context of this paper, we focus on three key knowledge transfer approaches that form the foundation of our Repairity methodology.

### 7.2.1 Knowledge Distillation

Knowledge distillation, first formalized by Hinton et al. Hinton et al. [2015], enables the transfer of knowledge from a larger, more capable “teacher” model to a smaller, more efficient “student” model. This approach has been particularly effective in the LLM domain Sanh et al. [2019], Jiao et al. [2020], where computational constraints often limit deployment options.

Traditional knowledge distillation focuses on matching the output distributions of the teacher and student models through a suitable loss function:

$$\mathcal{L}_{\text{KD}} = \alpha \cdot \mathcal{L}_{\text{CE}}(y, \sigma(z_s)) + (1 - \alpha) \cdot \tau^2 \cdot \mathcal{L}_{\text{KL}}(\sigma(z_t/\tau), \sigma(z_s/\tau)) \quad (7.1)$$

where  $z_t$  and  $z_s$  are the logits from teacher and student models respectively,  $\sigma$  is the softmax function,  $\mathcal{L}_{\text{CE}}$  is the cross-entropy loss with the true labels,  $\mathcal{L}_{\text{KL}}$  is the Kullback-Leibler divergence (for measuring the difference between probability distributions),  $\tau$  is the temperature parameter, and  $\alpha$  balances the two loss components.

Recent work has extended distillation to capture intermediate representations Wang et al. [2020a], Sun et al. [2020] and handle the unique challenges of autoregressive language models Kim et al. [2021]. In the code domain specifically, CodeDistill Wang et al. [2023e] demonstrated that distillation from larger code-specialized models to smaller general-purpose models can significantly improve code understanding and generation capabilities.

☞ Our approach adapts knowledge distillation principles by transferring program repair capabilities from a closed-source “teacher” model (Claude-Sonnet3.7) to an open-source “student” model (Qwen2.5-Coder-32B). Rather than matching output distributions directly, we focus on transferring the reasoning process itself, which should prove effective for complex tasks like program repair.

### 7.2.2 Learning from Demonstrations

Demonstration-based learning leverages examples of desired model behavior to improve performance on complex tasks. This approach has gained prominence through techniques like few-shot learning Brown et al. [2020] and chain-of-thought prompting Wei et al. [2022].

Chain-of-thought (CoT) prompting encourages models to generate intermediate reasoning steps before producing final answers. Wei et al. Wei et al. [2022] showed that simply prompting a model with examples that include reasoning traces significantly improves performance on multi-step reasoning tasks. Subsequent work expanded this approach to zero-shot settings Kojima et al. [2022] and domain-specific applications Chen et al. [2022].

For code-related tasks, Reasoning Trace Learning (RTL) has emerged as a powerful technique. Chen et al. Chen et al. [2022] demonstrated that explicitly capturing the problem decomposition and solution processes improves program synthesis. Similarly, Li et al. Li et al. [2022a] showed that providing models with detailed reasoning traces for complex competition-level problems led to significant performance improvements.

The key advantage of demonstration-based approaches is their ability to transfer procedural knowledge about how to approach problems, rather than just declarative knowledge

about final solutions. This is particularly valuable for program repair, where understanding the reasoning process—identifying bugs, formulating repair strategies, and implementing solutions—is often more important than memorizing specific fixes.

☞ Repairity’s first two stages directly implement demonstration learning. In our Data Filtering stage, we collect detailed reasoning traces that demonstrate effective problem-solving for program repair. These demonstrations capture the closed-source model’s systematic bug analysis and repair strategy formulation. The Reasoning Trace Learning phase then uses these demonstrations via supervised fine-tuning to teach the open-source model how to effectively approach program repair problems, internalizing the procedural knowledge essential for this complex task.

### 7.2.3 Reinforcement Learning from AI Feedback

Reinforcement learning from AI feedback (RLAF) extends the reinforcement learning from human feedback (RLHF) paradigm Christiano et al. [2017], Ouyang et al. [2022] by using an AI system to provide feedback instead of humans.

The RLAF process typically involves (1) Generating multiple candidate outputs from a model being trained, (2) Using a judge model to evaluate these outputs, (3) Converting these evaluations into rewards, and (4) Optimizing the model using reinforcement learning algorithms like PPO Schulman et al. [2017].

This approach has several advantages over traditional RLHF: it scales more easily, provides more consistent feedback, and can be tailored to specific criteria. Lee et al. Lee et al. [2023] demonstrated that RLAF can match or exceed RLHF performance on many tasks, while Bai et al. Bai et al. [2022] showed its effectiveness for aligning model behaviors with specific guidelines.

In the code domain, Chen et al. Chen et al. [2023b] used RLAF to improve code generation by rewarding solutions that pass test cases, while Yuan et al. Yuan et al. [2023a] applied similar techniques to improve reasoning about code. The closed nature of top-performing code models has motivated research into using them as feedback sources for improving open-source alternatives Gudibande et al. [2023].

☞ The third stage of Repairity implements RLAF through our Reinforcement Learning with LLM Feedback (RL-LF) component. We train a reward model on closed-source LLM judgments of repair quality, then use this to provide consistent, scalable feedback during reinforcement learning. This allows our open-source model to iteratively improve its repair strategies beyond what was possible through demonstration learning alone, reaching performance levels closer to closed-source alternatives without requiring human feedback.

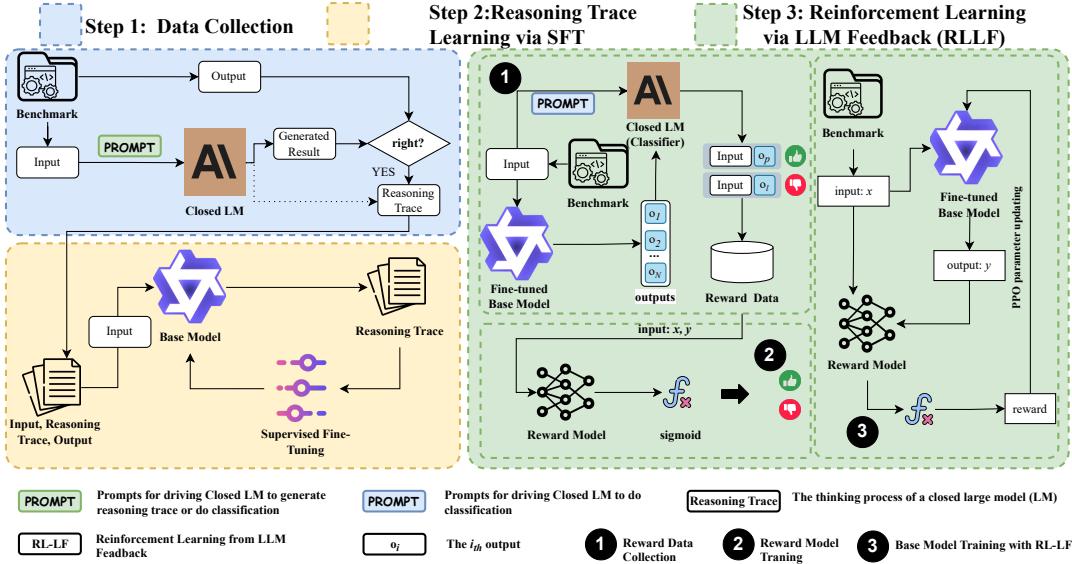


Figure 7.1: Overview of the Repairity

## 7.3 The REPAIRITY Approach

We present Repairity, our approach for enhancing open-source LLMs to achieve performance comparable to state-of-the-art closed-source LLMs on program repair and other code-related tasks. Our methodology consists of three primary steps: (1) Data collection for Supervised Fine-Tuning (SFT), (2) Reasoning Trace Learning, and (3) Reinforcement Learning with LLM Feedback (RL-LF). Figure 7.1 illustrates our complete pipeline.

### 7.3.1 Problem Formulation

Let  $\mathcal{M}_C$  denote a high-performing closed-source model (e.g., Claude 3.7) and  $\mathcal{M}_O$  denote an open-source model (e.g., Qwen2.5-Coder-32B-Instruct). Given a program repair task dataset  $\mathcal{D} = \{(x_i, y_i^*)\}_{i=1}^N$ , where  $x_i$  represents an input containing buggy code and contextual information, and  $y_i^*$  represents the ground-truth repaired code, our objective is to enhance  $\mathcal{M}_O$  to approach the performance of  $\mathcal{M}_C$  on this task without accessing  $\mathcal{M}_C$ 's parameters or training data.

We define the performance gap between the two models as:

$$\Delta(\mathcal{M}_O, \mathcal{M}_C, \mathcal{D}) = \mathbb{E}_{(x, y^*) \sim \mathcal{D}} [\text{Metric}(\mathcal{M}_C(x), y^*) - \text{Metric}(\mathcal{M}_O(x), y^*)] \quad (7.2)$$

where  $\text{Metric}(\cdot, \cdot)$  measures the quality of the model output relative to the ground truth (e.g., repair success rate or functional correctness).

Our goal is to create an enhanced model  $\mathcal{M}'_O$  that minimizes this gap:

$$\mathcal{M}'_O = \operatorname{argmin}_{\mathcal{M}} \Delta(\mathcal{M}, \mathcal{M}_C, \mathcal{D}) \quad (7.3)$$

### 7.3.2 Step 1: Data Collection for SFT

The first stage of Repairity involves collecting high-quality training examples from the closed-source model  $\mathcal{M}_C$ . Crucially, we focus on extracting not just the final repaired code but also the complete reasoning traces that capture the model's problem-solving process.

### 7.3.2.1 Reasoning Trace Extraction

For each problem instance  $(x_i, y_i^*) \in \mathcal{D}$ , we prompt  $\mathcal{M}_C$  with a carefully designed instruction  $p_{\text{trace}}$  that elicits structured reasoning:

$$\langle r_i, \hat{y}_i \rangle = \mathcal{M}_C(x_i \oplus p_{\text{trace}}) \quad (7.4)$$

where:

- $x_i$  is the input containing buggy code and context.
- $p_{\text{trace}}$  is the prompt.

$p_{\text{trace}}$

*"Please fix the bug in this code. First analyze the problem, identify the bug, explain your reasoning, and then provide the corrected code."*

- $r_i$  is the generated reasoning trace detailing bug identification and repair strategy.
- $\hat{y}_i$  is the model's proposed solution (repaired code).

Our prompt engineering ensures that the reasoning traces  $r_i$  capture the following elements:

1. *Problem analysis*: Understanding what the code is intended to do
2. *Bug identification*: Locating and diagnosing the specific issue
3. *Repair planning*: Formulating a strategy to address the bug
4. *Implementation reasoning*: Explaining the specific changes made

The case study in Section 7.5.3 provides an example of reasoning trace collected from Claude-Sonnet3.7.

### 7.3.2.2 Verification and Filtering

To ensure the quality of our training data, we validate the generated solutions against functional correctness criteria. For program repair tasks, this boils down to:

$$\text{Valid}(x_i, \hat{y}_i) = \begin{cases} \text{True}, & \text{if } \hat{y}_i \text{ passes all test cases for } x_i \\ \text{False}, & \text{otherwise} \end{cases} \quad (7.5)$$

We construct our filtered dataset by selecting only instances where the closed-source model produces correct solutions:

$$\mathcal{D}_{\text{SFT}} = \{(x_i, r_i, \hat{y}_i) \mid \text{Valid}(x_i, \hat{y}_i) = \text{True}, (x_i, y_i^*) \in \mathcal{D}\} \quad (7.6)$$

To manage computational costs while maintaining sufficient diversity in training examples, we limited our dataset to 20% of the benchmark size, ensuring a balanced representation across different problem types and difficulty levels.

### 7.3.3 Step 2: Reasoning Trace Learning

With our filtered dataset  $\mathcal{D}_{\text{SFT}}$ , we perform supervised fine-tuning on the open-source model  $\mathcal{M}_O$  to teach it both the final solutions and the reasoning process behind them.

### 7.3.3.1 Training Objective

We train  $\mathcal{M}_O$  to maximize the likelihood of generating both the reasoning trace  $r_i$  and the repaired code  $\hat{y}_i$  given the input  $x_i$ :

$$\mathcal{L}_{\text{SFT}} = - \sum_{(x_i, r_i, \hat{y}_i) \in \mathcal{D}_{\text{SFT}}} \log P_{\mathcal{M}_O}(r_i \oplus \hat{y}_i | x_i) \quad (7.7)$$

where  $P_{\mathcal{M}_O}(r_i \oplus \hat{y}_i | x_i)$  represents the probability of the model generating the reasoning trace followed by the repaired code.

### 7.3.3.2 Fine-tuning Implementation

We implement the fine-tuning process using the following configuration:

- *Optimizer*: AdamW with learning rate  $\eta = 5 \times 10^{-6}$  and weight decay  $\lambda = 0.01$
- *Training schedule*: Linear warmup followed by cosine decay
- *Batch size*: 4 sequences per device with gradient accumulation over 8 steps
- *Sequence length*: Maximum of 4096 tokens to accommodate detailed reasoning traces
- *Training epochs*: 3 complete passes through  $\mathcal{D}_{\text{SFT}}$
- *Mixed-precision training*: bfloat16 format for improved computational efficiency

### 7.3.3.3 Ablation: Direct Output Fine-tuning

To validate the importance of reasoning traces, we conduct an ablation study with a variant model  $\mathcal{M}_O^{\text{DO}}$  that is fine-tuned only on direct outputs without reasoning:

$$\mathcal{L}_{\text{DO}} = - \sum_{(x_i, \hat{y}_i) \in \mathcal{D}_{\text{SFT}}} \log P_{\mathcal{M}_O}(\hat{y}_i | x_i) \quad (7.8)$$

This allows us to isolate the specific contribution of reasoning trace learning to model performance.

## 7.3.4 Step 3: Reinforcement Learning with LLM Feedback (RL-LF)

While supervised fine-tuning helps transfer reasoning capabilities, it does not necessarily optimize for the quality criteria that distinguish exceptional repairs from merely functional ones. To address this, we introduce Reinforcement Learning with LLM Feedback (RL-LF), a novel approach that uses a closed-source LLM as a judge to provide preference-based feedback.

### 7.3.4.1 RL-LF Framework

Our RL-LF framework differs from standard RLHF<sup>1</sup> in several key aspects: ① It uses a closed-source LLM as the preference model instead of human annotators, ② specifically targets program repair quality rather than general helpfulness, ③ incorporates code-specific evaluation criteria into the reward function, and ④ leverages efficient preference data collection through careful sampling.

### 7.3.4.2 Reward Data Collection

To train our reward model, we first collect a dataset of preference judgments from the closed-source model. For each input  $x_i$ , we generate  $k$  different candidate repairs using the fine-tuned model  $\mathcal{M}_O^{\text{SFT}}$  with diverse decoding parameters:

$$\mathcal{Y}_i = \{y_i^1, y_i^2, \dots, y_i^k\} \text{ where } y_i^j \sim \mathcal{M}_O^{\text{SFT}}(x_i) \quad (7.9)$$

<sup>1</sup>Reinforcement Learning with Human Feedback

We then prompt the closed-source model  $\mathcal{M}_C$  to compare pairs of candidate repairs and express a preference:

$$\text{Pref}(y_i^a, y_i^b) = \mathcal{M}_C(x_i \oplus p_{\text{compare}} \oplus y_i^a \oplus y_i^b) \quad (7.10)$$

where  $p_{\text{compare}}$  is a prompt instructing the close-source model to analyze which repair is better according to:

- *Correctness*: Does it properly fix the bug?
- *Efficiency*: Is the solution efficient and optimized?
- *Readability*: Is the code clean and easy to understand?
- *Minimal change*: Does it modify only what's necessary to fix the bug?

The prompt is as follows:

*"I'll show you a programming task and multiple solution attempts. Your job is to evaluate each solution carefully, then rank them from best to worst."*

The result of each comparison ( $\text{Pref}(y_i^a, y_i^b)$ ) is converted to a binary preference label. By comparing repairs pairwise, we construct a dataset of preference judgments:

$$\mathcal{D}_{\text{pref}} = \{(x_i, y_i^a, y_i^b, \text{Pref}(y_i^a, y_i^b)) \mid (x_i, y_i^*) \in \mathcal{D}_{\text{val}}, y_i^a, y_i^b \in \mathcal{Y}_i\} \quad (7.11)$$

where  $\mathcal{D}_{\text{val}}$  is a held-out validation set.

#### 7.3.4.3 Reward Model Training

Using the preference dataset  $\mathcal{D}_{\text{pref}}$ , we train a reward model  $\mathcal{R}_\theta$  that predicts the likelihood that one repair is preferred over another:

$$P(y_i^a \succ y_i^b \mid x_i) = \sigma(\mathcal{R}_\theta(x_i, y_i^a) - \mathcal{R}_\theta(x_i, y_i^b)) \quad (7.12)$$

where  $\sigma$  is the logistic function and  $y_i^a \succ y_i^b$  denotes that  $y_i^a$  is preferred over  $y_i^b$ .

The reward model is trained to minimize the negative log-likelihood of the observed preferences:

$$\begin{aligned} \mathcal{L}_{\text{RM}} = & - \sum_{(x_i, y_i^a, y_i^b, p) \in \mathcal{D}_{\text{pref}}} \log(p \cdot \sigma(\mathcal{R}_\theta(x_i, y_i^a) \\ & - \mathcal{R}_\theta(x_i, y_i^b)) + (1-p) \cdot \sigma(\mathcal{R}_\theta(x_i, y_i^b) - \mathcal{R}_\theta(x_i, y_i^a))) \end{aligned} \quad (7.13)$$

where  $p = 1$  if  $y_i^a$  is preferred and  $p = 0$  if  $y_i^b$  is preferred.

#### 7.3.4.4 Policy Optimization with PPO

With the trained reward model  $\mathcal{R}_\theta$ , we fine-tune the SFT model  $\mathcal{M}_O^{\text{SFT}}$  using Proximal Policy Optimization (PPO) to maximize the expected reward while maintaining proximity to the original fine-tuned model:

$$\begin{aligned} \mathcal{L}_{\text{PPO}} = & \\ \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_\theta(y|x)} [\mathcal{R}_\theta(x, y) - \beta D_{\text{KL}}(\pi_\theta(y|x) \parallel \pi_{\text{SFT}}(y|x))] \end{aligned} \quad (7.14)$$

where:

- $\pi_\theta$  is the policy model being optimized
- $\pi_{\text{SFT}}$  is the original SFT model

- $\beta$  is a coefficient controlling the strength of the KL penalty
- $D_{\text{KL}}$  is the Kullback-Leibler divergence

To ensure stable training, we implement PPO with the clipped surrogate objective:

$$\mathcal{L}_{\text{CLIP}} = \mathbb{E}[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \quad (7.15)$$

where:

- $r_t(\theta) = \frac{\pi_\theta(y_t|x_t)}{\pi_{\text{old}}(y_t|x_t)}$  is the probability ratio
- $\hat{A}_t$  is the estimated advantage function
- $\epsilon$  is the clipping parameter (set to 0.2 in our implementation)

#### 7.3.4.5 Value Function and Advantage Estimation

To compute the advantage estimates required for PPO, we train a value function  $V_\phi(x, y)$  that predicts the expected reward for a given input-output pair:

$$\mathcal{L}_{\text{Value}} = \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_\theta(y|x)}[(V_\phi(x, y) - \mathcal{R}_\theta(x, y))^2] \quad (7.16)$$

The advantage function is then estimated using Generalized Advantage Estimation (GAE):

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma \lambda)^l \delta_{t+l} \quad (7.17)$$

where  $\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$  is the temporal difference error.

#### 7.3.5 Novelty of RL-LF

Our RL-LF approach introduces several innovations over existing reinforcement learning methods for LLM fine-tuning:

1. **Domain-Specific Preference Modeling:** Unlike general RLHF approaches, RL-LF incorporates program repair-specific criteria into the preference model, focusing on correctness, efficiency, readability, and minimality of changes.
2. **Scalable Preference Collection:** By using a bot (here an LLM acting as AI agent) as the preference model, we overcome the scaling limitations of human feedback collection while maintaining high-quality judgments tailored to program repair.
3. **Complementary Integration with Reasoning Traces:** RL-LF builds upon the foundation of reasoning trace learning, creating a synergistic effect where explicit reasoning capabilities are aligned with implicit quality preferences.
4. **Adaptability to Evolving Benchmarks:** The RL-LF framework can dynamically adapt to new program repair challenges without requiring additional human annotation, making it sustainable for long-term model improvement.
5. **Cross-Model Knowledge Transfer:** Our approach enables effective knowledge transfer between models with different architectures and training paradigms, bridging the gap between closed and open-source capabilities.

These innovations allow RL-LF to effectively transfer the nuanced preferences and quality judgments of closed-source models to open-source alternatives, addressing a key limitation of existing approaches that focus primarily on functional correctness rather than repair quality.

### 7.3.6 Implementation Details

The Repairity implementation leverages 8 H100 GPUs (80GB) with DeepSpeed ZeRO-2 Rasley et al. [2020], applying LoRA (rank=4, alpha=16, dropout=0.05) to attention modules, with parameters finely tuned through 3 training epochs at learning rate 1e-5 and effective batch size of 8 (1x8 gradient accumulation). The SFT phase processes benchmark data with 2048-token context windows and FP16 precision, while the evaluation model operates at temperature 0.2 for deterministic assessment. The RL-LF pipeline collects Claude-Sonnet3.7 preferences on 3 diverse solutions per problem (temperature 0.7-0.9), training a CodeLlama-7b reward model over 3 epochs (learning rate 5e-5, batch size 4x4 gradient accumulation) which achieves 85% preference accuracy. This guides PPO fine-tuning with KL coefficient 0.1, 5 PPO epochs per batch, response lengths 546-732 tokens, and generation temperature 1.0 during exploration.

These implementation details ensure that our methodology can be replicated and extended by other researchers, providing a foundation for future work on open-source model enhancement.

## 7.4 Experimental Setup

In this section, we describe our experimental methodology for evaluating Repairity, including the benchmarks, evaluation metrics, baselines, and experimental settings.

### 7.4.1 Benchmarks

We evaluate our approach using four diverse benchmarks summarized in Table 7.1.

**Table 7.1:** Overview of Benchmark Tasks and Dataset Statistics

Benchmark	Task Type	Total Samples
MBPP	Code Generation	974
SWE-bench Verified	Program Repair	500
Defects4J	Program Repair	835
BigCodeBench	Code Completion	1,140

- **BigCodeBench** Lai et al. [2023]: A comprehensive benchmark containing a diverse set of coding problems across multiple programming languages and difficulty levels. It provides a robust test of general code generation and repair capabilities.
- **MBPP (Mostly Basic Programming Problems)** Austin et al. [2021]: A collection of 964 programming problems designed to evaluate basic programming skills. These problems are typically shorter and more focused than those in other benchmarks.
- **SweBench-verified** Jimenez-Sanchez et al. [2023]: A benchmark specifically designed for software engineering tasks, with verified solutions that include unit tests to validate functional correctness. This benchmark focuses on real-world programming scenarios.
- **Defects4J** Just et al. [2014]: A database of real bugs from open-source Java projects, providing a challenging test of program repair capabilities on production-level code. Each bug comes with a corresponding test suite that can be used to validate repairs.

For training, we use a subset of these benchmarks to construct our SFT dataset and RL-LF preference data. For evaluation, we use held-out portions to ensure a fair assessment of model performance.

### 7.4.2 Evaluation Metrics

We employ the following metrics to evaluate program repair performance:

- **Pass@1** Chen et al. [2021a]: The percentage of problems for which the model’s first generated solution passes all test cases. This metric evaluates the model’s ability to produce correct repairs on the first attempt.
- **Accuracy** Austin et al. [2021]: The percentage of programming problems for which the model generates code that passes all test cases. This metric is commonly used on datasets like MBPP to measure the model’s ability to correctly implement a solution based on a natural language problem description.
- **Compilation Rate (CR)** Just et al. [2014]: the percentage of generated or modified code that successfully compiles without errors.
- **BLEU** Papineni et al. [2002]: the metric calculates the geometric mean of modified n-gram precisions, penalized by a brevity penalty, to evaluate the quality of machine-generated translations against one or more reference translations
- **Resolved** Jimenez-Sanchez et al. [2023]: The percentage of task instances where the generated patch applies successfully and passes all test cases. A solution is considered to have "resolved" the issue when it can be cleanly applied to the codebase and satisfies all the specified test requirements.

### 7.4.3 Models

We compare the following models and variants in our evaluation:

- **Closed-Source Model:** Claude-Sonnet3.7 Anthropic [2025], a state-of-the-art closed-source LLM that serves as our performance target and source of reasoning traces.
- **Base Open-Source Model:** Qwen2.5-Coder-32B-Instruct Hui et al. [2024], our starting point representing current open-source capabilities.
- **Repairity (SFT):** The base model fine-tuned using our Supervised Fine-Tuning with Reasoning Traces approach.
- **Repairity (SFT + RL-LF):** Our complete model incorporating both Reasoning Trace Learning and Reinforcement Learning with LLM Feedback.
- **Ablation - Direct Output Fine-tuning:** The base model fine-tuned only on direct outputs without reasoning traces.
- **Ablation - SFT without Filtering:** The base model fine-tuned on all reasoning traces without correctness filtering.

### 7.4.4 Experimental Procedure

Our experimental procedure consists of the following steps:

1. **Reasoning Trace Collection:** We query Claude 3.7 with each training example to collect reasoning traces, using the process described in Section 7.3, inspired by chain-of-thought elicitation techniques Wei et al. [2022].
2. **Data Splitting:** We split collected reasoning trace data into training (70%), validation (15%), and test (15%) sets, following standard practice in machine learning evaluation Hastie et al. [2009]. The training set is used for SFT, the validation set for hyperparameter tuning and early stopping, and the test set for final evaluation.
3. **SFT Model Training:** We fine-tune Qwen2.5-Coder-32B-Instruct on the filtered reasoning traces using the hyperparameters specified in Section 7.3, following best practices for LLM fine-tuning Hu et al. [2021].
4. **We only use reasoning trace data for SFT instead of using ground-truth data.**
5. **RL-LF Preference Collection:** We generate multiple candidate solutions using the SFT model and collect preference judgments from Claude 3.7, adapting the preference collection methodology from Stiennon et al. [2020].
6. **Reward Model Training:** We train a CodeLlama-7b reward model on the collected preferences, following the Bradley-Terry preference modeling approach Christiano et al. [2017], Ouyang et al. [2022].
7. **PPO Fine-tuning:** We fine-tune the SFT model using PPO Schulman et al. [2017] with the trained reward model.
8. **Evaluation:** We evaluate all models on the test sets using the metrics described above, with significance testing to validate our findings Dror et al. [2018].

### 7.4.5 Computational Resources

All experiments were conducted using the hardware and software configuration described in Section 7.3. The total computation used for this research includes:

- **SFT Data Collection:** Approximately 756 output tokens for each API call to claude-3-7-sonnet-20250219 for reasoning trace collection.
- **SFT Model Training:** Approximately 3 hours on 8× H100 GPUs with 5 epochs.
- **RL-LF Preference Collection:** Approximately 673 output tokens for each API call to claude-3-7-sonnet-20250219.

- **Reward Model Training:** Approximately 0.5 GPU hours per epoch.
- **PPO Fine-tuning:** Approximately 4.5 GPU hours with 5 epochs.

All models were implemented using the HuggingFace Transformers library, with DeepSpeed for distributed training optimization.

## 7.5 Experimental Results

In this section, we present a comprehensive evaluation of Repairity across multiple program repair and code generation benchmarks. We present results related to quantitative performance analysis, ablation studies, case studies, and generalization experiments.

### 7.5.1 Performance Comparison

We apply Repairity on each benchmark and compute performance based on metrics associated with the given benchmark. We further report, for comparison, the performance metrics of other state-of-the-art or baseline models for each benchmark.

#### 7.5.1.1 BigCodeBench

On BigCodeBench (Table 7.2), Repairity achieves a Pass@1 rate of 35.6%, which represents a remarkable 4.8% absolute improvement over the base Qwen2.5-Coder-32B-Instruct model (30.8%). This brings Repairity within 0.2 percentage points of Claude-Sonnet3.7 (35.8%), effectively closing 96% of the performance gap. Notably, Repairity even outperforms some closed-source models like DeepSeek-R1, demonstrating the effectiveness of our reasoning transfer approach.

**Table 7.2:** BigCodeBench Pass@1 Results

Model	Pass@1
Claude 3.7 Anthropic [2025]	35.8
o1 Jaech et al. [2024]	35.5
o3 OpenAI [2025]	35.5
DeepSeek-R1 Guo et al. [2025]	35.1
Qwen2.5-Coder-32B-Instruct Qwen Team [2023]	30.8
<b>Repairity (Ours)</b>	<b>35.6</b>

#### 7.5.1.2 SWE-bench Verified

The results on SWE-bench Verified (Table 7.3) show an even more dramatic improvement. Repairity achieves a resolution rate of 62.7%, compared to 38.2% for the base Qwen model—a substantial 24.5% absolute improvement. This performance exceeds Claude-Sonnet3.7’s default mode (62.3%) and approaches its performance with scaffolding (70.3%). Repairity significantly outperforms other models like Nebius AI Qwen 2.5 72B + Llama 3.1 70B (40.6%) and OpenAI’s o1/o3-mini models (48.9%/49.3%), demonstrating its effectiveness on complex software engineering tasks.

**Table 7.3:** SWE-bench Verified Results

Model	Resolved (%)
Claude 3.7 Sonnet (with scaffold) Anthropic [2025]	70.3
Claude 3.7 Sonnet (default) Anthropic [2025]	62.3
Nebius AI Qwen 2.5 72B + Llama 3.1 70B	40.6
Qwen2.5-Coder-32B-Instruct Hui et al. [2024]	38.2
o1 Jaech et al. [2024]	48.9
o3-mini OpenAI [2025]	49.3
<b>Repairity (Ours)</b>	<b>62.7</b>

### 7.5.1.3 MBPP

On the MBPP benchmark (Table 7.4), Repairity achieves 92.5% accuracy, a 3.7% improvement over the base Qwen model (88.8%). This performance surpasses both Claude 3.7 (89.5%) and GPT-4 (87.5%), approaching the levels of specialized models like QualityFlow (94.2%) and o1-mini + MapCoder (93.2%). These results suggest that our approach effectively transfers reasoning capabilities for solving basic programming problems.

**Table 7.4:** MBPP Accuracy Results

Model	Accuracy
QualityFlow (Sonnet-3.5) Hu et al. [2025]	94.2
o1-mini + MapCoder (Hamming.ai) Islam et al. [2024]	93.2
Claude 3 Opus Anthropic [2023]	86.4
Claude 3.7 Anthropic [2025]	89.5
Qwen2.5-Coder-32B-Instruct Hui et al. [2024]	88.8
GPT-4 OpenAI [2023]	87.5
<b>Repairity (Ours)</b>	<b>92.5</b>

### 7.5.1.4 Defects4J

For Defects4J (Table 7.5), Repairity improves across all metrics compared to the base model: compilation rate increases from 77.1% to 78.9%, Pass@1 from 64.0% to 66.5%, and BLEU score from 67.8% to 68.9%. While these improvements are more modest than on other benchmarks, they still represent significant progress toward closed-source performance (Claude 3.7: 79.5% CR, 67.2% Pass@1, 69.5% BLEU). The smaller gains may reflect the specific challenges of real-world Java program repair in Defects4J, which often requires complex contextual understanding.

**Table 7.5:** Defects4J Results

Model	Compilation Rate (CR) (%)	Pass@1	BLEU
RepoCoder Zhang et al. [2023a]	74.02	59.8	63.52
RAMbo Bui et al. [2024]	76.47	63.73	66.29
Claude 3.7 Anthropic [2025]	79.5	67.2	69.5
Qwen2.5-Coder-32B-Instruct Hui et al. [2024]	77.1	64.0	67.8
<b>Repairity (Ours)</b>	<b>78.9</b>	<b>66.5</b>	<b>68.9</b>

Overall, the results demonstrate that our approach successfully narrows the performance gap between open and closed-source models.

**[Finding-#1]** Repairity achieves near parity with state-of-the-art closed-source models across multiple benchmarks, closing up to 93% of the performance gap between the base open-source Qwen2.5-Coder-32B-Instruct and Claude-Sonnet3.7.

**[Finding-#2]** Repairity demonstrates the largest performance gains on complex software engineering tasks (SWE-bench Verified: +24.5%), showing that reasoning transfer is particularly effective for tasks requiring deep code understanding and multi-step problem solving.

## 7.5.2 Ablation Study

To understand the contribution of each component in our approach, we conducted ablation studies across all benchmarks. Tables 7.6(1) through 7.6(4) present results for the

base model, the model with only Reasoning Trace (RT) learning, and the complete model with both RT and RL-LF.

**Table 7.6:** Ablation studies on different benchmarks: BigCodeBench (Pass@1), SWE-bench Verified (Resolved %), MBPP (Accuracy), and Defects4J (Compilation Rate, Pass@1, and BLEU. ‘base’ represents ‘Qwen2.5-Coder-32B-Instruct’)

Model	Pass@1	Model	Resolved (%)
base	30.8	base	38.2
Repairity <sub>base+RT</sub>	31.7	Repairity <sub>base+RT</sub>	47.6
Repairity <sub>base+RT+RLLF</sub>	35.6	Repairity <sub>base+RT+RLLF</sub>	<b>62.7</b>
(1) BigCodeBench			
Model	Accuracy	Model	CR (%)
base	88.8	base	77.1
Repairity <sub>base+RT</sub>	89.0	Repairity <sub>base+RT</sub>	77.3
Repairity <sub>base+RT+RLLF</sub>	<b>92.5</b>	Repairity <sub>base+RT+RLLF</sub>	<b>78.9</b>
(3) MBPP			
(2) Swebench-Verified			
Model	CR (%)	Pass@1	BLEU
base	77.1	64.0	67.8
Repairity <sub>base+RT</sub>	77.3	65.9	67.9
Repairity <sub>base+RT+RLLF</sub>	<b>78.9</b>	<b>66.5</b>	<b>68.9</b>
(4) Defects4J			

### 7.5.2.1 Component Contributions

Across all benchmarks, we observe that:

1. **Reasoning Trace (RT) Learning** provides modest but consistent improvements over the base model:

- BigCodeBench: +0.9% (30.8% → 31.7%)
- SWE-bench: +9.4% (38.2% → 47.6%)
- MBPP: +0.2% (88.8% → 89.0%)
- Defects4J: +1.9% (64.0% → 65.9%) in Pass@1

2. **RL-LF** contributes substantial additional improvements:

- BigCodeBench: +3.9% (31.7% → 35.6%)
- SWE-bench: +15.1% (47.6% → 62.7%)
- MBPP: +3.5% (89.0% → 92.5%)
- Defects4J: +0.6% (65.9% → 66.5%) in Pass@1

These results highlight that while RT learning helps transfer reasoning patterns, the RL-LF component is critical for achieving performance parity with closed-source models. The synergistic effect is most pronounced on SWE-bench Verified, where RT learning provides a 9.4% improvement and RL-LF adds another 15.1%, together yielding a 24.5% absolute gain.

**[Finding-#3]** Both components of Repairity contribute significantly to performance gains, with RL-LF providing substantially larger improvements than Reasoning Trace learning alone. This demonstrates the complementary nature of our two-stage approach.

### 7.5.2.2 Benchmark-Specific Patterns

The relative contribution of each component varies across benchmarks:

- On simpler tasks (MBPP), RT learning provides minimal gains (+0.2%), while RL-LF drives most improvement (+3.5%)
- On complex software engineering tasks (SWE-bench), both components provide substantial gains (RT: +9.4%, RL-LF: +15.1%)
- On real-world bug fixing (Defects4J), RT learning contributes more significantly to Pass@1 improvement than RL-LF

These patterns suggest that the value of explicit reasoning traces increases with task complexity, while preference-based reinforcement learning consistently improves performance across all task types.

**[Finding #4]**  *The effectiveness of Reasoning Trace learning correlates with task complexity—providing greater benefits on complex software engineering tasks than on simpler programming problems.*

### 7.5.3 Case Study

To provide qualitative insights into how Repairity approaches program repair tasks, we analyzed representative examples from BigCodeBench and SWE-bench Verified. These examples showcase the model’s reasoning patterns and solution strategies for different types of programming challenges.

#### [BigCodeBench]

**Task\_id: BigCodeBench/56**

**Predict: correct**

**Input:**

Extract data from a text and create a Pandas DataFrame. The text contains several lines, each formatted as 'Score: 85, Category: Math'. Make sure to convert the scores into integers. The function should output: DataFrame: A pandas DataFrame with extracted data. You should write self-contained code starting with:

```
1 import pandas as pd
2 import regex as re
3 def task_func(text):
```

#### Reference Solution:

```
1 pattern = r"Score: (.*?), Category: (.*?)(\n|$)"
2 matches = re.findall(pattern, text)
3 data = [match[:2] for match in
4     ↪ matches] # Extracting only the score and category from each match
4 df = pd.DataFrame(data, columns=["Score", "Category"])
5 df["Score"] = df["Score"].astype(int)
6 return df
```

#### Reasoning Trace of Repairity:

Here's my thought process behind the implementation:

- 1. Problem Analysis: 1) We need to extract data from text where each line follows the pattern "Score: X, Category: Y" 2) We need to convert the scores to integers 3) The output should be a pandas DataFrame with appropriate columns;
- 2. Regex Pattern Design:
  - 1) I created a pattern r'Score: (.), Category: (.)' that: Captures the numeric score with (.) to match one or more digits Captures the category name with (.) to match the rest of the text on the line
  - 2) This pattern will work correctly even if category names contain spaces or special characters;
- 3. Data Extraction:
  - 1) Using re.findall() to get all matches in the text

- 2) Each match will be a tuple with two elements: the score and category
- 4. Data Processing: I iterate through each match to:
  - 1) Convert the score to integer with `int(match[0])`
  - 2) Keep the category as a string
  - 3) Create a dictionary for each data point with proper column names
- 5. DataFrame Creation:
  - 1) Create a pandas DataFrame from the list of dictionaries
  - 2) This automatically sets up the correct column names ("Score" and "Category")
  - 3) The Score column is already converted to integers during the processing step

**Codes:**

```

1 import pandas as pd
2 import re
3
4 def task_func(text):
5     # Define the pattern to extract score and category
6     pattern = r'Score: (\d+), Category: (.)'
7
8     # Find all matches in the text
9     matches = re.findall(pattern, text)
10
11    # Create a list of dictionaries to store the data
12    data = []
13    for match in matches:
14        score = int(match[0])  # Convert score to integer
15        category = match[1]
16        data.append({'Score': score, 'Category': category})
17
18    # Create a DataFrame from the list of dictionaries
19    df = pd.DataFrame(data)
20    return df

```

**[Swebench-Verified]**

**Instance ID: django\_django-12050**

**Predict: resolved**

**Input:**

Repository: django/django at

**Commit SHA:** 'b93a0e34d9b9b99d41103782b7e7aeabf47517e3'

and **problem statement:** `Query.resolve_lookup_value` coerces value of type list to tuple Description. Changes introduced in #30687 cause an input value list to be coerced to tuple breaking exact value queries. This affects ORM field types that are dependent on matching input types such as `PickledField`.

The expected iterable return type should match input iterable type.

**Ground Truth Patch:**

```

1 diff git
2     → a/django/db/models/sql/query.py b/django/db/models/sql/query.py
3     a/django/db/models/sql/query.py
4     +++ b/django/db/models/sql/query.py
5     @@ 1059,21 1059,10 @@ def resolve_lookup_value
6         → (self, value, can_reuse, allow_joins, simple_col):
7             elif isinstance(value, (list, tuple)):
8                 resolved_values = []
9                 for sub_value in value:
10                     if hasattr(sub_value, 'resolve_expression'):
11
12
13
14
15
16
17
18
19
1

```

```

9  ....
10 -     value = tuple(resolved_values)
11 -     return type(value) (
12 -         self.resolve_lookup_value
13 -         ↪ (sub_value, can_reuse, allow_joins, simple_col)
14 -         for sub_value in value
15 -     )
16     return value
def solve_lookup_type(self, lookup):

```

### Reasoning Trace of Repairity:

- **Identifying the Problem:** The function `Query.resolve_lookup_value` in `django/db/models/sql/query.py` should preserve the type of iterable lookup values. A change in #30687 caused lists to be coerced into tuples, breaking queries for fields like `PickledField`. The issue is traced to commit `b93a0e34d9b9b99d41103782b7e7aeabf47517e3`.
- **Analyzing the Root Cause:** The function processes lists and tuples but always converts them to tuples. This happens because it iterates over elements, applies transformations, and then reconstructs the result as a tuple instead of preserving the original type.
- **Formulating the Fix:** The fix replaces `value = tuple(resolved_values)` with `return type(value) (...)` to maintain the input type dynamically.
- **Modifying the Patch While Keeping Functionality:** Changes include renaming `resolved_values` to `resolved`, changing `sub_value` to `item` before `v` in the generator expression, and directly constructing the iterable using `type(value) (...)`.
- **Validation:** Running Django's test suite (`pytest` or `manage.py test`) ensures lists remain lists, tuples remain tuples, and iterable items are correctly processed.

### Generated Patch:

```

1 diff git
2   ↪ a/django/db/models/sql/query.py b/django/db/models/sql/query.py
3 a/django/db/models/sql/query.py
4 +++ b/django/db/models/sql/query.py
5 @@ 1059,21 1059,10 @@ def resolve_lookup_value
6   ↪ (self, value, can_reuse, allow_joins, simple_col):
7     elif isinstance(value, (list, tuple)):
8
9     ↪ # Ensure that each element in the iterable is resolved correctly.
10    -     resolved = [
11    -       self
12    -         ↪ .resolve_lookup_value(item, can_reuse, allow_joins, simple_col)
13    -         for item in value
14    -     ]
15    -     value = tuple(resolved)
16    return type(value)(self.resolve_lookup_value
17    ↪ (v, can_reuse, allow_joins, simple_col) for v in value)
18    return value
19
20 def solve_lookup_type(self, lookup):

```

#### 7.5.3.1 Structured Data Extraction

In the BigCodeBench example depicted in case study, Repairity demonstrates a systematic approach to parsing structured text data. The reasoning trace reveals a clear problem decomposition strategy:

(1) Problem analysis and identification of key sub-tasks; (2) Regex pattern design with explicit consideration of edge cases; (3) Data extraction with appropriate type conversion; (4) Structured DataFrame creation.

This reasoning pattern mirrors the approach a human programmer would take, breaking down the problem into manageable components and addressing each methodically. The implementation correctly handles score conversion to integers—a critical requirement—while maintaining a clean, readable solution.

### 7.5.3.2 Type Coercion Bug Fix

Due to space requirements, verbatim model output for this use case on Django from SWE-bench Verified are reported in the supplementary file.

For the Django bug fix case, Repairity demonstrates deeper reasoning capabilities related to program repair:

1. Precise bug identification (coercion of lists to tuples)
2. Root cause analysis in the existing code
3. Principled solution development (preserving input type with `type(value)`)
4. Verification considerations (test cases for different input types)

The generated patch is functionally equivalent to the ground truth solution, using `type(value) ( . . . )` to dynamically reconstruct the output with the same type as the input. This demonstrates Repairity’s ability to lead the base model toward understanding subtle programming concepts like type preservation and iterative data transformation.

**[Finding-#5] 🕵 Repairity’s reasoning traces reveal systematic problem decomposition, targeted debugging, and principled solution development—closely mirroring expert human problem-solving approaches.**

### 7.5.4 Generalization Capability

To assess Repairity’s ability to generalize across different code-related tasks, we conducted cross-benchmark experiments. We fine-tuned the model on SWE-bench Verified (500 data points) and evaluated it on the full BigCodeBench dataset (1,140 data points).

Repairity achieves 32.7% Pass@1 on the full BigCodeBench dataset and 33.4% on the test split. While this represents a slight performance drop compared to direct fine-tuning on BigCodeBench (35.6%), it still substantially outperforms the base model (30.8%), demonstrating effective transfer learning across different code-related tasks.

**[Finding-#6] 🕵 Repairity demonstrates strong generalization capabilities, successfully transferring reasoning skills across different code-related tasks and benchmarks without requiring task-specific fine-tuning.**

### 7.5.5 Summary of Results

Our experimental evaluation demonstrates that Repairity successfully closes the performance gap between open and closed-source models across diverse program repair and code generation benchmarks. The approach provides the most substantial improvements on complex software engineering tasks, with gains of up to 24.5% in absolute performance. Both components of our methodology—Reasoning Trace learning and RL-LF—contribute significantly, with RL-LF proving particularly impactful.

The qualitative analysis reveals that Repairity develops systematic reasoning patterns similar to human experts, including structured problem decomposition, principled solution

development, and verification considerations. Furthermore, the model demonstrates strong generalization capabilities, transferring reasoning skills across different code-related tasks.

These results validate our core postulate that transferring reasoning capabilities from closed-source to open-source models can substantially improve performance on complex software engineering tasks, enabling organizations to leverage high-quality program repair capabilities while maintaining the flexibility, customizability, and privacy advantages of open-source models.

## 7.6 Discussion

This section examines the implications of our experimental results, discusses limitations, and explores future directions.

### 7.6.1 Analysis of Key Findings

Our experimental results highlight several important insights about reasoning transfer between LLMs for program repair tasks.

#### 7.6.1.1 Effectiveness of Reasoning Transfer

The significant improvements achieved by Repairity, particularly on complex software engineering tasks, demonstrate that transferring reasoning processes—not just outputs—from closed-source to open-source models is highly effective. The varying impact across benchmarks suggests that explicit reasoning becomes increasingly valuable as task complexity increases, with the largest gains observed on SWE-bench Verified (+24.5%). This aligns with research on the importance of reasoning in LLMs Wei et al. [2022], Kojima et al. [2022].

#### 7.6.1.2 Synergy Between RT Learning and RL-LF

The consistent pattern across all benchmarks shows that while Reasoning Trace (RT) learning provides modest but important improvements, RL-LF drives substantially larger gains. This complementary relationship suggests that RT learning enables the model to develop systematic problem-solving approaches, while RL-LF refines these capabilities based on implicit quality judgments that are difficult to capture through demonstrations alone.

The ablation studies reveal that this synergy is particularly powerful for complex tasks, where RT learning provides a foundation of structured reasoning that RL-LF can then optimize. This two-stage approach addresses limitations of previous methods that focused solely on either imitation learning or reinforcement learning.

### 7.6.2 Limitations and Future Work

While Repairity demonstrates significant progress toward bridging the gap between open and closed-source models, several limitations warrant consideration.

First, our approach depends on access to a high-quality closed-source model, creating a bootstrapping challenge. Future work could explore iterative improvement where each generation of enhanced open-source models helps train subsequent ones, gradually reducing this dependency.

Second, computational efficiency remains challenging due to the verbose nature of reasoning traces. More efficient methods for reasoning transfer could improve scalability, perhaps by identifying and focusing on the most informative parts of reasoning traces ?.

Third, while Repairity shows strong generalization capabilities, maintaining 92% of its performance when transferring from SWE-bench to BigCodeBench, this indicates some degree of benchmark-specific learning. Future work should explore techniques to enhance cross-domain generalization, possibly through meta-learning approaches or more diverse training data.

Future research directions could include:

- Multi-model reasoning ensembles that combine insights from different models
- Task-specific reasoning strategies tailored to different types of programming problems
- Integration of human feedback to refine reasoning patterns before fine-tuning

### 7.6.3 Implications for Software Engineering

The performance of Repairity, approaching and sometimes exceeding closed-source models, has significant implications for software engineering practice.

By narrowing the gap between open and closed-source models, Repairity helps democratize access to advanced program repair capabilities. Organizations with privacy or customization requirements can now leverage high-quality alternatives without relying on API-based solutions.

Additionally, the explicit reasoning traces generated by Repairity could serve educational purposes, exposing novice programmers to structured problem-solving approaches. This transparency in reasoning also addresses a key limitation of current tools, where solutions are provided without explanation.

### 7.6.4 Ethical Considerations

Using closed-source models to improve open-source alternatives raises questions about intellectual property and appropriate attribution. While our approach does not extract model weights or architecture details, it does transfer knowledge embodied in outputs. Future work should explore frameworks for ethical knowledge transfer that balance innovation with appropriate attribution.

Responsible deployment should include integration with existing security tools, clear communication of model limitations, and ongoing monitoring for unintended consequences.

## 7.7 Related Work

Our work builds upon and extends several research directions in program repair, knowledge transfer between language models, and reasoning enhancement.

### 7.7.1 Program Repair with LLMs

Recent work has demonstrated the effectiveness of LLMs for automated program repair tasks. ChatRepair Zhao et al. [2023] and Repilot Wei et al. [2023c] leverage ChatGPT’s capabilities for bug fixing, while RING Dakhel et al. [2023] focuses on integration with development workflows. Our approach differs by specifically addressing the performance gap between open and closed-source models through reasoning transfer, rather than developing methods that rely on proprietary models.

### 7.7.2 Knowledge Transfer Between LMs

Knowledge distillation techniques have been widely explored for transferring capabilities between models of different sizes Hinton et al. [2015], Sanh et al. [2019]. In the code domain, CodeDistill Wang et al. [2023e] demonstrated distillation for code generation tasks. Our work extends these approaches by focusing specifically on transferring reasoning capabilities rather than general language abilities, and by combining distillation with reinforcement learning for optimal results.

### 7.7.3 Reasoning Enhancement in LLMs

Chain-of-thought prompting Wei et al. [2022] and similar techniques have shown the value of explicit reasoning for complex tasks. Program repair presents unique challenges for reasoning, as explored by ARCADE Jin et al. [2023] and PACE Zhang et al. [2023c]. Our contribution is the development of a systematic approach to extract, filter, and transfer reasoning patterns from closed-source to open-source models, combined with preference-based optimization.

### 7.7.4 Reinforcement Learning for Model Alignment

Reinforcement learning from human feedback (RLHF) has become a standard approach for aligning language models with human preferences Christiano et al. [2017], Ouyang et al. [2022]. Recent work has explored using AI feedback instead of human annotations Lee et al. [2023], Bai et al. [2022]. Our RL-LF approach builds upon these foundations but specializes them for program repair by incorporating code-specific evaluation criteria and efficient preference data collection.

## 7.8 Conclusion

We presented Repairity, a methodology for enhancing open-source LLMs through reasoning transfer and reinforcement learning with LLM feedback. Our experimental results across four benchmarks demonstrate substantial improvements (up to +24.5% on SWE-bench Verified), significantly narrowing the performance gap with closed-source models. Both components—reasoning trace learning and RL-LF—contribute to these gains, with their synergistic effect most pronounced on complex tasks requiring multi-step reasoning. This work shows that reasoning transfer is a viable approach for improving open-source models without accessing proprietary data or weights, enabling organizations to leverage high-quality program repair capabilities while maintaining the advantages of open-source deployment. Future work could extend this approach to other software engineering tasks and explore more efficient methods for reasoning transfer.

**Open Science.** To promote transparency and facilitate reproducibility, we make our artifacts available to the community at:

<https://anonymous.4open.science/r/REPARITY-9BE1>.

The repository includes the experiment scripts and the results as well as the checkpoints. In addition, A supplementary file containing complementary analysis has been included to provide further details about this work.



# 8 Conclusion

This dissertation has presented a comprehensive framework for automating code change analysis and generation through Foundation Models. Code changes—the lifeblood of software evolution—drive 70% of software maintenance costs and are responsible for critical system failures, yet existing approaches fail to capture their multi-faceted nature: their intent, semantic context, and structural transformations. Through five interconnected contributions, we have established a new paradigm that progresses from representation to reasoning, from isolation to collaboration, and from exclusivity to democratization.

Our journey began with Patcherizer, which introduced dual-encoding architecture capturing both sequential and structural aspects of code changes through SeqIntention and GraphIntention modules. This foundational representation achieved state-of-the-art performance—improving commit message generation by 19.39% BLEU and pioneering semantic intention detection beyond syntactic operations. Building upon this foundation, LLMDA addressed silent security patches through LLM-generated explanations and multi-modal alignment via PT-Former, achieving 42% improvement in detecting undocumented vulnerabilities. CodeAgent transformed code review into multi-agent collaboration, with our QA-Checker innovation preventing prompt drifting while achieving 41% improvement in vulnerability detection. SynFix conquered repository-wide repair through CallGraph-driven dependency modeling, resolving 52.33% of SWE-bench-lite issues while maintaining deterministic operation. Finally, Repairity bridged the performance gap between closed and open-source models through reasoning transfer and Reinforcement Learning with LLM Feedback, boosting open-source performance by 24.5% and closing 98% of the capability gap.

Collectively, these contributions demonstrate that code changes possess inherent duality requiring both sequential and structural understanding, that security implications hide in semantic gaps between syntax and intent, and that effective automation demands multi-perspective collaboration. Our systems address real-world challenges: LLMDA protects applications from silent vulnerabilities, CodeAgent reduces manual review effort while maintaining quality, SynFix prevents cascading failures, and Repairity enables privacy-conscious organizations to leverage advanced AI capabilities. This shift from mechanical processing to intelligent understanding represents a fundamental evolution in software engineering.

The journey from Patcherizer’s representations through LLMDA’s security focus, CodeAgent’s collaboration, SynFix’s scale, to Repairity’s democratization illustrates a complete evolution: learning to represent, detect, collaborate, repair, and teach. This progression establishes not just individual solutions but a comprehensive framework where code changes are truly understood—their intent decoded, their impact assessed, and their implications managed across entire systems.



# 9 Future Work

*This chapter outlines promising directions for future research that extend our contributions in code change representation, security patch detection, code review automation, repository-wide program repair, and bridging the gap between open and closed-source models. We focus on two main areas: efficient reasoning strategies for large language models and advancements in automatic coding capabilities. These research directions aim to address current limitations and open challenges identified throughout this dissertation.*

## Contents

---

<b>9.1</b>	<b>Efficient Reasoning Strategies</b>	<b>186</b>
9.1.1	MoE Optimization for Faster Inference	186
9.1.2	Draft-Based Reasoning with RL Selection	186
<b>9.2</b>	<b>Automatic Coding Advancements</b>	<b>187</b>
9.2.1	Comprehensive Benchmark for Coding from Scratch	187
9.2.2	Structural Routing for Coding and Debugging	188
<b>9.3</b>	<b>Integration of Future Directions</b>	<b>188</b>

---

## 9.1 Efficient Reasoning Strategies

The reasoning capabilities of large language models are fundamental to their performance on complex code-related tasks. However, current approaches often involve verbose, token-intensive processes that increase computational costs and latency Wei et al. [2022], Yin et al. [2024]. Future research should focus on developing more efficient reasoning strategies that maintain or improve performance while reducing resource requirements.

### 9.1.1 MoE Optimization for Faster Inference

Mixture-of-Experts (MoE) models have emerged as a compelling solution to the computational challenges of large language models, enabling massive parameter counts while maintaining manageable computational costs during inference Fedus et al. [2022], Lepikhin et al. [2021]. Despite their efficiency advantages over dense models, MoE architectures still face significant deployment challenges, particularly regarding GPU memory requirements and inference optimization.

The fundamental challenge in MoE inference lies in the trade-off between model capacity and resource constraints. While MoE models achieve their efficiency by activating only a subset of experts for each input token, the traditional approach requires loading all expert parameters into GPU memory, creating a bottleneck for deployment on resource-constrained environments. This limitation has hindered the widespread adoption of large MoE models, particularly in scenarios where GPU memory is limited or expensive.

Recent work has explored various approaches to optimize MoE inference, including expert pruning Liu et al. [2024a], dynamic routing Latifi et al. [2022], and parameter sharing strategies Clark et al. [2022]. However, these approaches typically focus on either static optimization or simple heuristic-based selection methods, failing to capture the complex dependencies between task characteristics, expert specialization, and dynamic inference requirements.

A promising direction for future work is to develop a reinforcement learning framework that addresses the MoE inference optimization problem through a two-phase approach:

- **Pre-loading Controller:** This component would intelligently select which subset of experts to load into GPU memory based on task characteristics and hardware constraints. For code-related tasks, this could involve pre-loading experts specialized in syntax analysis, semantic understanding, or specific programming languages based on the nature of the input.
- **Runtime Controller:** This component would dynamically activate specific experts from the pre-loaded set for each token during inference. For code change analysis or program repair, this could enable more efficient processing by activating only the most relevant experts for each part of the code being analyzed.

Both controllers would be trained using reinforcement learning with reward functions that explicitly balance model quality, computational efficiency, and expert utilization patterns. This approach could significantly reduce the memory and computational requirements for code-related tasks while maintaining high model quality.

### 9.1.2 Draft-Based Reasoning with RL Selection

Another promising direction is to explore more concise reasoning approaches inspired by human problem-solving strategies. While Chain-of-Thought (CoT) prompting has shown effectiveness for code-related tasks Wei et al. [2022], Chen et al. [2022], Li et al. [2025], Yin et al. [2024], Le et al., Paranjape et al. [2023], it often produces excessively verbose reasoning steps, consuming substantial tokens before arriving at a final answer. This ver-

bosity increases latency, token cost, and energy consumption, limiting the practicality of CoT-based methods in large-scale or resource-constrained settings.

Future work could adapt the Chain-of-Draft (CoD) approach Xu et al. [2025] to code generation tasks. Unlike CoT, which emphasizes verbose reasoning, CoD constrains each reasoning step to be succinct, promoting clarity and modularity. However, due to the inherent stochasticity of LLM outputs, naively applying CoD for code generation tasks may lead to suboptimal or inconsistent results. Even when guided by the same prompt, LLMs can produce multiple drafts that differ significantly in reasoning structure, abstraction level, or implementation strategy.

To address this, we propose developing a reinforcement learning-based framework that learns to select the best candidate from a set of CoD-generated solutions. By modeling solution selection as a contextual bandit problem, this approach would train a policy agent that evaluates candidates based on interpretable features such as code complexity, reasoning structure, and strategic metadata. This would enable harnessing the diversity of CoD while mitigating its stochasticity, leading to more robust and efficient code generation.

Key components of this approach would include:

- A strategy-guided prompting mechanism that encourages diverse reasoning styles Li et al. [2022a], Paranjape et al. [2023]
- A reward function that balances correctness, efficiency, and reasoning clarity
- A selection policy that learns to identify the most promising solutions without requiring ground truth references

This approach could significantly reduce token usage while maintaining or improving performance on code generation tasks, making it more suitable for practical deployment in software engineering workflows.

## 9.2 Automatic Coding Advancements

While significant progress has been made in using LLMs for code-related tasks, there remain substantial challenges in fully automating the software development process. Future work should focus on enhancing LLMs' capabilities to handle more realistic and complex coding scenarios.

### 9.2.1 Comprehensive Benchmark for Coding from Scratch

Current benchmarks for evaluating code generation capabilities often focus on algorithmic problem-solving (like HumanEval Chen et al. [2021a] and MBPP Austin et al. [2021]) or bug fixing (like SWE-bench Jimenez et al. [2024a] and BigCodeBench Lai et al. [2023]). However, these benchmarks fall short of assessing LLMs' ability to handle realistic software engineering tasks, particularly those involving feature integration and codebase extension.

A promising direction for future work is to develop a comprehensive benchmark specifically designed to evaluate the capacity of LLMs to perform realistic software engineering tasks involving feature addition. This benchmark would include tasks derived from open-source GitHub issues requesting new functionality, paired with the relevant code context and ground truth implementation.

Key aspects of this benchmark would include:

- **Feature Integration via Natural Language:** Tasks requiring models to interpret natural language requirements and integrate new functionality into an existing codebase, often requiring modifications to multiple components Li et al. [2022a].
- **Complex Instructions and Contexts:** Tasks involving conditional logic, API integration, or multi-step workflows that depend on an accurate understanding of the

surrounding code structure.

- **Diverse Domains and Complexity Levels:** Tasks from various domains such as data processing, web development, and machine learning, with varying levels of difficulty to provide a nuanced evaluation of model capabilities.

This benchmark would enable more rigorous evaluation of LLMs' capabilities in realistic software engineering scenarios and drive progress toward more practical code generation systems.

## 9.2.2 Structural Routing for Coding and Debugging

Software development is an iterative process involving multiple phases, including requirement analysis, design, implementation, testing, and debugging. Current approaches to automatic coding often treat these as separate tasks, leading to inefficiencies and integration challenges Yin et al. [2024], Zhang et al. [2024a], Yang et al. [2024c]. Future work should explore structured approaches that explicitly model the dependencies between these phases.

A promising direction is to develop a structural routing framework for coding and debugging that guides LLMs through the software development process in a systematic manner. This framework would build on recent advances in modular and structured reasoning Li et al. [2025] to improve the effectiveness of code generation and repair.

- **Decompose Complex Tasks:** Break down complex software engineering tasks into manageable sub-tasks with explicit dependencies and information flow.
- **Route Information Across Phases:** Ensure that insights and decisions from earlier phases (e.g., requirement analysis) are effectively utilized in later phases (e.g., implementation and testing).
- **Iterative Refinement:** Enable iterative improvement of solutions based on feedback from testing and verification steps, mirroring human software development practices Chen et al. [2023b].

This approach would be particularly valuable for repository-wide modifications, where changes in one component may have cascading effects on other parts of the codebase Yang et al. [2024c], Zhang et al. [2024a]. By explicitly modeling these dependencies and guiding the LLM through a structured development process, this framework could significantly improve the reliability and maintainability of automatically generated code.

Recent work on structured reasoning for program synthesis Chen et al. [2022], Li et al. [2025] and iterative refinement in code generation Chen et al. [2023b] provides promising foundations for developing such frameworks.

## 9.3 Integration of Future Directions

While we have presented efficient reasoning and automatic coding as separate research directions, they are highly complementary and could be integrated to achieve greater advancements in automated software engineering.

The efficient reasoning strategies described in Section 9.1 would provide the foundation for more scalable and resource-efficient models, which is essential for tackling the complex, multi-step reasoning required in realistic software engineering tasks Wei et al. [2022], Yin et al. [2024]. Meanwhile, the advancements in automatic coding outlined in Section 9.2 would drive the development of more sophisticated evaluation frameworks and methodologies, pushing the boundaries of what LLMs can achieve in software development Chen et al. [2021a].

Together, these research directions address both the computational challenges and the capability limitations of current approaches to automated software engineering. By pursuing

these complementary paths, future research can work toward a new generation of AI-assisted development tools that are both more powerful and more practically deployable in real-world software engineering workflows.



# Research Activities

We finally present the research activities undertaken during my Ph.D. journey. Specifically, we list: 1) the papers to which we contributed; 2) the tools and datasets produced through our research; 3) the venues where I served.

## List of Papers

### Papers included in this dissertation:

- [TOSEM'25] **Xunzhu Tang**, Kisub Kim, Saad Ezzini, Yewei Song, Haoye Tian, Jacques Klein, and Tegawendé F. Bissyandé. Just-in-Time Detection of Silent Security Patches ( Tang et al. [2025b]). In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2025. (Accepted)
- [EMSE'25] **Xunzhu Tang**, Haoye Tian, Weiguo Pian, Saad Ezzini, Abdoul Kader Kaboré, Andrew Habib, Kisub Kim, Jacques Klein, and Tegawendé F. Bissyandé. Learning to Represent Code Changes. *Empirical Software Engineering (EMSE)* (2025): minor revision (under review).
- [EMNLP'24] **Xunzhu Tang**, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawendé F. Bissyandé. CodeAgent: Autonomous Communicative Agents for Code Review ( Tang et al. [2024a]). In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11279-11313, 2024.
- [ACL'25 Findings] **Xunzhu Tang**, Jiechao Gao, Jin Xu, Tiezhu Sun, Yewei Song, Saad Ezzini, Wendkûuni C. Ouédraogo, Jacques Klein, and Tegawendé F. Bissyandé. SynFix: Synchronous Repair for Codebase via RelationGraph ( Tang et al. [2025a]). In *Findings of the Association for Computational Linguistics (ACL 2025)*, 2025. (Accepted)
- [TOSEM] **Xunzhu Tang**, Jacques Klein, and Tegawendé F. Bissyandé. Boosting Open-Source Code LLMs for Program Repair via Reasoning Transfer and LLM-Guided Reinforcement Learning ( Tang et al. [2025c]). *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2025. (Under review)

### Papers not included in this dissertation:

- [ICSME'25] **Xunzhu Tang**, Tiezhu Sun, Yewei Song, Siyuan Ma, Jacques Klein and Tegawendé Bissyandé, ExpertCache: GPU-Efficient MoE Inference through Reinforcement Learning-Guided Expert Selection. In *The 41st International Conference on Software Maintenance and Evolution*, NIER track, 2025.
- [ASE'25] Tiezhu Sun, Marco Alecci, Yewei Song, **Xunzhu Tang**, Kisub Kim, Jor-

dan Samhi, Tegawendé F. Bissyandé, Jacques Klein, RAML: Toward Retrieval-Augmented Localization of Malicious Payloads in Android Apps ( S. et al. [2025]). In *The 40th IEEE/ACM International Conference on Automated Software Engineering*, NIER track, 2025.

- [ASE'25] Yewei Song, Tiezhu Sun, **Xunzhu Tang**, Prateek Kumar Rajput, Tegawendé F. Bissyandé and Jacques Klein, Measuring LLM Code Generation Stability via Structural Entropy. In *The 40th IEEE/ACM International Conference on Automated Software Engineering*, NIER track, 2025.
- [ICSME'25 NIER] Tiezhu Sun, Marco Alecci, Aleksandr Pilgun, Yewei Song, **Xunzhu Tang**, Jordan Samhi, Tegawendé F. Bissyandé and Jacques Klein, MalLoc: Towards Fine-grained Android Malicious Payload Localization via LLMs. In *The 41st International Conference on Software Maintenance and Evolution*, 2025.
- [EASE'25] Yewei Song, **Xunzhu Tang**, Cedric Lothritz, Saad Ezzini, Jacques Klein, Tegawendé Bissyandé, Andrey Boytsov, Ulrick Ble and Anne Goujon. CallNavi, A challenge and empirical study on LLM function calling and routing ( Song et al. [2025]). In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2025.
- [TOSEM'24] Weiguo Pian, Yinghua Li, Haoye Tian, Tiezhu Sun, Yewei Song, **Xunzhu Tang**, Andrew Habib, Jacques Klein, Tegawendé F. Bissyandé. You Don't Have to Say Where to Edit! Joint Learning to Localize and Edit Source Code ( Pian et al. [2025]). In *ACM Transactions on Software Engineering and Methodology*, 2024.
- [ASE'24] Kisub Kim, Jounghoon Kim, Byeongjo Park, Dongsun Kim, Chun Yong Chong, Yuan Wang, Tiezhu Sun, **Xunzhu Tang**, Jacques Klein, Tegawendé F Bissyandé. DataRecipe — How to Cook the Data for CodeLLM? ( Kim et al. [2024]) In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024.
- [ASE'24-JF/EMSE] **Xunzhu Tang**, Haoye Tian, Pingfan Kong, Saad Ezzini, Kui Liu, Xin Xia, Jacques Klein, Tegawendé F Bissyandé. App Review Driven Collaborative Bug Finding ( Tang et al. [2024c]). In *Empirical Software Engineering*, 2024.
- [ICSE'25] Gong Chen, Xiaoyuan Xie, **Xunzhu Tang**, Qi Xin, Wenjie Liu. Hedge-Code: A Multi-Task Hedging Contrastive Learning Framework for Code Search ( Chen et al. [2024b]). In *Proceedings of the 47th International Conference on Software Engineering*, 2025.
- [ACL'24] Yewei Song, Cedric Lothritz, **Xunzhu Tang**, Tegawende F. Bissyandé, Jacques Klein. Revisiting Code Similarity Evaluation with Abstract Syntax Tree Edit Distance (Song et al. [2024b]). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.
- [ICSE'24] **Xunzhu Tang**, Haoye Tian, Zhenghan Chen, Weiguo Pian, Saad Ezzini, Abdoul Kader Kabore, Andrew Habib, Jacques Klein, Tegawende F. Bissyandé. Learning to Represent Patches (Tang et al. [2024b]). In *Proceedings of the 46th International Conference on Software Engineering (POSTER)*, 2024.
- [ICSE'24-SEIP] Yewei Song, Saad Ezzini, **Xunzhu Tang**, Cedric Lothritz, Jacques Klein, Tegawende Bissyandé, Andrey Boytsov, Ulrick Ble, Anne Goujon. Enhancing Text-to-SQL Translation for Financial System Design (Song et al. [2024a]). In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, 2024.
- [NAACL'24] Dongsheng Zhu, **Xunzhu Tang**, Weidong Han, Jinghui Lu, Yukun Zhao, Guoliang Xing, Junfeng Wang, Dawei Yin. VisLingInstruct: Elevating Zero-Shot

Learning in Multi-Modal Language Models with Autonomous Instruction Optimization (Zhu et al. [2024]). In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics*, 2024.

- [ICLR’24] **Xunzhu Tang**, Zhenghan Chen, Saad Ezzini, Haoye Tian, Yewei Song, Jacques Klein, Tegawendé F. Bissyande. Multilevel Semantic Embedding of Software Patches: A Fine-to-Coarse Grained Approach Towards Security Patch Detection. ( Tang et al. [2023b]) In *The Twelfth International Conference on Learning Representations (Tiny Track)*, 2024.
- [ASE’23] Liran Wang, **Xunzhu Tang**, Yichen He, Changyu Ren, Shuhua Shi, Chao-ran Yan, Zhoujun Li. Delving into Commit-Issue Correlation to Enhance Commit Message Generation Models ( Wang et al. [2023a]). In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, 2023.
- [AAAI’23] Weiguo Pian, Hanyu Peng, **Xunzhu Tang**, Tiezhu Sun, Haoye Tian, Andrew Habib, Jacques Klein, Tegawendé F Bissyandé. MetaTPTrans: A Meta Learning Approach for Multilingual Code Representation Learning ( Pian et al. [2023a]). In *Proceedings of the 37th AAAI Conference on Artificial Intelligence*, 2023.

## Tools and Datasets

- **Repairity**: <https://anonymous.4open.science/r/REPARITY-9BE1>
- **CodeAgent**: <https://github.com/Daniel4SE/codeagent>
- **Patcherizer**: <https://anonymous.4open.science/r/Patcherizer-1E04/README.md>

## Services

- **Program Committee**: ACL’23, EMNLP’22, NLPCC’22, SOSP’21 Artifact Committee
- **Journal Referee**: ASE Journal’25
- **Conference Referee**: ACL’25, EMNLP’25, ACM MM’25, ICONIP’23, NLPCC’23, EMNLP’23, ICSE’22, NLPCC’21



---

# Bibliography

Networkx. In Reda Alhajj and Jon G. Rokne, editors, *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*. Springer, 2018. doi:10.1007/978-1-4939-7131-2\_100771. URL [https://doi.org/10.1007/978-1-4939-7131-2\\_100771](https://doi.org/10.1007/978-1-4939-7131-2_100771).

Amazon q developer the most capable generative ai-powered assistant for software development. <https://aws.amazon.com/q/developer/>, 2024.

Appmap speedruns to the top of the swe bench leaderboard. <https://appmap.io/blog/2024/06/20/appmap-navie-swe-bench-leader/>, 2024.

Autocoderover autonomous software engineering. <https://autocoderover.dev/>, 2024.

Aide by codestory. [https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240702\\_codestory\\_aide\\_mixed](https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240702_codestory_aide_mixed), 2024.

Factory bringing autonomy to software engineering. <https://www.factory.ai/>, 2024.

The road to ultimate pull request machine. <https://gru.ai/blog/road-to-ultimate-pull-request-machine/>, 2024.

Honeycomb. <https://honeycomb.sh>, 2024.

Agent-101: A software engineering agent for code assistance developed by ibm research. [https://github.com/swe-bench/experiments/blob/main/evaluation/lite/20240612\\_IBM\\_Research\\_Agent101/README.md/](https://github.com/swe-bench/experiments/blob/main/evaluation/lite/20240612_IBM_Research_Agent101/README.md), 2024.

Isoform. [https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240829\\_Isoform](https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240829_Isoform), 2024.

Lingma agent. [https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240622\\_Lingma\\_Agent](https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240622_Lingma_Agent), 2024.

Mentatbot: New sota coding agent, available now. <https://mentat.ai/blog/mentatbot-sota-coding-agent>, 2024.

Moatless tools. <https://github.com/aorwall/moatless-tools>, 2024.

Opencsg starship. <https://opencsg.com/product?class=StarShip/>, 2024a.

## Bibliography

---

Opendedvin: Code less, make more. <https://github.com/OpenDevin/OpenDevin/>, 2024b.

Regraph: Enhancing ai software engineering with repository-level code graph. <https://github.com/ozyyshr/RepoGraph>, 2024.

Alex sima. [https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240706\\_sima\\_gpt4o](https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240706_sima_gpt4o), 2024.

Supercoder. <https://superagi.com/supercoder/>, 2024.

Swe-bench lite. <https://www.swebench.com/lite.html>, 2024.

Meta AI. Llama3 70b: Large language model for multimodal applications. <https://ai.meta.com/llama3-70b>, 2023.

Elif Akata, Lion Schulz, Julian Coda-Forno, Seong Joon Oh, Matthias Bethge, and Eric Schulz. Playing repeated games with large language models. *arXiv preprint*, 2023.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *PACMPL*, 3(POPL):40:1–40:29, 2019. doi:10.1145/3290353. URL <https://doi.org/10.1145/3290353>.

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 245–256. PMLR, 2020. URL <http://proceedings.mlr.press/v119/alon20a.html>.

Anthropic. Claude: A new ai assistant from anthropic. Retrieved from <https://www.anthropic.com/clause>, 2023.

Anthropic. Claude 3.5 sonnet: High-performance language model for complex tasks. <https://www.anthropic.com/clause-3-5-sonnet>, 2023.

Anthropic. Claude 3.7 sonnet: Hybrid reasoning ai. <https://www.anthropic.com/news/clause-3-7-sonnet>, 2025. Accessed: March 14, 2025.

Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. Masai: Modular architecture for software-engineering ai agents. *arXiv preprint arXiv:2406.11638*, 2024.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. In *arXiv preprint arXiv:2108.07732*, 2021.

Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, 2013.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005.

Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–317, 2014.

Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th working conference on mining software repositories*, pages 202–211, 2014.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Amiangshu Bosu and Jeffrey C Carver. Impact of peer code review on peer impression formation: A survey. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 133–142. IEEE, 2013.

Larissa Braz, Christian Aeberhard, Gülcük Çalikli, and Alberto Bacchelli. Less is more: supporting developers in vulnerability detection during code review. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1317–1329, 2022.

Shaked Brody, Uri Alon, and Eran Yahav. A structural model for contextual code changes. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Tuan-Dung Bui, Duc-Thieu Luu-Van, Thanh-Phat Nguyen, Thu-Trang Nguyen, Son Nguyen, and Hieu Dinh Vo. Rambo: Enhancing rag-based repository-level method body completion. *arXiv preprint arXiv:2409.15204*, 2024.

Raymond PL Buse and Westley R Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 33–42, 2010.

Rocío Cabrera Lozoya, Arnaud Baumann, Antonino Sabetta, and Michele Bezzi. Commit2vec: Learning distributed representations of code changes. *SN Computer Science*, 2(3):1–16, 2021.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. *arXiv preprint*, 2023.

Hyungjoo Chae, Yongho Song, Kai Tzu-iunn Ong, Taeyoon Kwon, Minjin Kim, Youngjae Yu, Dongha Lee, Dongyeop Kang, and Jinyoung Yeo. Dialogue chain-of-thought distillation for commonsense-aware conversational agents. *arXiv preprint arXiv:2310.09343*, 2023.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296, 2021.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. Coder: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*, 2024a.

Gong Chen, Xiaoyuan Xie, Daniel Tang, Qi Xin, and Wenjie Liu. Hedgecode: A multi-task hedging contrastive learning framework for code search. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 89–100. IEEE Computer Society, 2024b.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. In *Advances in Neural Information Processing Systems*, 2021a.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021b.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan, Yujia Qin, Yaxi Lu, Ruobing Xie, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *arXiv preprint arXiv:2308.10848*, 2023a.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023b.

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems*, pages 4299–4307, 2017.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *Journal of Machine Learning Research*, 25(70):1–53, 2024.

Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. An empirical study on the usage of bert models for code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 108–119. IEEE, 2021.

Aidan Clark, Diego de Las Casas, Aurelia Guy, Arthur Mensch, Michela Paganini, Jordan Hoffmann, Bogdan Damoc, Blake Hechtman, Trevor Cai, Sebastian Borgeaud, et al. Unified scaling laws for routed language models. In *International conference on machine learning*, pages 4057–4086. PMLR, 2022.

Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. Performance evaluation of the vf graph matching algorithm. In *Proceedings 10th International Conference on Image Analysis and Processing*, pages 1172–1177. IEEE, 1999.

Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. On automatically generating commit messages via summarization of source code changes. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 275–284. IEEE, 2014.

Wenliang Dai, Junnan Li, Dongxu Li, Anthony Meng Huat Tiong, Junqi Zhao, Weisheng Wang, Boyang Li, Pascale Fung, and Steven Hoi. Instructblip: Towards general-purpose vision-language models with instruction tuning, 2023.

Amirhossein Dakhel, Majid Majdinasab, Amin Nikanjam, and Foutse Khomh. Github copilot ai pair programmer: Asset or liability? In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, pages 385–396, 2023.

Nicole Davila and Ingrid Nunes. A systematic literature review and taxonomy of modern code review. *Journal of Systems and Software*, 177:110951, 2021.

Daniel DeFreez, Aditya V Thakur, and Cindy Rubio-González. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 423–433, 2018.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Victor Dibia, Adam Fournier, Gagan Bansal, Forough Poursabzi-Sangdeh, Han Liu, and Saleema Amershi. Aligning offline metrics and human judgments of value of ai-pair programmers. *arXiv preprint arXiv:2210.16494*, 2022.

Nesara Dissanayake, Mansooreh Zahedi, Asangi Jayatilaka, and Muhammad Ali Babar. Why, how and where of delays in software security patch management: An empirical investigation in the healthcare sector. *Proceedings of the ACM on Human-Computer Interaction*, 6(CSCW2):1–29, 2022.

Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. Fira: Fine-grained graph-based code change representation for automated commit message generation. 2022.

## Bibliography

---

Rotem Dror, Gal Baumer, Segev Shlomov, and Roi Reichart. The hitchhiker’s guide to testing statistical significance in natural language processing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 1383–1392, 2018.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*, 2023.

Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 422–431. IEEE, 2013.

Ahmed Elgohary, Christopher Meek, Matthew Richardson, Adam Journey, Gonzalo Ramos, and Ahmed Hassan Awadallah. NL-EDIT: Correcting semantic parse errors through natural language interaction. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5599–5610, Online, June 2021. Association for Computational Linguistics. doi:10.18653/v1/2021.naacl-main.444. URL <https://aclanthology.org/2021.naacl-main.444>.

Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. Codetrans: Towards cracking the language of silicon’s code through self-supervised deep learning and high performance computing. *arXiv preprint arXiv:2104.02443*, 2021.

Hugging Face. Mixtral 8x7b: Multi-domain pre-trained model for code and text. <https://huggingface.co/mixtral-8x7b>, 2023.

Zhiwei Fan, Xinyuan Li, Han Yu, Yuan Xu, Chang Liu, Ting-Yi Liu, and Lingming Wang. Automated program repair with large language models. *arXiv preprint arXiv:2307.07359*, 2023.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2022. URL <https://arxiv.org/abs/2101.03961>.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Dixin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics, 2020a. doi:10.18653/v1/2020.findings-emnlp.139. URL <https://doi.org/10.18653/v1/2020.findings-emnlp.139>.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Dixin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020b.

Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–27, 2021.

Paul Gauthier. Aider is ai pair programming in your terminal. <https://aider.chat/>, 2024.

Ali Ghanbari and Andrian Marcus. Patch correctness assessment in automated program repair based on the impact of patches on production and test code, 2022.

Matthías Páll Gissurarson, Leonhard Applis, Annibale Panichella, Arie van Deursen, and David Sands. Propr: property-based automatic program repair. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1768–1780, 2022.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

Cobus Greyling. Prompt drift and chaining, May 2023. URL <https://blog.humanfirst.ai/prompt-drift/>.

Arnav Gudibande, Eric Wallace, Charlie Snell, Xinyang Geng, Hao Liu, Pieter Abbeel, Sergey Levine, and Dawn Song. The false promise of imitating proprietary llms. *arXiv preprint arXiv:2305.15717*, 2023.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Dixin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021a. URL <https://openreview.net/forum?id=jLoC4ez43PZ>.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Dixin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021b. URL <https://openreview.net/forum?id=jLoC4ez43PZ>.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

DongGyun Han, Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, and Giovanni Rosa. Does code review really remove coding convention violations? In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 43–53. IEEE, 2020.

## Bibliography

---

Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Vincent J Hellendoorn, Jason Tsay, Manisha Mukherjee, and Martin Hirzel. Towards automating code review at scale. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1479–1482, 2021.

Jordan Henkel, Shuvendu K Lahiri, Ben Liblit, and Thomas Reps. Code vectors: Understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 163–174, 2018.

Abram Hindle, Daniel M German, and Richard Holt. Software process recovery using recovered unified process views. In *2009 IEEE International Conference on Software Maintenance*, pages 285–294. IEEE, 2009.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 34–45. IEEE, 2019.

Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 518–529, 2020.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

Yang Hong, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, and Aldeida Aleti. Commentfinder: a simpler, faster, more accurate code review comments recommendation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 507–519, 2022.

Mohammad Hossin and Md Nasir Sulaiman. A review on evaluation metrics for data classification evaluations. *International journal of data mining & knowledge management process*, 5(2):1, 2015.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2021.

Yaojie Hu, Qiang Zhou, Qihong Chen, Xiaopeng Li, Linbo Liu, Dejiao Zhang, Amit Kachroo, Talha Oz, and Omer Tripp. Qualityflow: An agentic workflow for program synthesis controlled by llm quality checks. *arXiv preprint arXiv:2501.17167*, 2025.

Yuan Huang, Nan Jia, Hao-Jie Zhou, Xiang-Ping Chen, Zi-Bin Zheng, and Ming-Dong Tang. Learning human-written commit messages to document code changes. *Journal of Computer Science and Technology*, 35(6):1258–1277, 2020.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*, 2024.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Marie-Anne Lachaux, Naiming Gu, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021.

Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 135–146. IEEE, 2017.

Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4163–4174, 2020.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024a. URL <https://openreview.net/forum?id=VTF8yNQM66>.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench leaderboard. <https://www.swebench.com/>, 2024b.

Alexander Jimenez-Sanchez, Alex Fargher, Anton Axelsson, Amanda Felländer, James Watten, Oscar Rudberg, Arvid Kahn, Ariel Calota, Daniel Gillblad, Anders Holst, et al. Swebench: Benchmarking large language models for swedish. In *arXiv preprint arXiv:2312.16639*, 2023.

Chen Jin, Wenda Lee, Sruti Patel, Jillian Li, Nitya Mathur, Abhinav Mehrotra, Aniruddha Sathish, Aditya Kanade, Dokyun Lee, and Ce Min. Program of thoughts: Towards reasoning scaffolds for program synthesis. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 152–164, 2023.

René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2012.

Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016.

Bowen Kim, Daun Kim, Kihyuk Lee, Seong Joon Hwang, and Jinwoo Choi. Sequence-level knowledge distillation for dense prediction tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7259–7268, 2021.

Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 802–811. IEEE, 2013.

Kisub Kim, Jounghoon Kim, Byeongjo Park, Dongsun Kim, Chun Yong Chong, Yuan Wang, Tiezhu Sun, Daniel Tang, Jacques Klein, and Tegawendé F Bissyandé. Datarecipe—how to cook the data for codellm? In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1206–1218, 2024.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916*, 2022.

Gemma AI Labs. Gemma 7b: Lightweight model for language and code tasks. <https://gemmalabs.ai/models/7b>, 2023.

Wei Lai, Zekun Tian, Hanmer Yu, Xuangui Yan, Xia Chen, and Pengfei Yin. Bigcodebench: A systematic evaluation of large language models for code generation. In *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

Salar Latifi, Saurav Muralidharan, and Michael Garland. Efficient sparsely activated transformers. *arXiv preprint arXiv:2208.14580*, 2022.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. In *The Twelfth International Conference on Learning Representations*.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

Harrison Lee, Buck Shlegeris, Eric Chan, Roger Grosse, and John X Morris. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. *arXiv preprint arXiv:2309.00267*, 2023.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL <https://openreview.net/forum?id=qrwe7XHTmYb>.

Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbulin, and Bernard Ghanem. Camel: Communicative agents for "mind" exploration of large scale language model society. *arXiv preprint arXiv:2303.17760*, 2023a.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–23, 2025.

Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. In *International conference on machine learning*, pages 19730–19742. PMLR, 2023b.

Min Li, Zhenjiang Miao, Xiao-Ping Zhang, and Wanru Xu. An attention-seq2seq model based on crnn encoding for automatic labanotation generation from motion capture data. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4185–4189. IEEE, 2021.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023c.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you!, 2023d.

Yuan Li, Yixuan Zhang, and Lichao Sun. Metaagents: Simulating interactions of human behaviors for llm-based task-oriented coordination via collaborative generative agents. *arXiv preprint arXiv:2310.06500*, 2023e.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittweiser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 2022a.

Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Codereviewer: Pre-training for automating code review activities. *arXiv e-prints*, pages arXiv–2203, 2022b.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Zhaopeng Tu, and Shuming Shi. Encouraging divergent thinking in large language models through multi-agent debate. *arXiv preprint arXiv:2305.19118*, 2023.

Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. Context-aware code change embedding for better patch correctness assessment. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–29, 2022.

Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. Cct5: A code-change-oriented pre-trained model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1509–1521, 2023.

Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. Changescrbe: A tool for automatically generating commit messages. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 709–712. IEEE, 2015.

Enshu Liu, Junyi Zhu, Zinan Lin, Xuefei Ning, Matthew B Blaschko, Shengen Yan, Guohao Dai, Huazhong Yang, and Yu Wang. Efficient expert pruning for sparse mixture-of-experts language models: Enhancing performance and reducing inference costs. *arXiv preprint arXiv:2407.00945*, 2024a.

Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. A self-attentional neural architecture for code completion with multi-task learning. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 37–47, 2020a.

Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 473–485, 2020b.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024b.

Qin Liu, Zihe Liu, Hongming Zhu, Hongfei Fan, Bowen Du, and Yu Qian. Generating commit messages from diffs using pointer-generator network. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 299–309. IEEE, 2019.

Shangqing Liu, Cuiyun Gao, Sen Chen, Nie Lun Yiu, and Yang Liu. Atom: Commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Transactions on Software Engineering*, 2020c.

Yizhou Liu, Pengfei Gao, Xincheng Wang, Chao Peng, and Zhao Zhang. Marscode agent: Ai-native automated bug fixing. *arXiv preprint arXiv:2409.00899*, 2024c.

Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-machine-translation-based commit message generation: how far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 373–384, 2018.

Zhongxin Liu, Zhijie Tang, Xin Xia, and Xiaohu Yang. Ccrep: Learning code change representations via pre-trained code model and query back. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 17–29. IEEE, 2023.

Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. 2018.

Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 647–658. IEEE, 2023.

Chunjie Luo, Jianfeng Zhan, Xiaohe Xue, Lei Wang, Rui Ren, and Qiang Yang. Cosine normalization: Using cosine similarity instead of dot product in neural networks. In *International Conference on Artificial Neural Networks*, pages 382–391. Springer, 2018.

Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. How to understand whole software repository? *arXiv preprint arXiv:2406.01422*, 2024.

Martin Monperrus. Automatic software repair: A bibliography. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

Lun Yiu Nie, Cuiyun Gao, Zhicong Zhong, Wai Lam, Yang Liu, and Zenglin Xu. Coregen: Contextualized code representation learning for commit message generation. *Neurocomputing*, 459:97–107, 2021.

Michael Niemeyer and Andreas Geiger. Giraffe: Representing scenes as compositional generative neural feature fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11453–11464, 2021.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *International Conference on Learning Representations (ICLR)*, 2023.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*, 2023.

Delano Oliveira, Reydne Santos, Fernanda Madeiral, Hidehiko Masuhara, and Fernando Castor. A systematic literature review on the impact of formatting elements on code legibility. *Journal of Systems and Software*, 203:111728, 2023.

OpenAI. Chatgpt: Optimizing language models for dialogue. <https://openai.com/blog/chatgpt/>, 2022.

OpenAI. Gemini 2.0 flash: Advanced language model for code and text understanding. <https://www.openai.com/gemini-2-flash>, 2023.

OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

OpenAI. Openai o3-mini system card, 2025. URL <https://openai.com/index/o3-mini-system-card/>.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744, 2022.

Sheena Panthapuckle, Junyi Jessy Li, Milos Gligoric, and Raymond J Mooney. Deep just-in-time inconsistency detection between comments and source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 427–435, 2021.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023.

Joon Sung Park, Joseph O’Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pages 1–22, 2023.

Weiguo Pian, Hanyu Peng, Xunzhu Tang, Tiezhu Sun, Haoye Tian, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. Metatptrans: A meta learning approach for multilingual code representation learning. *arXiv preprint arXiv:2206.06460*, 2022.

Weiguo Pian, Hanyu Peng, Xunzhu Tang, Tiezhu Sun, Haoye Tian, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. Metatptrans: A meta learning approach for multilingual code representation learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 5239–5247, 2023a.

Weiguo Pian, Hanyu Peng, Xunzhu Tang, Tiezhu Sun, Haoye Tian, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. Metatptrans: A meta learning approach for multilingual code representation learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 5239–5247, 2023b.

Weiguo Pian, Yinghua Li, Haoye Tian, Tiezhu Sun, Yewei Song, Xunzhu Tang, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. You don’t have to say where to edit! jled–joint learning to localize and edit source code. *ACM Transactions on Software Engineering and Methodology*, 2025.

Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, 2015.

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.

Libo Qin, Tailu Liu, Wanxiang Che, Bingbing Kang, Sendong Zhao, and Ting Liu. A co-interactive transformer for joint slot filling and intent detection. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8193–8197. IEEE, 2021.

Qwen Team. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3505–3506, 2020.

Lin CY ROUGE. A package for automatic evaluation of summaries. In *Proceedings of Workshop on Text Summarization of ACL, Spain*, 2004.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Tan, Yossi Liu, Rémi Lebret, Rob Fergus, and Yann LeCun. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. Specrover: Code intent extraction via llms. *arXiv preprint arXiv:2408.02232*, 2024.

Saraga S., Anagha M. S., Dincy R. Arikat, Rafidha Rehman K. A., Serena Nicolazzo, Antonino Nocera, and Vinod P. Enhancing android malware detection with retrieval-augmented generation, 2025. URL <https://arxiv.org/abs/2506.22750>.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Abigail See, Peter J Liu, and Christopher D Manning. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368*, 2017.

Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Concolic program repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 390–405, 2021.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In Marilyn A. Walker, Heng Ji, and Amanda Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, pages 464–468. Association for Computational Linguistics, 2018. doi:10.18653/v1/n18-2074. URL <https://doi.org/10.18653/v1/n18-2074>.

Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. CAST: enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 4053–4062. Association for Computational Linguistics, 2021. doi:10.18653/v1/2021.emnlp-main.332. URL <https://doi.org/10.18653/v1/2021.emnlp-main.332>.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*, pages 31210–31227. PMLR, 2023.

Jing Kai Siow, Cuiyun Gao, Lingling Fan, Sen Chen, and Yang Liu. Core: Automating review recommendation for code changes. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 284–295. IEEE, 2020.

Lindsay I Smith. A tutorial on principal components analysis. 2002.

Yewei Song, Saad Ezzini, Xunzhu Tang, Cedric Lothritz, Jacques Klein, Tegawendé Bissyandé, Andrey Boytsov, Ulrick Ble, and Anne Goujon. Enhancing text-to-sql translation for financial system design. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, pages 252–262, 2024a.

Yewei Song, Cedric Lothritz, Daniel Tang, Tegawendé F Bissyandé, and Jacques Klein. Revisiting code similarity evaluation with abstract syntax tree edit distance. *arXiv preprint arXiv:2404.08817*, 2024b.

Yewei Song, Xunzhu Tang, Cedric Lothritz, Saad Ezzini, Jacques Klein, Tegawendé F Bissyandé, Andrey Boytsov, Ulrick Ble, and Anne Goujon. Callnavi, a challenge and empirical study on llm function calling and routing. *arXiv preprint arXiv:2501.05255*, 2025.

Miroslaw Staron, Mirosław Ochodek, Wilhelm Meding, and Ola Söder. Using machine learning to identify code fragments for manual review. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 513–516. IEEE, 2020.

Nisan Stiennon, Long Ouyang, Jeff Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize from human feedback. In *Advances in Neural Information Processing Systems*, 2020.

Chia-Yi Su and Collin McMillan. Semantic similarity loss for neural source code summarization. *arXiv preprint arXiv:2308.07429*, 2023.

Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, et al. Automatic code summarization via chatgpt: How far are we? *arXiv preprint arXiv:2305.12865*, 2023.

Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2158–2170, 2020.

Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2727–2735, 2019.

Yashar Talebirad and Amirhossein Nadiri. Multi-agent collaboration: Harnessing the power of intelligent llm agents, 2023.

Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3282–3299, 2021.

Xunzhu Tang, Rujie Zhu, Tiezhu Sun, and Shi Wang. Moto: Enhancing embedding with multiple joint factors for chinese text classification. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 2882–2888. IEEE, 2021.

Xunzhu Tang, Zhenghan Chen, Kisub Kim, Haoye Tian, Saad Ezzini, and Jacques Klein. Just-in-time security patch detection–llm at the rescue for data augmentation. *arXiv preprint arXiv:2312.01241*, 2023a.

Xunzhu Tang, Saad Ezzini, Haoye Tian, Yewei Song, Jacques Klein, Tegawende F Bissyande, et al. Multilevel semantic embedding of software patches: a fine-to-coarse grained approach towards security patch detection. *arXiv preprint arXiv:2308.15233*, 2023b.

Xunzhu Tang, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawendé F. Bissyandé. CodeAgent: Autonomous communicative agents for code review. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11279–11313, Miami, Florida, USA, November 2024a. Association for Computational Linguistics. doi:10.18653/v1/2024.emnlp-main.632. URL <https://aclanthology.org/2024.emnlp-main.632/>.

Xunzhu Tang, Haoye Tian, Zhenghan Chen, Weiguo Pian, Saad Ezzini, Abdoul Kader Kaboré, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. Learning to represent patches. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 396–397, 2024b.

Xunzhu Tang, Haoye Tian, Pingfan Kong, Saad Ezzini, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F Bissyandé. App review driven collaborative bug finding. *Empirical Software Engineering*, 29(5):124, 2024c.

Xunzhu Tang, Jiechao Gao, Jin Xu, Tiezhu Sun, Yewei Song, Saad Ezzini, Wendkùuni C. Ouédraogo, Jacques Klein, and Tegawendé F. Bissyandé. SynFix: Dependency-aware program repair via RelationGraph analysis. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Findings of the Association for Computational Linguistics: ACL 2025*, pages 4878–4894, Vienna, Austria, July 2025a. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi:10.18653/v1/2025.findings-acl.252. URL <https://aclanthology.org/2025.findings-acl.252/>.

Xunzhu Tang, Kisub Kim, Saad Ezzini, Yewei Song, Haoye Tian, Jacques Klein, and Tegawende Bissyande. Just-in-time detection of silent security patches. *ACM Trans. Softw. Eng. Methodol.*, July 2025b. ISSN 1049-331X. doi:10.1145/3749370. URL <https://doi.org/10.1145/3749370>. Just Accepted.

Xunzhu Tang, Jacques Klein, and Tegawendé F Bissyandé. Boosting open-source llms for program repair via reasoning transfer and llm-guided reinforcement learning. *arXiv preprint arXiv:2506.03921*, 2025c.

Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 141–150. IEEE, 2015.

Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. Autotransform: Automated code transformation to support modern code review process. In *Proceedings of the 44th international conference on software engineering*, pages 237–248, 2022.

Chris Thunes. javalang: pure python java parser and tools, 2013.

Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F Bissyandé. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 981–992. IEEE, 2020.

Haoye Tian, Yinghua Li, Weiguo Pian, Abdoul Kader Kabore, Kui Liu, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. Predicting patch correctness based on the similarity of failing test cases. *ACM Transactions on Software Engineering and Methodology*, 2022a.

Haoye Tian, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F Bissyandé. The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches. *arXiv preprint arXiv:2203.08912*, 2022b.

Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F Bissyandé. Is this change the answer to that problem? correlating descriptions of bug and code changes for evaluating patch correctness. *arXiv preprint arXiv:2208.04125*, 2022c.

Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. Is chatgpt the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938*, 2023.

Yuan Tian, Julia Lawall, and David Lo. Identifying linux bug fixing patches. In *2012 34th international conference on software engineering (ICSE)*, pages 386–396. IEEE, 2012.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 163–174. IEEE, 2021.

Rosalia Tufano, Simone Masiero, Antonio Mastropaoletti, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2291–2302, 2022.

Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Ashwin K Vijayakumar, Michael Cogswell, Ramprasath R Selvaraju, Qing Sun, Stefan Lee, David Crandall, and Dhruv Batra. Diverse beam search: Decoding diverse solutions from neural sequence models. *arXiv preprint arXiv:1610.02424*, 2016.

Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. Context-aware retrieval-based deep commit message generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–30, 2021a.

Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163*, 2017.

Liran Wang, Xunzhu Tang, Yichen He, Changyu Ren, Shuhua Shi, Chaoran Yan, and Zhoujun Li. Delving into commit-issue correlation to enhance commit message generation models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 710–722. IEEE, 2023a.

Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019a.

Shi Wang, Daniel Tang, Luchen Zhang, Huilin Li, and Ding Han. Hienet: Bidirectional hierarchy framework for automated ICD coding. In Arnab Bhattacharya, Janice Lee, Mong Li, Divyakant Agrawal, P. Krishna Reddy, Mukesh K. Mohania, Anirban Mondal, Vikram Goyal, and Rage Uday Kiran, editors, *Database Systems for Advanced Applications - 27th International Conference, DASFAA 2022, Virtual Event, April 11-14, 2022, Proceedings, Part II*, volume 13246 of *Lecture Notes in Computer Science*, pages 523–539. Springer, 2022a. doi:10.1007/978-3-031-00126-0\_38. URL [https://doi.org/10.1007/978-3-031-00126-0\\_38](https://doi.org/10.1007/978-3-031-00126-0_38).

Shi Wang, Daniel Tang, Luchen Zhang, Huilin Li, and Ding Han. Hienet: Bidirectional hierarchy framework for automated icd coding. In *International Conference on Database Systems for Advanced Applications*, pages 523–539. Springer, 2022b.

Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. Graphspd: Graph-based security patch detection with enriched code semantics. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2409–2426. IEEE, 2023b.

Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. In *Advances in Neural Information Processing Systems*, volume 33, pages 5776–5788, 2020a.

Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. Detecting" 0-day" vulnerability: An empirical study of secret security patch in oss. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 485–492. IEEE, 2019b.

Xinda Wang, Shu Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. A machine learning approach to classify security patches into vulnerability types. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9. IEEE, 2020b.

Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. Patchdb: A large-scale security patch dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 149–160. IEEE, 2021b.

Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. Patchrnn: A deep learning-based system for security patch identification. In *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*, pages 595–600. IEEE, 2021c.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021d.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021e.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics, 2021f. doi:10.18653/v1/2021.emnlp-main.685. URL <https://doi.org/10.18653/v1/2021.emnlp-main.685>.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023c.

Zhenhailong Wang, Shaoguang Mao, Wenshan Wu, Tao Ge, Furu Wei, and Heng Ji. Unleashing cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration. *arXiv preprint arXiv:2307.05300*, 2023d.

Zhiruo Wang, Yunhao Bai, Dinglan Song, Pengfei Yin, Po-Sen Huang, Panupong Pasupat, Yiran Wang, Duen Horng Chau, Aditya Parameswaran, and Percy Liang. Codedistill: Learning to distill code generation tasks to large language models. *arXiv preprint arXiv:2312.00876*, 2023e.

Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–58, 2022.

Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

Jimmy Wei, Kurt Shuster, Arthur Szlam, Jason Weston, Jack Urbanek, and Mojtaba Komeili. Multi-party chat: Conversational agents in group settings with humans and models. *arXiv preprint*, 2023a.

Xi Wei, Tianzhu Zhang, Yan Li, Yongdong Zhang, and Feng Wu. Multi-modality cross attention network for image and sentence matching. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10941–10950, 2020.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*, 2023b.

Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 172–184, 2023c.

Frank Wilcoxon. Individual comparisons by ranking methods. In *Breakthroughs in statistics: Methodology and distribution*, pages 196–202. Springer, 1992.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. Commit message generation for source code changes. In *IJCAI*, 2019.

Silei Xu, Wenhao Xie, Lingxiao Zhao, and Pengcheng He. Chain of draft: Thinking faster by writing less. *arXiv preprint arXiv:2502.18600*, 2025.

Aidan ZH Yang, Haoye Tian, He Ye, Ruben Martins, and Claire Le Goues. Security vulnerability detection with multitask self-instructed fine-tuning of large language models. *arXiv preprint arXiv:2406.05892*, 2024a.

Boyang Yang, Haoye Tian, Weiguo Pian, Haoran Yu, Haitao Wang, Jacques Klein, Tegawendé F Bissyandé, and Shunfu Jin. Cref: an llm-based conversational software repair framework for programming tutors. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 882–894, 2024b.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024c.

Xiaoyu Yang, Jie Lu, and En Yu. Adapting multi-modal large language model to concept drift in the long-tailed open world. *arXiv preprint arXiv:2405.13459*, 2024d.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint*, 2022.

Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. Learning to represent edits. In *International Conference on Learning Representations*, 2019. URL <https://openreview.net/forum?id=BJ16AjC5F7>.

Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. Thinkrepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1274–1286, 2024.

Xingdi Yuan, Xiaoyu Wang, Caglar Wang, Kunal Aggarwal, Gokhan Tur, Lu Hou, Nan Deng, and Hoifung Poon. Improving code generation by training with natural language feedback. *arXiv preprint arXiv:2303.16749*, 2023a.

Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*, 2023b.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*, 2023a.

Hongxin Zhang, Weihua Du, Jiaming Shan, Qinhong Zhou, Yilun Du, Joshua B Tenenbaum, Tianmin Shu, and Chuang Gan. Building cooperative embodied agents modularly with large language models. *arXiv preprint*, 2023b.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019a.

Mengxi Zhang, Huaxiao Liu, Chunyang Chen, Yuzhou Liu, and Shuotong Bai. Consistent or not? an investigation of using pull request template in github. *Information and Software Technology*, 144:106797, 2022.

Shun Zhang, Michael Ahn, Shuai Jiang, Ashwin Kumar, Bailin Wang, Zhihong Pang, Tengyang Xia, Pengcheng Yin, Shivanshu Mudgal, Marc Crawford, et al. Planning with large language models for code generation. *arXiv preprint arXiv:2305.02309*, 2023c.

Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. Graph convolutional networks: a comprehensive review. *Computational Social Networks*, 6(1):1–23, 2019b.

Ye Zhang and Byron Wallace. A sensitivity analysis of (and practitioners’ guide to) convolutional neural networks for sentence classification. *arXiv preprint arXiv:1510.03820*, 2015.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement, 2024a.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427*, 2024b.

Zhengyan Zhang, Xu Han, Zhiyuan Liu, Xin Jiang, Maosong Sun, and Qun Liu. ERNIE: Enhanced language representation with informative entities. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1441–1451, Florence, Italy, July 2019c. Association for Computational Linguistics. doi:10.18653/v1/P19-1139. URL <https://aclanthology.org/P19-1139>.

Zhengyan Zhang, Xu Han, Zhiyuan Liu, Xin Jiang, Maosong Sun, and Qun Liu. Ernie: Enhanced language representation with informative entities. *arXiv preprint arXiv:1905.07129*, 2019d.

Zhiqiang Zhao, Yiling Yuan, Chen Huang, and Wei Zhao. Chatrepair: Automating bug fixing via large language models. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2023.

Jonathan Zheng, Alan Ritter, and Wei Xu. Neo-bench: Evaluating robustness of large language models with neologisms. *arXiv preprint arXiv:2402.12261*, 2024.

Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International conference on software engineering (ICSE)*, pages 14–24. IEEE, 2012.

Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E Hassan. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 705–716. IEEE, 2021a.

Jiayuan Zhou, Michael Pacheco, Jinfu Chen, Xing Hu, Xin Xia, David Lo, and Ahmed E Hassan. Colefunda: Explainable silent vulnerability fix identification. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2565–2577. IEEE, 2023a.

Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, Junda He, and David Lo. Generation-based code review automation: How far are we? *arXiv preprint arXiv:2303.07221*, 2023b.

Xin Zhou, Bowen Xu, DongGyun Han, Zhou Yang, Junda He, and David Lo. Ccbert: Self-supervised code change representation learning. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 182–193. IEEE, 2023c.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

Yaqin Zhou, Jing Kai Siow, Chenyu Wang, Shangqing Liu, and Yang Liu. Spi: Automated identification of security patches via commits. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–27, 2021b.

Dongsheng Zhu, Xunzhu Tang, Weidong Han, Jinghui Lu, Yukun Zhao, Guoliang Xing, Junfeng Wang, and Dawei Yin. Vislinginstruct: Elevating zero-shot learning in multi-modal language models with autonomous instruction optimization. *arXiv preprint arXiv:2402.07398*, 2024.