Towards Model Compliance Using Generative Agents: A NetLogo to Sequence Diagrams Experiment

Benoît Ries[a], Nicolas Guelfi[b] and Tiago Sousa[c]
Department of Computer Science, University of Luxembourg, Esch-Belval, Luxembourg
{*benoit.ries, nicolas.guelfi, tiago.sousa*}*@uni.lu*

Keywords: Artificial Intelligence for Modeling Support, Model Transformations, Generative Approaches, Reverse Engineering, Domain-Specific Modeling.

Abstract: Simulation helps software engineering teams explore complex system behavior, yet NetLogo code remains hard to interpret without design-level documentation. Reverse engineering using generative agents offers a solution to reconstruct models that visually document design. This paper reports on an experiment using generative AI to reverse engineer NetLogo simulations into restricted UML sequence diagrams that represent execution scenarios, produced by specialized generative agents. To ensure compliant model generation, we orchestrate the generative AI task in multiple specialized steps with intermediate compliance audits, each step guided by personas and domain-specific rules. We evaluate the approach on ten public NetLogo simulations paired with eight generative AI models, for a total of 80 experimental runs. Results show that Gemini 2.5 Flash achieves the best results, followed by GPT-5-mini, GPT-5, Devstral; Qwen3, Maverick remain promising, whereas GPT-5-nano underperforms. The experiment shows that orchestrating generative agents with iterative compliance audits improves model compliance.

1 INTRODUCTION

Software engineering methodologies (Sommerville, 2015) are structured around software development lifecycles, which define the sequential phases for specifying, implementing, and validating software systems. These sequential activities constitute *forward engineering*, whereas *reverse engineering* (Chikofsky and Cross, 1990; Cross et al., 1992) reconstructs earlier artifacts from existing outputs, for instance producing design models from code.

NetLogo agent-based simulations have been widely adopted over the past two decades to model complex phenomena. These simulations are implemented in a programming language that is both domain-specific and technical, so they remain impractical for domain experts. Recent empirical studies show that documentation for research software is often incomplete, outdated, or treated as a one-off activity rather than a maintained artifact (Hermann and Fehr, 2022; Janssen et al., 2020). In practice, the only reliable description of a simulation model is frequently its source code.

In our context, reverse engineering is particularly interesting to produce design documents. This is typically required to ease communication with non-technical stakeholders. In this paper, we illustrate our approach with a reverse-engineering process that analyzes simulation code to generate a possible execution scenario specified as a restricted subset of UML (OMG, 2017) sequence diagrams that we call LUCIM.

In this paper, we consider the concept of *model compliance* as the satisfaction of all so-called *compliance rules* related to a given model. Instead, a model is considered non-compliant if it violates at least one compliance rule. Within our process, compliance is governed by a set of model-specific rules that serve as both guiding principles for model generation and evaluation criteria for auditing agents. This work addresses the following research question:

Can generative AI agents be orchestrated to reverse-engineer simulation code into compliant domain-specific models?

This paper makes the following contributions: (i) presents a generative AI agent orchestration process combining generation and auditing agents for compliance; (ii) defines a set of 54 domain-specific rules for

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

guiding reverse engineering of simulation code; (iii) evaluates on 10 NetLogo case studies using 8 generative AI models; and (iv) analyzes the resulting rule violations to discuss how rule complexity and orchestration decisions impact compliance.

Section 2 introduces background concepts. Section 3 details the orchestration process. Section 4 reports on related work. Section 5 concludes and proposes future directions.

2 BACKGROUND

2.1 NetLogo Simulation Language

NetLogo is defined as a programmable modeling environment for simulating natural and social phenomena. It supports modeling and simulation of complex systems composed of interacting agents over discrete time steps (Wilensky and Rand, 2015). The core concepts comprise agents (*turtles*), spatial cells (*patches*), connections between agents (*links*), and a controller overseeing the simulation (*observer*). The NetLogo simulation user interface provides real-time visualization of simulation variables and allows users to execute commands to guide the simulation path.

Listing 1 illustrates a NetLogo code snippet that declares ant and nest agents (lines 1-2) in the ant-colony case study. The procedure `look-for-food` (lines 3-7) first checks (line 4) whether an agent stands on a patch with food; if food is present, the ant gains energy, turns around and returns to its nest.

Listing 1: Listing NetLogo: Procedure code snippet.

```
1 breed [nests nest] ; the ants' home
2 breed [ants ant]   ; the red and blue ants
3 to look-for-food   ; turtle procedure
4   if [is-food?] of patch-here = true
5   [set energy energy + 1000
6     rt 180
```

2.2 UML Sequence Diagrams

The Unified Modeling Language (UML) (OMG, 2017) is the de facto general-purpose modeling language standard. It is used to model different aspects of the structure and behavior of software systems. UML models may be used as supporting artifacts to document design phases. It provides structural (e.g. class diagrams) and behavioral views (e.g. sequence diagrams). Sequence diagrams are a behavioral view that model the order of interactions between the different participants in a process of the system under study.

Figure 1 shows a possible scenario modeled as a sequence diagram, where an ant (named ant1) looks for food via the output event `oeLookForFood("ant1")`, the system acknowledges when food is found with location via theinput event `ieFoodFound(10,10)`, and then returns to nest from its current location.

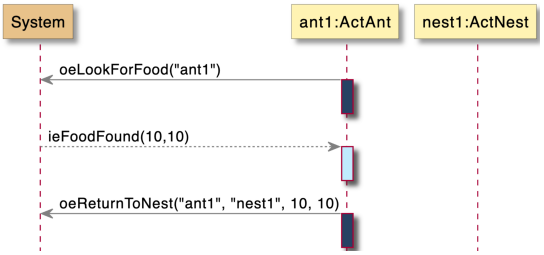

Figure 1: Example UML Sequence Diagram.

2.3 PlantUML

The target artifact of our approach is a sequence diagram modeled using the PlantUML textual syntax. PlantUML (Pla, 2025) is an open-source tool that allows users to generate UML diagrams graphically from textual descriptions. It supports a wide range of UML diagrams, including sequence diagrams, which are the focus of this paper. Listing 2 shows a PlantUML code snippet that was used to generate the UML sequence diagram in Figure 1.

Listing 2: Listing PlantUML: Sequence diagram snippet.

```
1  @startuml
2  [... omitted layout declarations ...]
3  participant System as system #E8C28A
4  participant "ant1:ActAnt" as ant1 #FFF3B3
5  participant "nest1:ActNest" as nest1
       #FFF3B3
6
7  ant1 -> system : oeLookForFood("ant1")
8  activate ant1 #274364
9  deactivate ant1
10
11 system --> ant1 : ieFoodFound(10,10)
12 activate ant1 #C0EBFD
13 deactivate ant1
14
15 ant1 -> system : oeReturnToNest("ant1",
       "nest1", 10, 10)
16 activate ant1 #274364
17 deactivate ant1
```

2.4 Reverse Software Engineering

Reverse software engineering aims to reconstruct artifacts from an existing software application. These

artifacts may be design models, requirements specifications, or other artifacts from the software development lifecycle (Sommerville, 2015). In our context, we are interested in reconstructing a model that documents the design of the simulation code under study. The process produces informative scenario sequence diagrams to ease the understanding of the simulation code.

2.5 Generative AI Agents Orchestration

Generative AI (GenAI) relies on transformer-based deep learning models trained on large datasets of text and other content types. Based on a given input, a prompt, and using attention mechanisms, they generate replies among the most probable ones given the input. One major challenge of GenAI models is to use them such that they provide compliant results. To better control the generation process, one technique consists of decomposing this process into several steps that alternate verification and regeneration phases. Using orchestration of specialized generative AI agents helps to guide the attention of the LLMs towards the specific aspects of the current task. Therefore, the approach orchestrates a set of GenAI agents to ensure a rigorous task decomposition, which in turn is expected to improve the quality of the final generation.

3 SimVis PROCESS: ORCHESTRATION OF GenAI AGENTS FOR COMPLIANT REVERSE-ENGINEERED MODELS

3.1 LUCIM Language

LUCIM is a language defined as a restriction of UML sequence diagrams. As an illustration, such restrictions include, but are not limited to: exactly one *System* participant exists per diagram; the System participant is always positioned first on the leftmost lifeline; and messages from the System to itself are forbidden. For example, Figure 1 shows a valid LUCIM diagram where the System is on the left, with an actor `ant1:ActAnt` sending an output message `oeLookForFood` to the System, followed by the System responding with an input message `ieFoodFound` to the actor; however, any message originating from the System and targeting the System itself violates the LUCIM constraints and is considered non-compliant.

3.2 SimVis Process Presentation

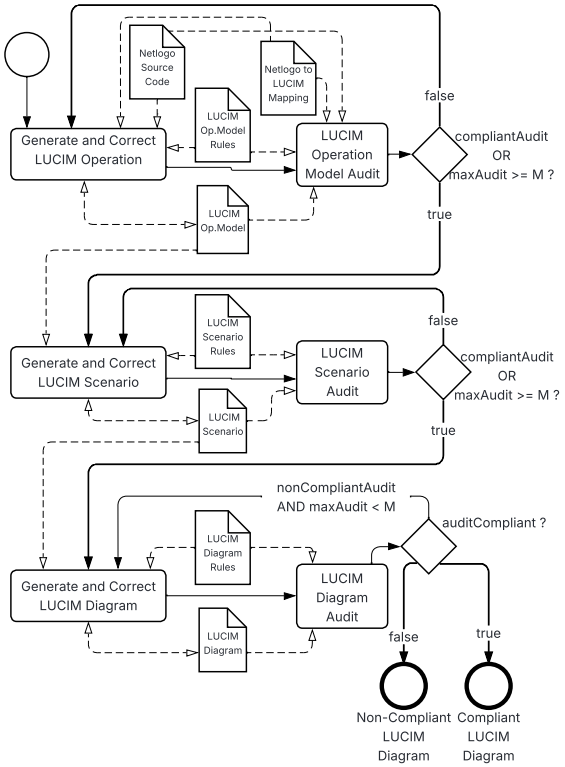

Figure 2: BPMN model of the SimVis Process.

The SimVis process reverse-engineers NetLogo simulations into the three LUCIM artifacts: operation model, scenario, and diagram. Each stage applies persona-guided generation and auditing steps that strictly follow the domain rules. This orchestration keeps the artifacts aligned with the source simulation while enforcing compliance throughout the process tasks.

3.2.1 Orchestration Overview

The SimVis process is structured around the orchestration of specialized GenAI agents to ensure the generation and compliance of domain-specific artifacts. Figure 2 is a BPMN (OMG, 2013) model of the SimVis orchestration process that consists of six tasks until reaching the two possible end states, i.e., a compliant or non-compliant diagram. These tasks are organized into three stages: Generation & Audit of an Operation Model, a Scenario, and a Diagram.

Each stage is composed of specialized generation and audit tasks. If the generated artifact is found non-compliant by the GenAI auditor, the generator iteratively revises it based on the auditor's feedback until M iterations are reached. When M iterations for a given stage are reached, the process continues with

the latest generated artifact regardless of its compliance status.

3.2.2 Compliance and Compliance Rules

We consider an artifact *GenAI-compliant* if it satisfies all the compliance rules given to the GenAI auditor agent; instead, an artifact is non-compliant if it violates at least one compliance rule. Within the process, compliance is governed by a set of artifact-specific rules that serve as both guiding principles for artifact generation and evaluation criteria for auditing agents. The 54 rules can be found in (Ries, 2026) and are organized into nine categories: *Valid Format* (syntax conformance), *Naming* (conventions for actors and events), *Cross-artifact Consistency* (type and name consistency across artifacts), *Message Flow* (event directions and activation bar sequencing), *Condition* (conditional logic expressions), *Temporal* (event ordering and parameter consistency), *Quantitative* (limits on actors and events), *Graphical Syntactic* (visual representation requirements), and *Textual Syntactic* (declaration syntax and formatting).

3.2.3 Generative AI Agents

The process involves six GenAI agents, two per stage, orchestrated deterministically by a non-GenAI orchestrator that manages execution order, input provisioning, and output retrieval.

All GenAI agents are defined with persona descriptions specifying tasks, qualities, tone, output format, and error handling. Each agent's system prompt concatenates its persona with agent-specific inputs. The three *Generator* agents produce *LUCIM Operation Model*, *LUCIM Scenario*, and *LUCIM Diagram* artifacts, taking as input compliance rules and other files shown in Figure 2. For iterations beyond the first, generators also receive the artifact and the audit report from the previous iteration to improve compliance probability. The three *Auditor* agents accept the same inputs as their generator counterparts and produce *Audit Reports*.

3.2.4 Artifacts and Metamodeling

Modeling artifacts must be compliant with their metamodel, which defines allowed concepts and well-formedness constraints. In our context, *LUCIM Diagram* models are a restricted version of UML sequence diagrams, with restrictions defined as compliance rules (see Sect. 3.2.2). A sample rule (LDR5, see Listing 3) forbids messages from the System participant to itself. An OCL specification of this constraint is shown in Listing 4.

Listing 3: Listing LDR5: Messages must never connect System to System.

```
<LDR5-SYSTEM-NO-SELF-LOOP>
Events must never be from System to System,
i.e. System -> System
</LDR5-SYSTEM-NO-SELF-LOOP>
```

Listing 4: Listing OCL-LDR5: Constraint preventing System-to-System messages.

```
context Interaction
inv LDR5_SYSTEM_NO_SELF_LOOP:
self.message->forAll(m: Message |
not (m.sendEvent.covered.name = 'System'
and m.receiveEvent.covered.name = 'System'))
```

The *Audit Report* stores the audit verdict ("compliant" or "non-compliant"), non-compliant rules if any, and correction strategies. Listing 5 shows an example audit report with a non-compliant verdict. *Personas* guide GenAI agent behavior through instruction sets; Listing 6 shows a persona example for the *Diagram generator* agent. Placeholders (e.g., `<AUDIT-REPORT>`) are replaced with actual artifact values when constructing system prompts.

Listing 5: Example Audit Report with non-compliant verdict.

```
{ "verdict": "non-compliant",
  "non-compliant-rules": [ {
     "rule": ↩
          "LDR22-EVENT-PARAMETER-FLEX-QUOTING",
     "line": "11",
     "msg": "Event parameter 'oeSetup()' ↩
          is not properly quoted as per ↩
          LDR22-EVENT-PARAMETER-FLEX-QUOTING"
} ] }
```

3.3 Experimentation

3.3.1 Experimentation Development and Setup

The complete experimentation code is developed in Python and available here (Ries, 2026). PlantUML (Pla, 2025) is used to render the LUCIM sequence diagrams graphically.

In addition to the orchestrator and the GenAI agents, a *classic auditor* is developed in Python and serves as a reference for verifying the compliance results of the GenAI auditor agents. This Python code is thoroughly tested with valid and invalid inputs with a target error rate of less than 2%. The classic auditor is used to check the validity of the audit reports of the GenAI auditor agents, as shown in Table 2. We introduce the notion of *classic-compliance* for artifacts that satisfy every rule checked by the classic auditor;

Listing 6: Extract of the persona for the LUCIM PlantUML Diagram Generator agent.

```
**Main Task**
You are an assistant specialized in generating and
      correcting LUCIM PlantUML Diagrams based on
      input <LUCIM-SCENARIO> and LUCIM PlantUML
      Diagram  validation constraints
      <RULES-LUCIM-PLANTUML-DIAGRAM>.
**Missions:**
You have two main missions:
- **Mission 1:** When provided with an empty
      <PREVIOUS-LUCIM-PLANTUML-DIAGRAM> and an empty
      <AUDIT-REPORT>:
  - Generate a complete, human- and machine-readable
        LUCIM PlantUML Diagram that conforms to
        <RULES-LUCIM-PLANTUML-DIAGRAM>.
- **Mission 2:** When provided with a non-empty
      <PREVIOUS-LUCIM-PLANTUML-DIAGRAM> and a
      non-empty <AUDIT-REPORT>:
  - Revise <PREVIOUS-LUCIM-PLANTUML-DIAGRAM> and
        produce a revised LUCIM PlantUML diagram by
        applying fix-suggestions provided in
        <AUDIT-REPORT>. The revised model must comply
        with all rules in
        <RULES-LUCIM-PLANTUML-DIAGRAM>, with a
        particular attention to the non-compliant
        rules in <AUDIT-REPORT>.
```

instead, an artifact is non-compliant when it violates at least one rule.

The orchestration has been executed with a maximum of 5 iterations for each stage, i.e., M=5, see Figure 2. Also, the orchestrator implements a retry mechanism with a maximum of 3 retries for each task, this retry is used to handle the errors from the third-party APIs that may occur during the execution of the GenAI models (e.g. API rate limiting, API timeout, Network errors, etc.).

3.3.2 NetLogo Simulation Codes Dataset

We evaluate SimVis on ten publicly available (Net, 2025) NetLogo case studies: *3d-solids*, *altruism*, *ant-adaptation*, *artificial-nn*, *boiling*, *continental-divide*, *diffusion-network*, *frogger*, *piaget-vygotsky*, and *signaling-game*.

Table 1 summarizes each case study's complexity across five metrics: lines of code (LoC), agent types, procedures, variables, and conditions. The most complex case study is *ant-adaptation* (512 LoC, 10 agent types, 72 procedures, 18 variables, 85 conditions), that models an ant colony behavior with multiple interacting agent types. The least complex is *boiling* (20 LoC, 1 agent type, 3 procedures, 0 variables, 0 conditions), that models heat diffusion across a grid.

3.3.3 AI Models Selection

GenAI models become outdated very quickly, as the market is evolving rapidly. Thus, the most important aspect in our selection is to have selection criteria that cover a wide range of different GenAI models.

Eight GenAI models are selected for this experiment based on their capabilities, reasoning, context window size, cost-efficiency profiles, and proprietary versus open-source. The selected models are: GPT-5, GPT-5-mini, GPT-5-nano, Gemini 2.5 Pro, Gemini 2.5 Flash, Llama 4 Maverick, Qwen 3 235b-a22b-thinking, and Devstral.

All selected models are general-purpose (except for Devstral from MistralAI that is code-specialized), and instructions-tuned, with varying parameter sizes, reasoning capabilities and context window size. Qwen3 and Maverick are open-source models (i.e. open-weight and self-hostable), while the others are proprietary models. GPT-5 and Gemini 2.5 pro are flagship frontier models, and the most costly, while the others are cost-efficient models.

3.3.4 Compliance Metrics

In order to discuss and reflect on the experimentation results in Section 3.4, we use the following metrics.

In Table 2, we use four metrics: (✓ — ✓) *true positive* (TP): both auditors report compliant; (× — ×) *true negative* (TN): both auditors identify non-compliance; (✓ — ×) *false positive* (FP): GenAI auditor reports compliant but classic auditor finds non-compliant; (× — ✓) *false negative* (FN): GenAI auditor reports non-compliant but classic auditor finds compliant.

Table 3 informs about the *total violation counts per rule category*, i.e., the sums of violations counts for all GenAI models, and all cases, grouped by category. As mentioned earlier, we defined and used 54 rules: 10 for the "LUCIM Operation Model" stage (LOMx), 15 for the "LUCIM Scenario" stage (LSCx), and 29 for the "LUCIM Diagram" stage (LDRx). In addition, Figure 3 shows the evolution of the *total violation counts per stage and per iteration*, i.e., the sums of violations counts for all GenAI models, and all cases, grouped by iteration and stage.

In Table 4, we compare the *task capabilities of the GenAI models* for the six tasks of our process. This performance is computed by the ratio of the number of classic-compliant verdicts (i.e., TP+FN) to the total number of cases (i.e., 10) for each model and stage.

3.4 Discussion

3.4.1 On the Compliance Evolution through Iterations

We begin by analyzing the *final GenAI-compliance*, i.e., the true positives in the last iteration of the Diagram stage as shown in Table 2. Observing *true positive* counts shows that proprietary models achieve the highest compliance: Gemini Flash (10 TP on iter 1), GPT-5 and GPT-5-mini (9 TPs on iter 2), Gemini 2.5

Table 1: Case studies properties from NetLogo code files.

Num.	Case Study	#LoC	#Agents	#Commands	#Global/Local Variables	#Conditions
1.	3d-solids	175	1	16	3	6
2.	altruism	121	1	10	5	14
3.	**ant-adaptation**	**512**	**10**	**72**	**18**	**85**
4.	artificial-nn	139	7	20	36	9
5.	**boiling**	**20**	**1**	**3**	**0**	**0**
6.	continental-divide	216	2	14	34	10
7.	diffusion-network	88	7	12	29	5
8.	frogger	417	8	54	47	38
9.	piaget-vygotsky	176	1	29	35	31
10.	signaling-game	213	12	42	30	14

Table 2: GenAI- and classic-auditor compliant verdicts count evolution through iterations per stage and per model.

Model	Audit Verdict		Audit Verdicts Count																	
	GenAI	Classic	Operation Model					Scenario					Diagram							
			1st	2nd	3rd	4th	5th	1st	2nd	3rd	4th	5th	1st	2nd	3rd	4th	5th			
gemini-2.5-flash	✓	✓	5	9	10			6	8	9	10		10							
	×	×	2			-	-					-		-	-	-	-			
	✓	×																		
	×	✓	3	1				4	2	1										
gemini-2.5-pro	✓	✓	7	10				2	2	2	2	2	8							
	×	×			-	-	-	1	1	1	1			-	-	-	-			
	✓	×						7	7	7	7	7	2							
	×	✓	3									1								
gpt-5	✓	✓	10					5	5				8	9						
	×	×		-	-	-	-	1		-	-	-	1		-	-	-			
	✓	×						4	5				1	1						
	×	✓																		
gpt-5-mini	✓	✓	9					9	9	9	9		8	9						
	×	×		-	-	-	-	1	1	1		-	1		-	-	-			
	✓	×	1								1		1	1						
	×	✓																		
gpt-5-nano	✓	✓	5	6				2	4				1	1	2	2	2			
	×	×	2		-	-	-	1		-	-	-	5	2	1	1				
	✓	×	3	4				5	6				4	7	7	7	8			
	×	✓						2												
llama-4-maverick	✓	✓	1	1	2	2	2	2	2	2	3	3	3	3	3	3	3			
	×	×	4	5	7	6	6	2	3	3	3	2	2	1	1	1	1			
	✓	×						1	1	1	2	3	3	3	3	3	3			
	×	✓	2	3		1	2	3	2	3	2	2		1	1	1	1			
devstral	✓	✓								1	2	2	4	4	7	7	7			
	×	×	1		1		1	6	5	3	3	2								
	✓	×									1	1								
	×	✓	8	10	9	10	9	3	4	5	3	4	4	5	2	2	2			
qwen3	✓	✓	4	7	7	8	9	4	5	5	5		3	3	3	3				
	×	×	2			1		4	2	1		-	4	2	2	1	-			
	✓	×		1	1	1	1	1	3	4	5		2	5	5	6				
	×	✓	4	2	2			1												

Pro (8 TPs) and Devstral (7 TPs). Maverick, GPT-5-nano, and Qwen3 having the lowest count of TPs.

Analyzing the evolution of the *classic-compliance (i.e., TP+FN)* through iterations shows that the iterative feedback improves only slightly the classic-compliance, i.e., Operation Model: 62/68/65/66/67, Scenario: 43/43/46/45/47, Diagram: 49/53/54/54/54. However, finer-grained analysis reveals that iterative feedback provides more substantial benefits, as discussed in the following subsections.

3.4.2 On the Violated Rules Count through Iterations

Figure 3 quantifies how iterative auditing reduces the number of total violations, as computed from classic auditors verdicts, that falls from 151 (iteration 1) to 69 (iteration 2) to 33 (iteration 3) to 35 (iteration 4) to 15 (iteration 5), confirming that reinjecting audit reports into GenAI generators consistently reduces the number of total violations and thus helps in reaching

Table 3: All violated rule categories with classic auditing.

Category	Stage	Violations	Violated Rules	GenAI Models with Violations	GenAI Models without violations
Graphical Syntactic	Diagram	77	LDR11 (×12), LDR12 (×12), LDR13 (×15), LDR14 (×13), LDR15 (×12), LDR16 (×13)	gpt-5-nano, llama-4-maverick, qwen3	gemini-2.5-flash, gemini-2.5-pro, gpt-5, gpt-5-mini, mistralai-devstral
Cross-artifact consistency	Scenario	49	LSC11, LSC14 (×16), LSC15 (×16), LSC17 (×16)	gpt-5-mini, gpt-5-nano, llama-4-maverick, qwen3	gemini-2.5-flash, gemini-2.5-pro, gpt-5, mistralai-devstral
Temporal	Scenario	45	LSC6 (×45)	gemini-2.5-pro, gpt-5, gpt-5-nano, mistralai-devstral, qwen3	gemini-2.5-flash, gpt-5-mini, llama-4-maverick
Valid format	Operation Model	37	JSON (×3), LDR0 (×3), LOM0 (×20), LSC0 (×11)	gpt-5-mini, gpt-5-nano, llama-4-maverick, mistralai-devstral, qwen3	gemini-2.5-flash, gemini-2.5-pro, gpt-5
Naming	Diagram	37	LDR28 (×27), LOM1 (×10)	gemini-2.5-pro, gpt-5, gpt-5-mini, gpt-5-nano, llama-4-maverick, qwen3	gemini-2.5-flash, mistralai-devstral
Quantitative	Operation Model	29	LDR1 (×2), LOM8 (×8), LOM9 (×13), LSC2 (×3), LSC4 (×3)	gemini-2.5-flash, gemini-2.5-pro, gpt-5-nano, llama-4-maverick, mistralai-devstral, qwen3	gpt-5, gpt-5-mini
Textual Syntactic	Diagram	23	LDR17 (×12), LDR25 (×11)	gemini-2.5-pro, gpt-5, gpt-5-mini, gpt-5-nano, llama-4-maverick, qwen3	gemini-2.5-flash, mistralai-devstral
Message Flow	Diagram	5	LDR20, LDR7, LOM4 (×2), LOM5	gpt-5-nano, qwen3	gemini-2.5-flash, gemini-2.5-pro, gpt-5, gpt-5-mini, llama-4-maverick, mistralai-devstral

compliance.

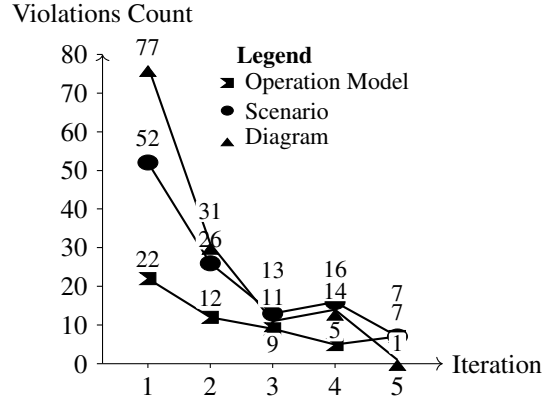

Figure 3: Evolution of classic auditor violations over iterations.

Table 3 reports at a finer-grained level the total violation count per rule category, in order to discuss on which rule categories are more difficult to reach compliance for which GenAI models.

Graphical Syntactic violations, audited during the Diagram stage, are the most frequent, i.e. 77 occurrences, affecting only gpt-5-nano, maverick and Qwen3. This category encompasses rules related to LUCIM domain-specific graphical constraints, such as the use of specific hex colors and shapes for actors, messages, and activation bars. All these rules are not typical UML sequence diagram constraints. All other GenAI models easily comply to this category by not violating these rules.

The *Message Flow* category is the least frequent, i.e. 10 occurrences, affecting only gpt-5-nano and Qwen3. This category encompasses rules that enforce a specific sequence of activation bars, and the direction of messages. All these rules are lucim-specific constraints,not typical UML sequence diagram constraints.

3.4.3 On Capabilities of GenAI Models by Tasks of the Process

Table 4 reports on the capabilities of the GenAI models selected for each task of the process, sorted by diagram generation classic-compliance score. This view complements the other tables by explicitly showing

Table 4: Capabilities of GenAI models for each process task (percentage of compliant artifacts).

Model	Audit Tasks				Generation Tasks			
	Operation Model	Scenario	Diagram	Overall	Operation Model	Scenario	**Diagram**	Overall
gemini-2.5-flash	93%	**100%**	**100%**	**98%**	**100%**	**100%**	**100%**	**100%**
gpt-5-mini	90%	90%	85%	89%	90%	90%	90%	90%
gpt-5	**100%**	50%	85%	74%	**100%**	50%	90%	80%
mistralai-devstral	94%	53%	**100%**	83%	90%	60%	90%	80%
gemini-2.5-pro	**100%**	**22%**	80%	49%	**100%**	**30%**	80%	70%
qwen3	86%	50%	31%	58%	90%	50%	30%	56%
llama-4-maverick	**36%**	53%	48%	46%	**40%**	50%	40%	43%
gpt-5-nano	55%	40%	**16%**	**30%**	60%	40%	**20%**	**40%**
Average	**82%**	57%	68%	66%	**85%**	59%	69%	70%

which GenAI models perform best at generation versus auditing and which tasks remain challenging.

Frontier, lightweight, and dev-specialized GenAI models consistently achieve 90-100% classic-compliance on the Operation Model Generation task, showing that most GenAI models can reconstruct a complete and structurally correct operation model from NetLogo code. GPT-5-nano and Llama 4 Maverick have difficulties for this task. GPT-5-nano misses events or misclassifies directions, and Llama 4 Maverick regularly produces structurally incomplete or invalid JSON.

The Scenario Generation task reveals a surprisingly different pattern. The lightweight (Flash and Mini) GenAI models achieve the best classic-compliance scores. GPT-5, Gemini 2.5 Pro, Qwen3, and Devstral frequently generate plausible scenarios but handle parameters in ways that violate LSC6 rules, resulting in lower compliance scores between 30-60%. These results suggest that, in our experiment, lightweight models are better at following rigid output schemas than more expressive larger models.

Diagram generation performance is the most important capability, as it produces the final artifact, i.e., the PlantUML diagram. Indeed, lower compliance in earlier generation tasks is acceptable as long as the final diagram achieves high compliance. Frontier and lightweight proprietary models outperform the other open-source qwen3 and maverick models and the ultra-lightweight nano model. These five GenAI models obtain 80–100% classic-compliance, which may be due to their expertise in the widespread PlantUML language. They also handle well LUCIM-specific conventions, i.e., specific message lines styles for input/output events, and activation bar ordering. Maverick and Qwen3 reach only 40% diagram compliance, with repeated confusion between input/output message syntax and mismanaged activation bars. GPT-5-nano performs worst (20%), accumulating multiple small violations (syntax, activations, and formatting).

Concerning auditing tasks, the two lightweight models (Flash and Mini) achieve the best compliance scores, as in the generation tasks. It should be noted that Devstral is particularly good at auditing the Diagram artifact, this is probably due to the fact that Devstral is a dev-specialized model and Diagram artifact is PlantUML code. Another observation is on the excellent auditing (100%) of the Operation Model artifacts by GPT-5 and Gemini Pro. Operation Models are tied to UML, that is common knowledge for these two frontier models. Overall audit performance reflects both the agreement rate between GenAI and classic auditors and the consistency with which different rule categories are handled across the three stages.

Our experiment shows that good performance on the final diagram generation task does not automatically imply uniform strength across all generation tasks. Some models, such as Gemini 2.5 Flash and GPT-5-mini, provide strong performance at each individual generation task, which makes them robust candidates for end-to-end reverse engineering. Frontier models (GPT-5 and Gemini Pro) perform poorly in Scenario generation but manage to self-correct in the last stage and reach high compliance on the Diagram stage. Others, such as Devstral and Qwen3, compensate weaker first-pass generations with effective iteration and self-correction, especially on the Operation Model and Diagram generation tasks, but remain fragile on Scenario generation. Finally, GPT-5-nano and Maverick illustrate that limited capacity or unstable behavior can be amplified across stages; they occasionally reach acceptable performance on one stage but fail to maintain compliance when constraints become more syntactic and graphical, as in the Diagram stage.

4 RELATED WORK

The lack of accessible source code and missing documentation elements in published agent-based models (Janssen et al., 2020) limits accessibility for domain experts who lack programming expertise. This gap motivates the need for automated reverse engineering approaches that can reconstruct domain-specific design models from simulation code, to bridge the abstraction gap between technical implementations and stakeholder communication.

Recent work has shown the potential of large language models (LLMs) in *UML modeling* contexts. Di Rocco et al. (Rocco et al., 2024) provide a comprehensive survey of LLM applications in UML contexts, highlighting challenges such as hallucination control and consistency maintenance, which align with our focus on compliance verification. Wang et al. (Wang et al., 2024) conducted a study examining how LLMs may assist students in UML modeling tasks, concluding that "at the current stage, LLMs cannot be considered a reliable tool for novice analysts in requirements analysis and software modeling."

The usage of LLMs for *reverse engineering* code into UML models has gained significant attention. Boronat et al. (Siala, 2024; Boronat and Mustafa, 2025) developed MDRE-LLM, a tool that analyzes Java codebases and applies LLMs to reverse engineer UML class diagrams. Campanello et al. (Campanello et al., 2025) investigated GPT-4's capabilities in reverse engineering class diagrams from Java.

Pajo et al. (Pajo, 2025) proposed *context engineering* as an approach beyond prompt engineering, promoting conversational history, tool integration, and context window optimization. In our approach, we use some context engineering techniques, like persona- and rule-based prompt injections, as well as cross-step context sharing, where outputs from earlier agents are reused as inputs to later agents.

A fundamental question in our work is whether LLMs can reliably follow and enforce *domain-specific rules*. Ling et al. (Ling et al., 2024) provide a survey of domain specialization techniques for LLMs. Mu et al. (Mu et al., 2024) examined whether LLMs can follow simple rules.

Zhu et al. (Zhu et al., 2024) demonstrated that LLMs can learn and apply rules through adequate training and prompting strategies. Lastly, Kogler et al. (Kogler et al., 2024) focused on reliable generation of formal specifications using LLMs, emphasizing consistency checks and validation mechanisms.

While existing work demonstrates LLM potential in SE tasks, gaps remain: most approaches use single-agent solutions, domain-specific reverse engineering is largely unexplored, and systematic compliance verification with domain-specific rules is insufficiently addressed.

5 CONCLUSION

The goal of this work is to open a research direction on LLM-based reverse engineering into domain-specific models. This work-in-progress position paper introduces a generative AI agent orchestration approach that transforms NetLogo simulation code into compliant domain-specific models. By coordinating specialized AI agents and integrating domain-specific rule-based auditing directly into the transformation orchestration process, we ensure that generated artifacts reliably meet domain-specific rules.

Our results show that some models consistently outperform the others. Gemini 2.5 Flash and Pro, GPT-5 and mini, and Devstral achieve the highest compliance, showing that orchestrated LLM-based reverse engineering can reach high compliance rates. Although iterations improve compliance, Maverick, Qwen3, and GPT-5-nano perform unevenly, with Qwen3 and Maverick relatively outperforming GPT-5-nano.

This experiment establishes a foundation for transforming simulation code into domain-specific models. Future work will extend the process by including metamodel specifications and Object Constraint Language (OCL) constraints, enabling generative agents to automatically derive natural-language compliance rules from these formal constraints and validate the resulting models. We will also integrate deterministic auditing tools to strengthen compliance verification and complement LLM-based audits. Another direction involves coupling the orchestration process with simulation engines to verify the behavioral correctness of generated scenarios.

REFERENCES

(2025). NetLogo Models Library. https://ccl.northwestern.edu/netlogo/models/.

(2025). PlantUML Language Reference Guide. https://plantuml.com/guide.

Boronat, A. and Mustafa, J. (2025). MDRE-LLM: A Tool for Analyzing and Applying LLMs in Software Reverse Engineering. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 850–854. IEEE.

Campanello, V., Shahbaz, S., Indykov, V., and Strüber, D. (2025). On the Use of GPT-4 in the Reverse Engineer-

ing of Class Diagrams. *The Journal of Object Technology*, 24(2):2:1.

Chikofsky, E. and Cross, J. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17.

Cross, J. H., Chikofsky, E. J., and May, C. H. (1992). Reverse Engineering*. In Yovits, M. C., editor, *Advances in Computers*, volume 35, pages 199–254. Elsevier.

Hermann, S. and Fehr, J. (2022). Documenting research software in engineering science. *Scientific Reports*, 12.

Janssen, M. A., Pritchard, C., and Lee, A. (2020). On code sharing and model documentation of published individual and agent-based models. *Environmental Modelling & Software*, 134.

Kogler, P., Falkner, A., and Sperl, S. (2024). Reliable Generation of Formal Specifications using Large Language Models.

Ling, C., Zhao, X., Lu, J., Deng, C., Zheng, C., Wang, J., Chowdhury, T., Li, Y., Cui, H., Zhang, X., Zhao, T., Panalkar, A., Mehta, D., Pasquali, S., Cheng, W., Wang, H., Liu, Y., Chen, Z., Chen, H., White, C., Gu, Q., Pei, J., Yang, C., and Zhao, L. (2024). Domain Specialization as the Key to Make Large Language Models Disruptive: A Comprehensive Survey.

Mu, N., Chen, S., Wang, Z., Chen, S., Karamardian, D., Aljeraisy, L., Alomair, B., Hendrycks, D., and Wagner, D. (2024). Can LLMs Follow Simple Rules?

OMG (2013). Business Process Model and Notation (BPMN), version 2.0.2. Full Specification formal/13-12-09, Object Management Group.

OMG (2017). Unified Modeling Language: Superstructure (UML), v. 2.5.1. Full Specification formal/17-12-05, Object Management Group.

Pajo, P. (2025). Context Engineering: Enhancing Large Language Model Performance Through Comprehensive Contextual Management.

Ries, B. (2026). Code repository: Netlogo to lucim reverse engineering. https://github.com/benoitries/code-modelsward-2026-netlogo-to-lucim.

Rocco, J. D., Ruscio, D. D., Sipio, C. D., Nguyen, P. T., and Rubei, R. (2024). On the use of Large Language Models in Model-Driven Engineering.

Siala, H. A. (2024). Enhancing Model-Driven Reverse Engineering Using Machine Learning. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 173–175.

Sommerville, I. (2015). *Software Engineering, 10th Edition*. Pearson, Boston, Munich.

Wang, B., Wang, C., Liang, P., Li, B., and Zeng, C. (2024). How LLMs Aid in UML Modeling: An Exploratory Study with Novice Analysts.

Wilensky, U. and Rand, W. (2015). *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo*. The MIT Press.

Zhu, Z., Xue, Y., Chen, X., Zhou, D., Tang, J., Schuurmans, D., and Dai, H. (2024). Large Language Models can Learn Rules.