A Semantically-Grounded Agentic Framework for Assisting BPMN Model Instance Execution

Tiago Sousa[a], Nicolas Guelfi[b] and Benoît Ries[c]
University of Luxembourg, Esch-sur-Alzette, Luxembourg

Keywords: Large Language Models, Multi-Agent Systems, BPMN, Semantic Validation, Business Process Modeling, Process Simulation, Chain-of-Thought, Agentic Workflows.

Abstract: While Large Language Models can efficiently learn the syntactic patterns of Business Process Model and Notation (BPMN), their probabilistic nature prevents reliable adherence to the deterministic execution rules governing process behavior. Drawing on the distinction between syntactic form and operational meaning, we argue that LLMs approximate BPMN's structural grammar but lack grounding in its formal semantics. This work presents an agent-based system that increases the syntactic and semantic correctness of engineered BPMN execution traces. This is achieved through role-specialized components and continuous validation against BPMN's operational semantics, increasing correctness during generation rather than as post-hoc verification. A process simulator verifies each intermediate BPMN trace and produces CoT-articulated diagnostic feedback when violations occur, guiding automated correction. Experimental evaluation shows marked improvements in conformance to semantic rules compared to baseline approaches, confirming the necessity of external semantic enforcement in model-driven generation tasks.

1 INTRODUCTION

Large Language Models (LLMs) have demonstrated remarkable capabilities in generating fluent, human-like text across diverse domains, from creative writing to technical documentation (Zhao et al., 2025). This success has led to their increasing application in complex, domain-specific generation tasks where structured outputs must conform to formal specifications (Dagdelen et al., 2024). One such domain is business process modeling, where the automated generation of diagrams in Business Process Model and Notation (BPMN) (Model, 2011) promises to accelerate the design and deployment of executable workflows.

However, deploying LLM-generated BPMN models in production introduces a critical challenge beyond syntactic correctness. Automated business processes must execute reliably without human intervention, requiring models that satisfy not only the visual grammar but also formal operational semantics governing runtime behavior. This execution semantics define how process instances traverse the model: which paths tokens follow at gateways, when parallel threads synchronize, and how events interrupt normal flow (Dijkman et al., 2008a). Unlike syntactic rules constraining graph structure, semantic rules govern computational interpretation during execution.

Current LLMs, despite producing syntactically valid diagrams, often violate execution constraints (Drakopoulos et al., 2025). We observe three classes of semantic violations: misconfigured gateway logic (parallel splits without corresponding joins), improper event sequencing (event-based gateways followed by tasks rather than catching events), and malformed control flows producing deadlocks or unreachable activities. An LLM may generate a parallel gateway split creating three concurrent threads but provide only two incoming paths to the corresponding join gateway, causing permanent blocking and preventing process termination. These dynamic semantic violations, invisible at the syntactic level, render generated processes unsuitable for direct execution.

Understanding LLM-generated semantic violations requires distinguishing distributional learning from operational grounding. This distinction maps to Frege's (Zei, 1892) differentiation between "sense"

[a] https://orcid.org/0000-0002-1006-8186
[b] https://orcid.org/0000-0003-0785-3148
[c] https://orcid.org/0000-0002-8680-2797

Sousa, T., Guelfi, N. and Ries, B.
A Semantically-Grounded Agentic Framework for Assisting BPMN Model Instance Execution.
DOI: 10.5220/0014446900004058

In *Proceedings of the 14th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2026)*, pages 97-109
ISBN: 978-989-758-798-6; ISSN: 2184-4348

(Sinn) and "reference" (Bedeutung) (Bender and Koller, 2020): sense represents the mode of presentation of meaning, while reference denotes the actual object or state of affairs.

LLMs operate through distributional learning on BPMN syntax, optimizing parameters to capture statistical co-occurrence patterns characterizing syntactically valid models. This enables reproduction of BPMN's "sense": surface forms, common configurations, and typical element arrangements defining what a syntactically valid diagram *looks like*. The model captures that parallel gateways often appear in matched pairs and that certain element types frequently follow others.

However, BPMN's meaning extends beyond distributional properties. The notation possesses formal denotational semantics mapping syntactic structures to execution behaviors. This operational semantics corresponds to Frege's "reference": it specifies how process instances traverse the model during execution (e.g., exclusive gateways route exactly one token, parallel joins synchronize by waiting for tokens on all incoming paths). These execution rules are deterministic and compositional, defined independently of statistical frequency in training data.

Because LLMs optimize a purely syntactic training objective over token sequences, they lack direct access to this denotational level. The model is a statistical pattern recognizer, not a symbolic executor or theorem prover (Zhao et al., 2025). When generating a gateway element, the LLM does so because this token sequence has high conditional probability, not through explicit computation of the gateway's role. Consequently, semantic correctness cannot emerge reliably from pattern matching alone and must be imposed through external enforcement mechanisms.

To address this semantic gap, we propose an agent-based generation framework where specialized components focus on specific BPMN constructs, and a process simulator continuously validates each intermediate trace against execution semantics, providing diagnostic feedback for corrective regeneration. This research addresses three questions:

1. **Agentic Impact:** How do agentic workflows improve semantic conformance of monolithic LLM-based BPMN generation?
2. **Validation Efficiency:** How does simulator-driven validation affect the number of generation iterations required to produce a semantically valid trace, and what is the distribution of error types detected?
3. **CoT Feedback Effectiveness:** Do CoT-articulated diagnostic messages produce more effective error correction than simple violation notifications, as measured by reduction in repeated errors and convergence rate to valid traces?

2 BACKGROUND

2.1 Large Language Models

Large Language Models (LLMs) operate as probabilistic sequence generators over discrete tokens (Brown et al., 2020). Given a language $L = \{w \mid \Gamma(w)\}$ defined by formation rules Γ, an LLM optimizes its parameters to approximate the probability distribution over L from training data (Kaplan et al., 2020). The generation process is autoregressive: the model constructs an output sequence

$$s_i = \overbrace{\left(w_i^1 \cdot w_i^2 \cdots \cdot w_i^n\right)}^{w_i^{in}} \cdot \overbrace{\left(w_i^{n+1} \cdot w_i^{n+2} \cdots \cdot w_i^{n+p}\right)}^{w_i^{out}}$$

where $\cdot$ denotes string concatenation and each w_i^k is an atomic symbol. It generates one word[1] at a time, conditioned on a prompt $s_{\mathrm{i}}^{\mathrm{in}}$ and all previously generated words. Through optimization, an LLM captures statistical regularities, syntactic patterns, and structural templates that characterize valid outputs in the training distribution.

When applied to structured notations like BPMN, we define $L_{BPMN} = \{m \mid m \text{ conforms to } SYR_{BPMN}\}$ as the language of all syntactically valid models conforming to BPMN's syntactic formation rules SYR_{BPMN}. When prompted to generate a BPMN model, an LLM produces a token sequence s (typically XML markup) that can be parsed to generate a graphical representation. Because LLMs are trained on corpora including BPMN examples, generated sequences are often syntactically plausible, reproducing distributional patterns like element nesting, attribute usage, and graph topologies observed in training data.

However, generation is fundamentally probabilistic, based on captured statistical patterns rather than explicit enforcement of SYR_{BPMN} (Austin et al., 2021). The model approximates which token sequences are common, not which structures are formally correct. High frequency in training data does not imply formal correctness, and syntactic malformations remain common (Austin et al., 2021). Even

[1]Throughout this work, we use "word" as the primary element that LLMs take as input and provide as output, since our approach does not depend on the tokenizer phases of LLM architectures.

when syntactic correctness is achieved, a more fundamental limitation remains.

The deeper challenge lies in semantic correctness. We require a semantic interpretation function $[\![\cdot]\!] : L_{BPMN} \rightarrow O$ mapping syntactically valid models to their denotation in semantic domain O (transition systems) (Nielson and Nielson, 1992). We define semantic correctness with respect to behavioral properties $SER_{BPMN} = \{p_1, \ldots, p_n\}$ (e.g., termination, deadlock-freedom): a model m is semantically correct if and only if $[\![m]\!]$ satisfies all properties in SER_{BPMN}.

Satisfaction depends on adherence to BPMN's operational semantics, which define execution-level constraints: parallel joins synchronize tokens from all incoming paths, event-based gateways must be followed by catching events, and every token must reach a terminating state. A model can satisfy SYR_{BPMN} while violating SER_{BPMN}: a parallel split creating three paths with a join having only two incoming flows is syntactically valid but causes permanent deadlock.

The fundamental limitation lies in the mismatch between LLM training objectives and semantic correctness requirements.

Transformer-based (Vaswani et al., 2017) architectures optimize objectives defined exclusively over token sequences, maximizing likelihood under the training distribution. The self-attention mechanism computes contextualized representations by attending to statistical dependencies (Clark et al., 2019).

The model has no access to $[\![\cdot]\!]$ or SER_{BPMN} during training or generation (Valmeekam et al., 2023). The training corpus contains only surface forms (XML syntax), not execution traces or semantic annotations. When an LLM generates a gateway, it produces tokens with high conditional probability, not through verification of the gateway's role. The model captures that structural patterns are statistically common but not the execution semantics they denote (e.g., an XOR-split requires exactly one outgoing path per instance).

This disconnect explains why LLM-generated models frequently violate operational constraints (Mitchell and Krakauer, 2023). As discussed in Section 1, this reflects the Fregean distinction between sense and reference (Zei, 1892; Bender and Koller, 2020): LLMs grasp distributional patterns but lack access to denotational semantics defining execution behavior. Statistical pattern matching cannot ensure semantic correctness without external enforcement.

2.2 Business Process Model and Notation

BPMN (Model, 2011) serves as both a graphical modeling language and a formal specification with executable semantics. BPMN models consist of flow objects (Events, Activities, and Gateways) connected by sequence flows.

Model correctness is evaluated at two levels. Syntactic correctness (SYR_{BPMN}) concerns structural formation rules constraining graph topology, verifiable through static analysis. Semantic correctness (SER_{BPMN}) concerns operational execution rules governing runtime behavior. BPMN's execution semantics uses token-based interpretation (Dijkman et al., 2008b): a process instance begins with a token at the start event that traverses the graph according to control-flow rules.

Critical semantic constraints govern gateway behavior (Corradini et al., 2018). An Exclusive Gateway (XOR) routes exactly one token to one outgoing path or merges without synchronization. A Parallel Gateway (AND) duplicates tokens to all outgoing paths or synchronizes by waiting for tokens on all incoming paths. An Event-Based Gateway routes to the path whose event occurs first and can only be followed by catching events. Global constraints require every token to reach an end event and prohibit deadlocks. This distinction reveals why LLM-generated BPMN models fail during execution despite appearing syntactically valid (Drakopoulos et al., 2025).

3 RELATED WORK

In this section, we review methods for constraint satisfaction in structured generation, multi-agent LLM approaches, and planning systems relevant to BPMN model generation.

3.1 Improving Constraint Satisfaction

Constraint satisfaction in structured generation requires LLMs to produce outputs conforming to formal specifications beyond distributional patterns. Prompt engineering leverages LLMs for diverse tasks without extensive fine-tuning (Liu et al., 2023), ranging from zero-shot to few-shot in-context learning. However, prompt engineering faces reliability challenges: slight phrasing variations yield substantially different outputs, and approaches remain fragile and model-dependent (Hassan et al., 2024). Standard prompting relies primarily on distributional knowledge and cannot enforce formal constraints in structured notation

generation.

Recent work has applied prompt engineering to business process management tasks (Busch et al., 2023). Prompt design strategies enhance semantic quality and completeness (Ayad and Alsayoud, 2024) by guiding LLMs to suggest missing elements based on domain knowledge. While these methods improve completeness, they do not systematically validate operational semantics in generated BPMN models. CoT prompting enhances reasoning through intermediate steps (Wei et al., 2023) but does not guarantee adherence to domain-specific execution semantics (SER_{BPMN}).

Beyond prompting, recent work investigates explicit rule following and rule induction. Mu et al. (Mu et al., 2024) show that current LLMs frequently violate simple stated rules, exploring test-time steering and supervised fine-tuning. Zhu et al. (Zhu et al., 2024) introduce Hypotheses-to-Theories, inducing reusable rule libraries and reporting 10–30% accuracy gains. However, these advances target textual rule compliance and do not expose BPMN's denotational semantics or guarantee execution-level semantic correctness (SER_{BPMN}). Enforcing token-flow constraints requires evaluating $[\![m]\!]$ through simulation.

Constrained decoding enforces syntactic constraints during LLM generation using formal grammars or structural specifications (Shin et al., 2021). These approaches prevent syntactically invalid structures (Geng et al., 2023), demonstrating improvements with cost reductions of 26–85% (Beurer-Kellner and Vechev, 2023). For business automation workflows, constrained decoding achieves accuracy improvements from 0.50 to 0.75 (Desmond et al., 2022). However, applications to BPMN are limited: while these methods guarantee syntactic well-formedness (SYR_{BPMN}), they cannot verify execution-level semantic correctness (SER_{BPMN}). Constrained decoding enforces distributional patterns (Frege's sense) but lacks access to denotational semantics defining operational behavior. For BPMN, syntactic constraints cannot detect deadlocks from path-asymmetric gateway pairs, unreachable nodes, or incorrect token flow synchronization. These semantic properties require evaluating $[\![m]\!]$ through process simulation, motivating hierarchical validation that enforces execution constraints through integrated simulation-based checking.

3.2 Multi-Agent LLM Systems

LLM-based autonomous agents tackle complex tasks exceeding monolithic, single-shot model capabilities (Wang et al., 2024), distributing cognitive load across specialized components through coordination mechanisms and planning frameworks. However, autonomous agents face challenges regarding hallucinations and limited reliability (Zou et al., 2025). While human-agent systems address these limitations, they introduce scalability constraints and cannot systematically enforce formal correctness in structured generation tasks. Existing multi-agent frameworks primarily address open-ended tasks rather than formal generation with well-defined operational semantics.

3.2.1 Agent Coordination and Communication

Multi-agent collaboration frameworks address complex software engineering tasks through role-based specialization and structured communication (Qian et al., 2024). ChatDev employs a waterfall-inspired approach with chat-based dialogues and communicative dehallucination techniques. AutoGen enables flexible conversation patterns combining LLM capabilities with human inputs and external tools (Wu et al., 2023). AgentVerse explores task-solving pipelines that dynamically recruit agents (Chen et al., 2023). These approaches utilize natural language coordination, with agents exchanging messages to negotiate solutions.

However, natural language communication lacks type constraints and structured schemas necessary for enforcing formal correctness in domain-specific generation tasks. For BPMN generation, unstructured dialogue cannot guarantee elements conforming to specification-level type constraints or execution semantics. This limitation motivates coordination mechanisms employing typed state schemas and specification-driven validation.

3.2.2 Planning and Orchestration Frameworks

Planning mechanisms enable multi-agent systems to decompose high-level goals into actionable subtasks through explicit reasoning about task dependencies (Lan et al., 2024). Recent work explores agent collaboration strategies demonstrating that LLM agents coordinate through turn-based interaction protocols. Survey work categorizes planning approaches into task decomposition strategies where orchestration layers assign subtasks to specialized agents (Wang et al., 2024).

However, planning frameworks typically address open-ended tasks where correctness is subjective, rather than formal generation tasks with well-defined operational semantics. Planning focuses on action sequencing and goal satisfaction rather than ensuring generated artifacts satisfy execution constraints.

For BPMN generation, orchestration must decompose tasks and enforce that assembled outputs conform to token-based execution semantics through systematic validation. This limitation motivates our five-phase orchestration workflow integrating scenario parsing, specialized generation, assembly, simulation-based validation, and targeted repair.

4 METHODOLOGY

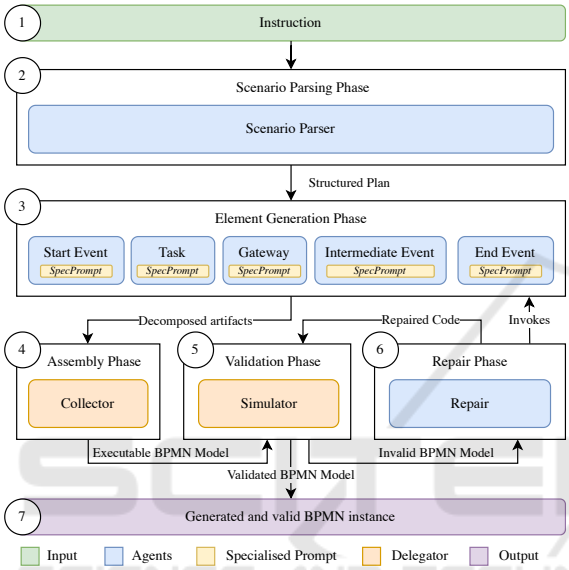

Figure 1: Overview of methodology phases and the seven specialized agents involved in the process.

We present a methodology for generating, validating, and iteratively repairing BPMN models through LLM orchestration. The approach comprises a seven-agent architecture operating across five phases, a six-level semantic validation hierarchy grounded in token-based execution semantics, and an iterative repair mechanism guided by CoT-articulated diagnostic feedback. This section specifies system design, operational assumptions, and formal notation (SER_{BPMN}, $[\![\cdot]\!]$) enabling systematic enforcement of execution semantics.

4.1 Multi-Agent Architecture Overview

Seven specialized agents are orchestrated through a five-phase workflow: Scenario Parsing, Element Generation, Assembly, Validation, and Repair. Scenario Parsing converts natural language descriptions into structured BPMN plans. Element Generation distributes construction across five type-specialized agents (Start Event, Task, Gateway, Intermediate Event, End Event), each operating on isolated subtasks. Assembly combines generated elements into executable process code. Validation executes assembled code in a process simulator verifying six hierarchical levels. Repair analyzes validation failures, maps semantic violations to responsible elements, and invokes specialist agents for targeted regeneration. This iterative loop continues until validation succeeds or resource constraints are exceeded.

Agents coordinate through structured state with enforced type schemas. The state management protocol defines four data structures: `scenario_plan` (parsed elements and flows), `elements` (generated BPMN constructs with metadata), `validation_result` (hierarchical checker output), and `error_map` (violation-to-element mappings). This typed state architecture reduces inter-agent communication ambiguity and enables systematic error propagation from validation to repair. Sequential orchestration ensures dependency satisfaction: parsing precedes generation, generation precedes assembly, assembly precedes validation, and validation precedes repair, preventing cascading violations.

The instruction engineering mechanism combines modular rule composition with selective exposure. Rules are organized into nine categories spanning behavioral constraints, cognitive scaffolding, and construct-specific semantics. Each agent receives a filtered instruction set containing only pertinent rule categories, preventing prompt dilution and focusing attention on relevant constraints. All agents access BPMN 2.0.2 specification rules as few-shot exemplars, providing concrete constraint illustrations. The validation hierarchy applies token-based execution semantics, evaluating $[\![m]\!]$ rather than solely surface structure. The complete implementation is publicly available (Anonymous, 2025).

4.2 Design Principles

Bridging the gap between distributional learning and execution semantics requires explicit architectural mechanisms compensating for probabilistic pattern matching limitations. Three principles guide the system's design to systematically enforce semantic correctness.

4.2.1 Agent Specialization

Autoregressive models optimize token-level conditional probabilities locally without access to global structural constraints. Decomposing generation into construct-specific subtasks exploits LLMs' local pattern matching capabilities while narrowing the space

of semantically valid configurations. By scoping each agent to a well-defined BPMN construct type (events, activities, gateways), we reduce the generation space from all possible BPMN models to type-consistent element configurations, enabling more precise constraint satisfaction.

4.2.2 Hierarchical Validation

Layered validation enforces progressively sophisticated semantic constraints, from basic structural well-formedness through execution-level properties. This stratification enables systematic detection of violations at different abstraction levels, distinguishing syntactic malformation from semantic violations. The hierarchy reflects the compositional nature of BPMN semantics: higher-level execution properties (topological soundness, reachability) presuppose lower-level syntactic correctness (well-formedness, type consistency).

4.2.3 CoT-Articulated Diagnostic Feedback

Repair guidance connects syntactic violations to operational consequences through execution-grounded explanations. This enables LLMs to reason about corrections by articulating *why* constraints matter for token flow behavior, rather than merely identifying *which* rules were violated. Feedback explains *how* violations manifest during execution, providing actionable correction strategies aligned with $[\![\cdot]\!]$ rather than surface-level pattern matching.

4.3 Agent Specialization and Responsibilities

Each agent addresses a narrowly scoped task through selective rule exposure, few-shot examples, and local context scoping. All agents receive BPMN 2.0.2 specification rules as external semantic knowledge.

4.3.1 Parsing Agent

The Scenario Parser converts natural language descriptions into structured BPMN plans through three coordinated operations. Entity extraction identifies flow objects by detecting task mentions with type annotations and inferring gateway presence from control flow indicators. Relationship inference determines sequence flows by mapping temporal and logical connectives to directed edges. Type classification assigns BPMN construct types based on contextual markers. The output is a typed schema `scenario_plan` containing `elements` and `flows` lists with explicit type annotations, serving as the coordination contract that constrains downstream generation and enables type-safe agent dispatch.

4.3.2 Element Generation Agents

Five specialized agents generate distinct BPMN construct categories: Start Event, Task, Gateway, Intermediate Event, and End Event. Each agent receives construct-specific rule subsets through filtered instruction sets. Generated code instantiates BPMN objects with metadata enabling correct assembly.

4.3.3 Assembly and Quality Assurance Agents

The Collector retrieves all element specifications from shared state and assembles them into executable code following a fixed template. The Validator executes assembled code in an isolated environment and invokes hierarchical correctness checking through the six-level semantic hierarchy.

The Repair Agent analyzes validation failures through a two-stage error mapping strategy. When validation output contains explicit element identifiers, direct mapping associates violations with responsible constructs. When identifiers are absent, the algorithm infers responsible elements through keyword-based classification using violation type indicators and contextual cues. This error mapping produces `error_map`, enabling targeted regeneration rather than full process reconstruction. The repair agent incorporates construct-specific violation knowledge through instruction engineering, receiving descriptions of common error patterns and their semantic consequences. Repair proceeds through iterative regeneration: extract failing element code, apply violation-specific corrections, update shared state, and trigger revalidation. The repair loop continues until validation succeeds or resource constraints are exceeded.

4.4 Hierarchical Validation

Semantic correctness requires systematic validation beyond syntactic checking. We introduce a six-level hierarchy reflecting the layered nature of BPMN semantics, where higher levels assume lower-level correctness and verify progressively sophisticated invariants. Validation proceeds sequentially, terminating at the first failure and reporting violations for repair.

The hierarchy addresses three violation classes: misconfigured gateway logic (Levels 4–5), improper event sequencing (Levels 3–4), and malformed control flows (Levels 5–6). Lower levels (1–2) check structural well-formedness and type con-

straints; higher levels (3–6) enforce execution semantics requiring simulation-based verification.

4.4.1 Level 1: Syntactic Validation

Verifies fundamental structural formation rules: exactly one start event, at least one end event, all flow objects connected to sequence flows, and no direct event-to-event flows without intervening tasks or gateways. These constraints define well-formed BPMN graph structures satisfying SYR_{BPMN}.

4.4.2 Level 2: Static Semantic Validation

Checks type constraints and referential integrity: task objects must have valid `task_type` attributes, gateway objects must have valid `gateway_type` attributes, sequence flow references must point to valid flow objects, and all element identifiers must be unique. These constraints ensure elements conform to BPMN 2.0.2 specification types and preserve referential consistency.

4.4.3 Level 3: Event Rules Validation

Verifies event placement constraints: start events have no incoming flows, end events have no outgoing flows, and intermediate events have both. These constraints ensure events conform to their roles in token initiation, termination, and intermediate occurrence.

4.4.4 Level 4: Structural Rules Validation

Enforces domain-specific best practices: no more than four tasks in sequence without intervening gateways, no consecutive service tasks, script tasks must be followed by gateways for error handling, and intermediate events must not appear between tasks of the same type.

4.4.5 Level 5: Topological Rules Validation

Verifies global graph properties requiring graph traversal algorithms: path symmetry (every splitting gateway has a corresponding joining gateway with matching cardinality), valid loop structures (cycles contain at least one task and have exit paths), and gateway exclusivity. Path symmetry ensures token synchronization correctness, preventing deadlocks where joins wait for tokens that never arrive.

4.4.6 Level 6: Reachability Validation

Ensures all elements participate in execution through forward reachability analysis from the start event and backward reachability analysis from all end events. Nodes not reachable from the start are unreachable islands; nodes that cannot reach end events are dead ends. Token flow reachability requires traversing the graph according to execution rules encoded in $[\![m]\!]$.

4.5 Process Simulator and Feedback Mechanism

Semantic validation requires execution-based checking rather than syntactic analysis alone. The process simulator evaluates $[\![m]\!]$ by instantiating the BPMN model in an isolated environment and verifying token flow semantics through the hierarchy. Semantic correctness depends on execution behavior: how tokens traverse the graph, where they synchronize, and which paths they can reach. The simulator provides grounded feedback by reporting constraint violations and their operational manifestations (deadlock scenarios, unreachable paths, unsynchronized token flows).

The feedback mechanism translates validation output into actionable repair instructions through error mapping, localizing faults to specific constructs and articulating violations in terms of execution consequences rather than syntactic descriptions. For example, path asymmetry violations are reported as "split gateway creates N concurrent paths but join gateway expects M paths, causing token accumulation" rather than merely "path symmetry constraint violated." This CoT-articulated diagnostic feedback enables LLMs to reason about operational implications and apply semantically motivated corrections.

The approach employs iterative refinement with termination conditioned on validation success. While practical implementations impose safety limits, the repair protocol prioritizes correctness: the loop continues until validation succeeds or resource constraints are exceeded. Semantic violations often require multiple repair cycles, as correcting one issue may reveal previously masked violations. Extended iteration enables progressive error resolution rather than premature convergence to invalid states.

The refinement protocol follows a generate-validate-repair cycle, tracking iteration counts and violation types to measure validation efficiency across repair cycles, as formalized in Algorithm 1.

5 EVALUATION FRAMEWORK

We evaluate the proposed approach through controlled experiments on a stratified test suite designed to systematically probe semantic conformance across the six-level validation hierarchy. The evaluation

Algorithm 1: Iterative Refinement.

```
Function IterativeRefinement(description, max_iter):
    plan ← parse_scenario(description);
    elements ← generate_elements(plan);
    iteration ← 0;
    repeat
        code ← assemble_code(elements, plan.flows);
        validation_result ← simulate_and_validate(code);
        if not validation_result.valid then
            error_map ← map_errors(validation_result.output, elements);
            elements ← repair_elements(elements, error_map);
        end
        iteration ← iteration + 1;
    until validation_result.valid or iteration ≥ max_iter;
    return code;
```

quantitatively measures semantic correctness, validation efficiency, and feedback-driven repair effectiveness, addressing three research questions that investigate the interplay between agent specialization, hierarchical validation, and CoT-articulated feedback in bridging distributional learning with execution semantics.

5.1 Test Scenario Design

We define 40 test scenarios, organized into six hierarchical categories, that systematically probe semantic conformance from foundational correctness to global reasoning capabilities. Scenarios were constructed bottom-up from BPMN 2.0.2 constraint decomposition, not reverse-engineered. This stratification enables attribution of performance deltas to the intended capabilities being assessed, rather than to confounding errors. It characterizes where distributional learning aligns with, or diverges from, execution semantics.

Level 1: Basic Syntax (7 Scenarios): tests well-formedness constraints satisfying SYR_{BPMN}: exactly one start event, at least one end event, full connectivity, and no direct event-to-event flows. These establish baseline comprehension before progressing to complex capabilities.

Level 2: Static Semantics (6 Scenarios): verifies adherence to BPMN type system and referential integrity: correct task and gateway types per BPMN 2.0.2 specification, unique element identifiers. This distinguishes symbolic correctness from structural aspects.

Level 3: Event Rules (8 Scenarios): probes event semantics through token-based execution constraints: start events have no incoming flows, end events have no outgoing flows, intermediate events maintain bidirectional connectivity. This tests whether models internalize directionality and role responsibilities.

Level 4: Structural Rules (7 Scenarios): assesses local compositional reasoning: maximum task sequence length, prohibition of consecutive service tasks, error handling on script tasks, gateway branching constraints. These test adherence to patterns ensuring maintainability and proper error handling.

Level 5: Topological Properties (6 Scenarios): measures global graph reasoning through control-flow soundness: proper loop structures with exit paths, split/join gateway symmetry, absence of dead ends. These require whole-graph analysis of token synchronization properties.

Level 6: Integration (6 Scenarios): evaluates multi-constraint generalization requiring simultaneous satisfaction of event, structural, and topological constraints. These reveal compounding error modes and constraint interactions reflecting real-world complexity.

5.2 Experimental Setup

We evaluate nine LLMs spanning multiple capability tiers: frontier models (GPT-5 (OpenAI, 2025)), efficient variants (GPT-5 Mini, GPT-5 Nano), Gemini 2.5 family (Comanici et al., 2025) (2.5 Pro, 2.5 Flash, 2.5 Flash Lite), specialized coding models (Qwen3 Coder 30B (Yang et al., 2025)), and alternative architectures (Grok 4 Fast, GLM 4.6 (Team et al., 2025)). Anthropic models were excluded due to API rate constraints during evaluation. This selection enables assessment of consistent improvements across models with varying inductive biases and training objectives. We did not control or set the temperature parameters for any of the models, using each model's default configuration to ensure fair comparison and avoid introducing additional experimental variables that could

confound attribution of performance differences to architectural choices. Additionally, GPT-5 variant models (GPT-5, GPT-5 Mini, GPT-5 Nano) do not accept temperature parameters in their API, further necessitating this approach.

This isolates architectural contributions from base capabilities. For each configuration, 360 test instances (40 scenarios × 9 models) are validated against SER_{BPMN} via $[\![m]\!]$. All models complete all scenarios or reach a 50-iteration cap.

5.2.1 Monolithic Baseline

Direct BPMN code generation from natural language descriptions without agent decomposition or repair mechanisms. This represents standard single-prompt generation. Generated code undergoes post-hoc validation using the proposed process simulator to measure semantic conformance.

5.2.2 Proposed Multi-Agent System

Complete architecture with specialized agent decomposition (StartEventAgent, TaskAgent, GatewayAgent, IntermediateEventAgent, EndEventAgent), hierarchical validation through the six-level semantic hierarchy, and simulation-based feedback for iterative repair with CoT-articulated diagnostics. This configuration implements the full methodology described in Section 2.

All experimental code, including model configurations, test scenarios, and evaluation scripts, is available (Anonymous, 2025).

5.3 Evaluation Metrics

We evaluate the effectiveness of our approach using metrics aligned with the three research questions, ensuring that each metric targets a distinct aspect of semantic conformance enforcement.

5.3.1 Impact on Semantic Conformance

To quantify how agent specialization with domain-specific constraints improves semantic correctness over monolithic generation:

- **Pass Rate:** Proportion of scenarios producing BPMN models m where $[\![m]\!]$ satisfies all six validation levels, measuring end-to-end semantic conformance with SER_{BPMN}
- **First-Attempt Correctness:** Proportion of scenarios passing all validation levels without repair iterations, measuring initial generation quality and indicating whether specialized agents better internalize execution semantics
- **Violation Distribution by Validation Level:** Frequency of violations detected at each hierarchical level (syntax, static semantics, event rules, structural rules, topological properties, integration), revealing at which abstraction level semantic divergence occurs
- **Violation Distribution by Element Type:** Frequency of violations attributed to each BPMN construct class (Gateway, Task, StartEvent, EndEvent, IntermediateEvent), inferred from rule violation patterns to identify which specialized agents face the greatest semantic challenges

5.3.2 Validation Efficiency

To evaluate simulator-driven validation integration and characterize repair dynamics:

- **Repair Iteration Count:** Number of generate-validate-repair cycles required to achieve semantic conformance, reported as mean, median, and 95th percentile to characterize convergence distributions
- **Convergence Rate:** Proportion of scenarios achieving semantic conformance within specified iteration thresholds (1, 3, 5, 10 iterations), indicating validation loop efficiency and bounded repair costs
- **Error Detection by Level:** Distribution of violations detected at each validation level, identifying fail-fast effectiveness where lower-level checks prevent malformed models from reaching expensive higher-level validation
- **Validation Overhead:** Computational cost measured through average execution time and token consumption per scenario, quantifying the trade-off between semantic correctness and computational efficiency

5.3.3 CoT Feedback Effectiveness

To assess whether CoT-articulated diagnostic feedback grounded in execution consequences improves error correction over generic violation notifications:

- **Self-Correction Rate:** Proportion of initially invalid scenarios achieving semantic conformance through iterative repair, measuring the framework's capacity to recover from violations through feedback-driven refinement
- **First-Repair Success:** Proportion of violations resolved in the first repair iteration following CoT-articulated feedback, measuring diagnostic

Table 1: Comparison of Monolithic Baseline vs Proposed Approach Across Models ($N = 40$).

Model	Pass Rate (%)		1st-try (%)		Self-Corr (%)		Time (s)		Tokens		Steps	
	Mono	Ours	Mono	Ours	Mono	Ours	Mono	Ours	Mono	Ours	Mono	Ours
Gemini 2.5 Pro	27.5	97.5	5.0	27.5	5.0	15.0	20.8	53.9	358	192	2.73	10.7
GLM-4.6	15.0	97.5	0.0	20.0	0.0	20.0	172.6	121.6	1467	1341	18.3	11.0
Gemini 2.5 Flash	37.5	95.0	0.0	20.0	0.0	22.5	49.9	42.8	706	221	12.2	10.9
GPT-5	97.5	92.5	32.5	32.5	15.0	67.5	52.2	110.8	191	492	4.6	11.9
Grok-4 Fast	77.5	87.5	15.0	22.5	15.0	40.0	11.55	29.73	109	247	3.9	10.10
Gemini 2.5 Flash Lite	0.0	82.5	0.0	15.0	0.0	22.5	82.4	32.5	3038	2640	24.7	11.1
Qwen3 Coder 30B	55.0	75.0	0.0	15.0	0.0	12.5	32.6	116.9	557	1172	5.9	11.5
GPT-5 Mini	57.5	72.5	0.0	52.5	0.0	42.5	77.0	124.9	259	447	4.9	11.9
GPT-5 Nano	50.0	57.5	0.0	20.0	0.0	20.0	76.7	144.6	410	802	9.7	10.5

actionability and indicating whether execution-grounded explanations enable immediate correction

- **Self-Correction by Validation Level:** Self-correction rate stratified across the six validation levels, revealing which constraint types benefit most from CoT feedback and identifying where execution-grounded reasoning provides maximum leverage
- **Self-Correction by Difficulty:** Self-correction rate stratified across scenario difficulty levels (1–6), assessing learning effectiveness as semantic complexity increases and multiple constraints interact

5.4 Results

5.4.1 Overall Performance Comparison

Table 2 presents aggregate performance across all models and scenarios, demonstrating the impact of agent specialization and simulator-driven repair on semantic conformance. Our approach achieves a pass rate of 84.2%, representing a 37.8 percentage point improvement over the monolithic baseline (46.4%), an 81.5% relative improvement. This substantial gain demonstrates that specialized agents with domain-specific constraint exposure and hierarchical validation substantially enhance adherence to SER_{BPMN}.

Table 2: Overall Performance Comparison ($N = 360$).

Metric	Ours	Mono	Δ
Pass Rate (%)	84.2	46.4	+37.8
1st-Attempt Correctness (%)	25.0	5.8	+19.2

Critically, first-attempt correctness improved 331% (25.0% vs 5.8%), indicating that our approach not only achieves higher final semantic conformance but generates valid process models more frequently without iterative repair. This suggests specialized agents with filtered rule sets better encode execution semantics than monolithic generation relying solely on distributional learning, supporting the theoretical prediction that scoped hypothesis spaces improve constraint satisfaction.

5.4.2 Per-Model Performance Analysis

Table 1 reveals substantial heterogeneity across models, with pass rates ranging from 57.5% (GPT-5 Nano) to 97.5% (Gemini 2.5 Pro, GLM-4.6) for our proposed system. The multi-agent architecture improves pass rates for 8 of 9 models, with improvements ranging from 20.0 to 82.5 percentage points. Lower-capability models exhibit the largest improvements. For instance, Gemini 2.5 Flash Lite: from 0% to 82.5%; GLM-4.6: from 15% to 97.5%, suggesting that agent decomposition and iterative repair compensate for weaker foundational capabilities in approximating SER_{BPMN}. High-capability models (GPT-5: from 97.5% to 92.5%) show a 5 percentage point decrease, indicating their monolithic performance already approaches the task ceiling, leaving limited room for architectural improvement.

5.4.3 Semantic Conformance by Validation Level

Table 3 stratifies pass rates across the six validation levels, revealing consistent improvements across all semantic constraint categories. The hierarchical pattern (declining pass rates from syntax (95.2%) to integration (72.1%)) aligns with the theoretical prediction that local constraints requiring pattern matching are easier to satisfy than global properties requiring multi-hop graph reasoning about token flow synchronization.

Syntax constraints achieve the highest pass rate (95.2%), demonstrating that basic well-formedness rules (start/end events, connectivity) are effectively enforced through specialized StartEventAgent and EndEventAgent components with explicit SYR_{BPMN} constraints. Topological and integration constraints exhibit lower pass rates (75.7%, 72.1%), indicating

Table 3: Pass Rate by Validation Level ($N = 360$).

Validation Level	Tests	Ours (%)	Mono (%)
Syntax	63	95.2	53.2
Static Semantics	54	88.9	50.0
Event Rules	54	88.9	49.1
Structural Rules	72	87.5	48.6
Topological Properties	70	75.7	44.8
Integration	43	72.1	40.5
Overall	356	84.2	46.4

that global control-flow properties (path symmetry requiring matching split/join gateway cardinalities, reachability analysis across all execution paths) remain challenging even with specialized agents. This finding aligns with established theory that distributional learning captures local co-occurrence patterns more readily than global structural invariants requiring explicit reasoning about $[\![m]\!]$ (Mitchell and Krakauer, 2023).

The consistent improvement margins (30.9–42.0 percentage points) across all categories indicate robust benefits across the entire semantic constraint spectrum, supporting the hypothesis that agent specialization with hierarchical validation provides systematic improvements rather than optimizing for specific constraint types.

Violation Patterns by Element Type. Table 4 analyzes violations by BPMN construct type, inferred from rule violation patterns. Gateway logic exhibits the highest violation rate (18.9%), indicating that path symmetry constraints (ensuring split gateways creating N parallel paths have corresponding join gateways expecting N incoming paths) remain challenging despite the specialized GatewayAgent. This finding reflects the fundamental difficulty of reasoning about token synchronization properties that depend on global graph topology rather than local patterns.

Table 4: Violation Frequency by Element Type, Proposed Approach ($N = 360$).

Element Type	Violations	Total Tests	Rate (%)
Gateway	68	360	18.9
Task	58	360	16.1
StartEvent	43	360	11.9
EndEvent	30	360	8.3
IntermediateEvent	19	360	5.3

Task sequencing violations (16.1%) primarily stem from structural rules, suggesting opportunities to enhance TaskAgent with more sophisticated compositional reasoning. Event placement violations show comparatively lower rates (StartEvent: 11.9%, EndEvent: 8.3%, IntermediateEvent: 5.3%), validating the effectiveness of event-specific agents.

5.4.4 Validation Efficiency

Table 5 compares validation and repair efficiency. While our approach incurs 35% higher execution time (86.42s vs 63.97s), it achieves 67% fewer repair attempts (0.66 vs 1.99) and 39% fewer validation attempts (1.54 vs 2.54), demonstrating improved convergence efficiency that offsets computational overhead.

Table 5: Validation Efficiency Comparison ($N = 360$).

Metric	Ours	Mono	Δ
Avg Repair Attempts	0.66	1.99	-1.33
Avg Validation Attempts	1.54	2.54	-0.99
Avg Execution Time (s)	86.42	63.97	+22.45

5.4.5 CoT-Articulated Feedback Effectiveness

Table 6 shows that CoT-articulated diagnostic feedback significantly improves self-correction capabilities in LLM-driven BPMN generation by providing execution-grounded explanations. When violations occur, our system resolves them on the first repair attempt in more than half of cases (56.8%), compared to only 8% for monolithic generation with generic constraint notifications. This seven-fold improvement supports the theoretical prediction that connecting syntactic violations to their operational consequences (explaining *how* path asymmetry causes token accumulation deadlocks, *why* event-based gateways require catching events) provides actionable correction strategies grounded in $[\![m]\!]$ rather than surface-level pattern adjustments.

Complete experimental results, including per-scenario performance metrics, violation logs, and repair traces, are available in the supplementary materials (Anonymous, 2025).

Table 6: Self-Correction Performance Comparison ($N = 360$).

Metric	Ours (%)	Mono (%)	Δ
Self-Correction Rate	29.2	3.9	+25.3
First-Repair Success	56.8	8.0	+48.8

6 DISCUSSION

The multi-agent architecture with hierarchical validation and CoT-articulated feedback improves semantic conformance across all evaluated LLMs, though improvement magnitude varies significantly. Models with weak baselines (Gemini 2.5 Pro: 27.5% to

97.5%, GLM-4.6: 15.0% to 97.5%) benefit substantially from agent specialization and validation loops. GPT-5 exhibits a distinct pattern: its monolithic baseline achieves 97.5% pass rate, while the multi-agent system yields 92.5%, but self-correction improves substantially (15.0% to 67.5%), indicating that CoT feedback enhances repair capabilities when errors occur.

Our evaluation employs standardized prompts across all LLMs. Recent work demonstrates substantial prompt sensitivity, with performance variations up to 76 percentage points due to formatting differences (Sclar et al., 2024), and optimal prompt templates do not generalize across models (Voronov et al., 2024). We prioritize experimental control to isolate architectural contributions. Model-specific prompt optimization would conflate architectural benefits with prompt engineering effects. By controlling prompts, we attribute improvements to agent specialization, hierarchical validation, and CoT feedback. Future work could employ prompt ensembles to quantify architecture-prompt interactions.

7 CONCLUSION

LLMs generate BPMN models through distributional learning over syntactic patterns, yet this statistical approach diverges from formal execution semantics, producing semantic violations that render models unsuitable for automated execution.

We address this gap through a multi-agent architecture combining specialized generation agents, hierarchical semantic validation, and CoT-articulated diagnostic feedback. Agent specialization narrows the generation space to type-specific constructs. Hierarchical validation enforces semantic properties from syntactic correctness through topological soundness. CoT feedback connects errors to operational consequences, providing explicit reasoning about token flow behavior.

Evaluation across nine LLMs demonstrates pass rate improvements up to 70 percentage points. Results hold across diverse architectures, scales, and training objectives, indicating architectural interventions provide value independent of base model characteristics. External semantic enforcement can bridge distributional learning and operational semantics in domains requiring formal constraint satisfaction.

Future work will evaluate industrial process models, investigate model-specific prompt optimization, and compare against emerging BPMN generation approaches.

REFERENCES

(1892). *Zeitschrift für Philosophie und spekulative Theologie*. Weber.

Anonymous, A. (2025). A Semantically-Grounded Agentic Framework for Assisting BPMN Model Instance Execution. Zenodo.

Austin, J., Odena, A., and Nye, M. (2021). Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Ayad, S. and Alsayoud, F. (2024). Prompt engineering techniques for semantic enhancement in business process models. *Business Process Management Journal*, 30(7):2611–2641.

Bender, E. M. and Koller, A. (2020). Climbing towards NLU: On meaning, form, and understanding in the age of data. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5185–5198.

Beurer-Kellner, M. and Vechev, M. (2023). Prompting Is Programming: A Query Language for Large Language Models. *LMQL as described in Prompting Is Programming: A Query Language for Large Language Models*, 7(PLDI):186:1946–186:1969.

Brown, T. B., Mann, B., Ryder, N., et al. (2020). Language Models are Few-Shot Learners.

Busch, K., Rochlitzer, A., and Sola, D. (2023). Just Tell Me: Prompt Engineering in Business Process Management. In van der Aa, H., Bork, D., Proper, H. A., and Schmidt, R., editors, *Enterprise, Business-Process and Information Systems Modeling*, pages 3–11, Cham. Springer Nature Switzerland.

Chen, W., Su, Y., and Zuo, J. (2023). AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors.

Clark, K., Khandelwal, U., and Levy, O. (2019). What does BERT look at? An analysis of BERT's attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 276–286.

Comanici, G., Bieber, E., Schaekermann, M., et al. (2025). Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities.

Corradini, F., Fornari, F., and Polini, A. (2018). A formal approach to modeling and verification of business process collaborations. *Science of Computer Programming*, 166:35–70.

Dagdelen, J., Dunn, A., and Lee, S. (2024). Structured information extraction from scientific text with large language models. *Nature Communications*, 15(1):1418.

Desmond, M., Duesterwald, E., and Isahagian, V. (2022). A No-Code Low-Code Paradigm for Authoring Business Automations Using Natural Language.

Dijkman, R. M., Dumas, M., and Ouyang, C. (2008a). Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 50(12):1281–1294.

Dijkman, R. M., Dumas, M., and Ouyang, C. (2008b). Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 50(12):1281–1294.

Drakopoulos, P., Malousoudis, P., and Nousias, N. (2025). Do LLMs Speak BPMN? An Evaluation of Their Process Modeling Capabilities Based on Quality Measures.

Geng, S., Josifoski, M., and Peyrard, M. (2023). Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning. In Bouamor, H., Pino, J., and Bali, K., editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10932–10952, Singapore. Association for Computational Linguistics.

Hassan, A. E., Oliva, G. A., and Lin, D. (2024). Rethinking Software Engineering in the Foundation Model Era: From Task-Driven AI Copilots to Goal-Driven AI Pair Programmers.

Kaplan, J., McCandlish, S., and Henighan, T. (2020). Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Lan, Y., Hu, Z., and Wang, L. (2024). LLM-Based Agent Society Investigation: Collaboration and Confrontation in Avalon Gameplay.

Liu, P., Yuan, W., and Fu, J. (2023). Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.*, 55(9):195:1–195:35.

Mitchell, M. and Krakauer, D. C. (2023). The debate over understanding in AI's large language models. *Proceedings of the National Academy of Sciences*, 120(13):e2215907120.

Model, B. P. (2011). Notation (BPMN) version 2.0. *OMG Specification, Object Management Group*, 19:52–60.

Mu, N., Chen, S., and Wang, Z. (2024). Can LLMs Follow Simple Rules?

Nielson, H. R. and Nielson, F. (1992). *Semantics with Applications: A Formal Introduction*. Wiley.

OpenAI (2025). Introducing GPT-5. https://openai.com/index/introducing-gpt-5/.

Qian, C., Liu, W., and Liu, H. (2024). ChatDev: Communicative Agents for Software Development.

Sclar, M., Choi, Y., and Tsvetkov, Y. (2024). Quantifying Language Models' Sensitivity to Spurious Features in Prompt Design or: How I learned to start worrying about prompt formatting.

Shin, R., Lin, C. H., and Thomson, S. (2021). Constrained Language Models Yield Few-Shot Semantic Parsers.

Team, G.-. ., Zeng, A., and Lv, X. (2025). GLM-4.5: Agentic, Reasoning, and Coding (ARC) Foundation Models.

Valmeekam, K., Marquez, M., and Sreedharan, S. (2023). On the planning abilities of large language models–A critical investigation. *Advances in Neural Information Processing Systems*, 36.

Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Voronov, A., Wolf, L., and Ryabinin, M. (2024). Mind Your Format: Towards Consistent Evaluation of In-Context Learning Improvements.

Wang, L., Ma, C., and Feng, X. (2024). A Survey on Large Language Model based Autonomous Agents. *Frontiers of Computer Science*, 18(6):186345.

Wei, J., Wang, X., and Schuurmans, D. (2023). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.

Wu, Q., Bansal, G., and Zhang, J. (2023). AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation.

Yang, A., Li, A., and Yang, B. (2025). Qwen3 Technical Report.

Zhao, W. X., Zhou, K., and Li, J. (2025). A Survey of Large Language Models.

Zhu, Z., Xue, Y., and Chen, X. (2024). Large Language Models can Learn Rules.

Zou, H. P., Huang, W.-C., and Wu, Y. (2025). A Survey on Large Language Model based Human-Agent Systems.