

# Constraint decay: The Fragility of LLM Agents in Backend Code Generation

Francesco Dente\*  
EURECOM, France  
francesco.dente@eurecom.fr

Dario Satriani\*  
University of Basilicata, Italy  
dario.satriani@unibas.it

Paolo Papotti  
EURECOM, France  
papotti@eurecom.fr

## Abstract

Large Language Model (LLM) agents demonstrate strong performance in autonomous code generation under loose specifications. However, production-grade software requires strict adherence to structural constraints, such as architectural patterns, databases, and object-relational mappings. Existing benchmarks often overlook these non-functional requirements, rewarding functionally correct but structurally arbitrary solutions. We present a systematic study evaluating how well agents handle structural constraints in multi-file backend generation. By fixing a unified API contract across 80 greenfield generation tasks and 20 feature-implementation tasks spanning eight web frameworks, we isolate the effect of structural complexity using a dual evaluation with end-to-end behavioral tests and static verifiers. Our findings reveal a phenomenon of constraint decay: as structural requirements accumulate, agent performance exhibits a substantial decline. Capable configurations lose 30 points on average in assertion pass rates from baseline to fully specified tasks, while some weaker configurations approach zero. Framework sensitivity analysis exposes significant performance disparities: agents succeed in minimal, explicit frameworks (e.g., Flask) but perform substantially worse on average in convention-heavy environments (e.g., FastAPI, Django). Finally, error analysis identifies data-layer defects (e.g., incorrect query composition and ORM runtime violations) as the leading root causes. This work highlights that jointly satisfying functional and structural requirements remains a key open challenge for coding agents.

## 1 Introduction

The landscape of software engineering is undergoing a fundamental shift, driven by the increasing use of Large Language Model (LLM) agents capable of autonomously generating code (Wang et al., 2025; SWE-agent Team, 2025; Yang et al., 2024). These agents have demonstrated remarkable proficiency in “shallow” generation tasks, where objectives are broad and requirements are loosely defined (Qwen Team, 2026; OpenAI, 2026; Hong et al., 2024; Kimi, 2026). This flexibility is well suited to a range of development scenarios, such as rapid prototyping, demo applications, frontend interfaces, and proof-of-concept implementations, where structural decisions can be left to the agent. However, this permissiveness becomes a liability when the target is a production-grade backend. Such systems must satisfy functional and non-functional requirements by exposing endpoints that adhere to API contracts, conform to architectural patterns, integrate with databases, and operate through designated ORM layers. The impact of these constraints on agent performance remains largely unexplored.

Recent benchmarks evaluate LLM agents on greenfield code generation (build an application from scratch) or GitHub Issue resolution, yet none adequately captures the challenges of constrained multi-file backend development. Existing efforts either focus on resolving specific issues given a description and existing codebases (Jimenez et al., 2024; Deng

---

\*Equal contribution.

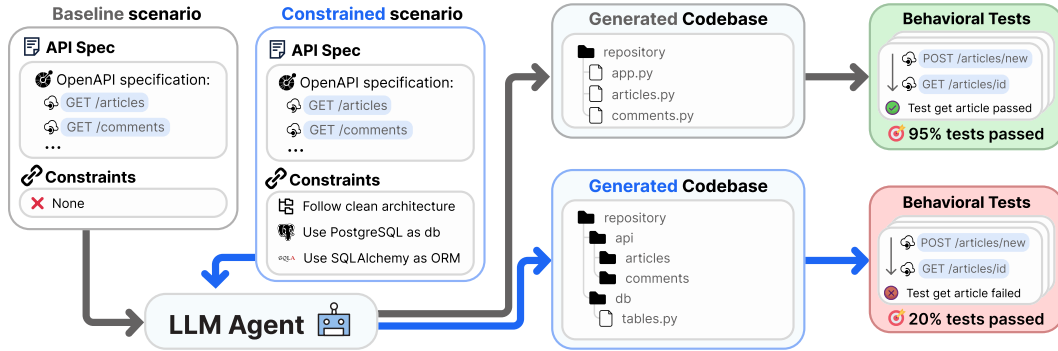


Figure 1: Given the same API specification, an LLM agent produces a functional codebase under no constraints (top), but performance degrades with structural constraints (bottom).

et al., 2025), reward unconstrained generation from under-specified natural language (NL) prompts (Luo et al., 2026), directly target single-file solutions via prompting (Vero et al., 2024), or provide skeleton code for agents to complete (Zhao et al., 2025), overlooking the effect of systematically varying the degree of structural constraint imposed on the agent.

To investigate this scenario, we conduct a systematic study to assess LLM agents in constrained backend generation tasks, leveraging OpenAPI specifications as a structured way of expressing functional requirements. Our design fixes the API contract and deploys a shared suite of end-to-end behavioral tests across all conditions, isolating the effect of *four non-functional constraint dimensions*: framework choice, architectural pattern, database backend, and ORM integration. By systematically layering these constraints, we measure performance degradation as task complexity increases, from baseline generation to fully specified systems requiring an architecture with a prescribed database and ORM layer.

Our empirical findings reveal that as the density of non-functional constraints increases, agent performance exhibits a decline. Figure 1 illustrates this phenomenon of constraint decay: agents that easily generate functional codebases under baseline conditions fail sharply when strict architectural and data-layer rules are enforced. We demonstrate that even state-of-the-art agents struggle to maintain functional compliance when unconstrained generation is replaced by rigorous adherence to structured specifications.

In this paper, we report the following contributions:

- **Evaluation Methodology.** An evaluation pipeline based on OpenAPI specifications and behavioral tests that decouples functional correctness from structural compliance. We open-source 80 greenfield generation tasks and 20 feature-implementation tasks.
- **Constraint Decay (RQ1).** The empirical identification of *constraint decay*: as explicit structural requirements (architecture, database, ORMs) accumulate, agent performance degrades, with capable models losing an average of 30 points in assertion pass rates (A%).
- **Framework sensitivity (RQ2).** A framework analysis revealing significant disparities in agent success under identical API contracts. Agents do better in lightweight, explicit frameworks (e.g., Flask) than in convention-heavy environments (e.g., FastAPI).
- **Root Cause Taxonomy (RQ3).** An extensive trajectory and error analysis revealing that data-layer defects, specifically incorrect query composition and ORM runtime violations, are the leading root causes, driving  $\sim 45\%$  of agent logic failures.

We fully open-source the evaluation pipeline, task suite, agent trajectories, and analysis scripts at <https://anonymous.4open.science/r/constraint-decay>.

## 2 Related Work

**Greenfield app generation.** Recent benchmarks have begun to evaluate LLM agents on generating entire apps from scratch, yet none adequately captures the challenges of constrained multi-file, repository-level backend development. Luo et al. (2026) measure output

against existing test suites from well-known libraries but reward unconstrained generation from under-specified prompts. Seo et al. (2026) target repository generation for reproducing research papers, while Zhao et al. (2025) provide skeleton libraries for agents to complete. Ding et al. (2025) evaluate coding agents on repository generation from reverse-engineered natural-language specifications of Python codebases. Neither targets backend systems nor systematically varies the degree of structural constraint. Vero et al. (2024) is closer to our setting, tasking LLMs with generating backend services from OpenAPI specifications across multiple frameworks and languages, with evaluation through end-to-end functional and security tests. However, it targets single-file solutions via direct prompting rather than multi-file, repositories generated through agentic interaction, and does not study how layering architectural constraints affects generation quality.

Our work fixes a single API contract and systematically layers structural constraints across eight frameworks, evaluating implementations against a shared suite of end-to-end behavioral tests fully decoupled from internal code structure, enabling fair comparison regardless of how the agent organizes its solution.

**Benchmarks for repository-level coding tasks.** Issue-based datasets, such as SWE-Bench (Jimenez et al., 2024), offer realistic software engineering environments, yet current agents are approaching saturation on this benchmark. Extensions broaden the scope through multi-language support, improved resistance to overfitting, and scalable environment construction (Deng et al., 2025; Rashid et al., 2025; Zan et al., 2025; Jain et al., 2025), while complementary benchmarks evaluate the ability of agents to add new functionalities (Li et al., 2025; Chen et al., 2025; Miserendino et al., 2025). Other works focus on repository-level function or line completion (Liu et al., 2024; Ding et al., 2023; Zhang et al., 2023) or scalable sandbox-based evaluation Xie et al. (2025).

In our work, we task agents with generating a backend system from scratch, rather than modifying an existing codebase, while systematically increasing structural constraints.

**Software Engineering Agents.** SWE-bench has catalyzed the development of agent architectures. OpenHands provides a framework with tools for file I/O, code execution, and version control in a ReAct-style planner for multi-step reasoning (Wang et al., 2025). SWE-Agent emphasizes configurable agent-computer interfaces for interacting with isolated environments (Yang et al., 2024); its variant, Mini-SWE-Agent, reduces the scaffold to 100 lines of Python with bash commands. Xia et al. (2024) forgo autonomous planning and decompose issue resolution into a fixed pipeline: localization, patch generation, and validation. Our study leverages two of these open-source architectures (Mini-SWE-Agent and OpenHands) to measure how current agents interact with structural constraint density.

### 3 Methodology

To isolate and evaluate the impact of structural constraints on code generation, we propose an evaluation methodology inspired by Behavior-Driven Development (North et al., 2006). The principle of our approach is to hold the functional specification constant while systematically varying the non-functional requirements (i.e., structural constraints). This experimental design isolates the effect of the constraints from the logical complexity of the task, thus enabling controlled, equitable comparisons across agents and models.

#### 3.1 Generation Task Design

We define a greenfield *generation task*: given a comprehensive NL description of a target service, an LLM agent must synthesize a complete, specification-compliant REST API from scratch, i.e., from an empty repository. To execute this task, the agent is provided with an input prompt with four components: (i) the API specification, (ii) the structural constraints to be enforced, (iii) the mandatory files required to execute the generated code, and (iv) a description of the pipeline used to assess the submission (full example in Appendix E).

**API specification.** We adopt the OpenAPI 3.0 specification<sup>1</sup> as the standardized functional requirement format for all tasks. OpenAPI is a machine-readable, language-agnostic interface description standard widely used in industry for API-first development. This structured format yields two methodological advantages over NL descriptions. First, it removes specification ambiguity: every functional behavior, endpoint signature, request body, response schema, and status code is formally defined. Second, it facilitates the preemptive definition of a behavioral test suite for evaluating any compliant implementation, irrespective of the underlying language, framework, or internal architecture. This approach guarantees a level of evaluation flexibility and behavioral coverage that traditional unit-test-based pipelines cannot achieve, as unit tests are inherently coupled to specific implementation details.

Our specification is derived from the RealWorld Conduit API<sup>2</sup>. The specification encompasses 19 standard Create, Read, Update, and Delete (CRUD) operations distributed across five resource groups (articles, comments, users, profiles, and tags). The choice of this open-source specification is deliberate. First, it defines operations at a highly granular level while remaining within the context window limits of LLMs. Moreover, the relative simplicity of the functional requests, combined with the models’ probable prior exposure to the specification during pre-training, ensures a precise disentanglement between the impact of structural constraints and baseline functional complexity.

We intentionally fix a single API contract to maximize internal validity: holding the functional target constant allows us to isolate the effect of structural constraints, framework, and agent scaffold without conflating them with differences in task semantics.

**Framework coverage.** Framework is treated as an experimental factor with eight discrete levels, allowing us to test whether the same functional specification exhibits framework-dependent difficulty under identical evaluation conditions. We selected popular frameworks across two of the most widespread language runtimes: Python 3.12 (Flask, FastAPI, Django, aiohttp) and Node.js 20 (Express, Fastify, Hono, Koa). Both runtimes are built upon dynamic and multi-paradigm programming languages for maximal freedom during code generation.

### 3.2 Constraint Dimensions

To systematically evaluate the effect of constraints, we define three orthogonal non-functional axes, which can be either included in or omitted from the code specification.

**Architectural pattern.** The code architecture is fixed: it must be organized following the Clean Architecture pattern (Martin, 2017). Rather than spontaneously define a monolithic or arbitrary file structure, the agent must split the codebase into four domain layers with strict top-down dependency direction: routes/handlers, services/use cases, models/entities, and repository/data access. Each layer must reside in its own directory.

**Database backend.** The database layer is fixed: PostgreSQL or SQLite. The agent must persist data using a specific database engine. For PostgreSQL tasks, a pre-configured instance is provided in the execution environment; for SQLite, no external service is needed. In both cases, the schema must be created automatically at server startup.

**ORM integration.** The Object-Relational Mapper (ORM) framework is fixed: SQLAlchemy for Python, Sequelize for Node. The agent must adhere to the idiomatic query logic of the prescribed framework, instead of falling back to raw SQL or alternative mapping.

Unlike the three axes above, the web framework is always specified: an agent must target some framework to produce executable code. We therefore treat framework as a fixed baseline constraint present at every level. To quantify the effect of additional structural complexity, we systematically compose the three axes into task variants of increasing specification density (L0 to L3), as in Table 1. Each variant is instantiated across all 8 frameworks, yielding 80 generation tasks in total.

Although simple and helpful for human developers, these non-functional requirements isolate constraint composition as an added challenge for LLM agents.

<sup>1</sup><https://spec.openapis.org/oas/v3.0.0.html>

<sup>2</sup><https://realworld-docs.netlify.app/>

Level	Active Constraints	# Variants
L0	WF only (baseline)	8
L1	WF+Arch   WF+SQLite   WF+PG	24
L2	WF+Arch+SQLite   WF+Arch+PG   WF+SQLite+ORM   WF+PG+ORM	32
L3	WF+Arch+SQLite+ORM   WF+Arch+PG+ORM	16

Table 1: Constraint levels and task variants. WF = Web Framework (always active), Arch = Clean Architecture, PG = PostgreSQL. Variants within a level are separated by |.

### 3.3 Evaluation Pipeline

To guarantee reproducibility and preclude state contamination, each task is executed in an isolated Docker environment following a *build* and *evaluate* pipeline. In the build phase, the agent operates within a first container, where it can code, build, and start the server. Next, in the evaluation phase, the agent code patch is applied in a pristine container and the behavioral test suite is executed. In both phases, a container-dedicated, ephemeral PostgreSQL 16 instance is available (full infrastructure details in Appendix I).

**Behavioral testing.** Since all task instances target a unified OpenAPI specification, all implementations are evaluated against a unified HTTP behavioral test suite covering the whole specification. The suite comprises 32 requests covering all 19 API endpoints and 291 assertions that verify response structure, types, status codes, and state transitions across a stateful sequence of interdependent CRUD operations (breakdown in Appendix G). This is a methodological choice: by testing the API behavior rather than its implementation, we decouple functional evaluation from internal code structure.

**Verifier functions.** To separate functional correctness from structural compliance, we introduce orthogonal verifier functions that detect whether the required architecture, database, and ORM constraints are actually satisfied (details in Appendix C). (i) **Architecture verifier:** assesses the compliance to the four canonical Clean Architecture layers (routes, services, repositories, models) and scans every import to verify dependency order. (ii) **Database verifier:** scans the source lines for usage evidence of the imposed database engine while checking that no alternative engines are used; lock files are excluded. (iii) **ORM verifier:** detects whether the designated Object-Relational Mapper is used rather than raw SQL or framework-native data layer.

We define task success as the intersection of behavioral validity and constraint compliance. A completed task passes the entire behavioral test suite and satisfies each applicable verifier.

## 4 Experimental Setup

**Metrics.** We define our primary evaluation axes as follows. **Assert%** (A%): the mean fraction of behavioral test assertions passed across all runs of a given configuration. It captures partial progress, distinguishing configurations that slightly miss full compliance from largely non-functional implementations. We report A% as the primary metric, since the all-or-nothing nature of pass@1 over the entire test suite amplifies noise. **Pass@k** (using the unbiased estimator of Chen et al., 2021): the probability that at least one of  $k$  independent samples satisfies the complete set of behavioral and structural constraints.

Each task is executed for  $n=3$  independent trials per agent-model configuration.

**Agents and Models.** We evaluate two popular and open-source agent architectures of different tooling complexity. (i) **Mini-SWE-Agent:** a minimal scaffold (~100 lines of Python) that interacts with the environment exclusively through bash commands (SWE-agent Team, 2025). Despite its simplicity, it achieves strong performance on SWE-Bench with frontier LLMs (>70%). (ii) **OpenHands:** a full-featured agent framework providing dedicated tools for file editing, terminal execution, code search, and task tracking (Wang et al., 2025).

To ensure experimental validity and calibrate the agents for code generation, we apply systematic adjustments to the scaffold configuration through prompt refinement and pipeline adjustments. We inject a re-aligned system prompt into Mini-SWE-Agent to steer its execution trace toward a generative pipeline. Additionally, to mitigate premature halting behaviors (Ding et al., 2025) and enforce complete evaluation trajectories, we introduce a termination criterion for OpenHands (details in Appendix J). For both agents, we impose a highly permissive upper bound on the maximum number of iterations (300 for Mini-SWE-Agent and 200 for OpenHands) while leaving the cost limit unbounded.

We pair each scaffold with models spanning a broad capability range. We distinguish four tiers. (i) **Small open agentic models**: compact code-specialists, i.e., Devstral-Small (24B) (Rastogi et al., 2025) and Qwen3-Coder-Next (80B) (Cao et al., 2026). (ii) **Large open instruct model**: general purpose architectures, i.e., Qwen3-235B-Instruct (235B total / 22B active) (Yang et al., 2025). (iii) **Large open agentic models**: SOTA open-weight models, specifically MiniMax-M2.5 (MiniMax, 2026) and Kimi-K2.5 (Kimi, 2026). (iv) **Closed models**: frontier proprietary engines, i.e., GPT-5-mini (OpenAI, 2025) and GPT-5.2 (OpenAI, 2025). The larger models rank among the top 15 on the SWE-Bench Verified leaderboard.

**Cost-driven evaluation scope.** OpenHands’s richer tool set leads to substantially larger context windows than Mini-SWE-Agent’s scaffold:  $12.9\times$  more tokens for GPT-5-mini. To reduce cost for generation tasks we restrict Kimi-K2.5 and GPT-5.2 to Mini-SWE-Agent only. Moreover, as in Zhao et al. (2025), for these two LLMs and MiniMax-M2.5 we use a *representative task subset* (marked \* in Table 2). The subset spans two frameworks per runtime: aiohttp and FastAPI (Python), Express and Fastify (Node), and one variant per constraint level: L0, L1 (Clean Architecture), L2 (Clean Architecture + PostgreSQL), L3 (Clean Architecture + PostgreSQL + ORM), yielding 16 tasks and 48 runs per model-agent pair. Across the full study, our evaluation used  $\sim 5$  billion tokens (details in Appendix H).

## 5 Results

We organize the analysis around three research questions.

**RQ1.** How does the accumulation of structural constraints affect LLM agent performance (§5.1)? **RQ2.** Does the choice of web framework significantly influence agent success under the same API contract (§5.2)? **RQ3.** What are the primary root causes of agent failure in constrained backend generation (§5.3)?

### 5.1 RQ1: Effect of Constraint Accumulation

Table 2 summarizes the main results. Each cell reports assertion pass rate A% and pass@1 (%), computed per-task and averaged across tasks. Three models (MiniMax-M2.5, Kimi-K2.5, GPT-5.2) were evaluated on the 16-task subset due to cost constraints. To validate this choice, we re-evaluated the six full-set configurations (3 models  $\times$  2 agents, excluding Devstral-Small) on the same 16 tasks: A% scores correlate at Pearson  $r = 0.98$  and Spearman  $\rho = 0.95$  ( $p < 10^{-12}$ ,  $N = 24$ ). Specifically, we compared each full-set model’s A% on all 80 tasks against its A% on the same 16-task subset, obtaining 24 paired observations (6 model-agent configurations  $\times$  4 constraint levels) that show near-perfect agreement in both absolute scores and ranking order (details in Appendix A).

Without structural constraints (L0), agents perform well: Qwen3-Coder-Next, MiniMax-M2.5, and Kimi-K2.5 exceed 85% A%, confirming that compact code-specialists such as Qwen3-Coder-Next handle end-to-end backend generation when given full architectural freedom. Smaller agentic models and non-specialized ones struggle even at L0. Given its near-zero Assert % (see Table 8), we exclude Devstral-Small from the remainder of the discussion.

**Overall decay.** Structural constraints cause steep, universal performance degradation. Restricting to the eight capable configurations (L0 A% > 50%) in Table 2, A% drops by an average of 30 percentage points from L0 to L3: a relative loss of 40% of baseline performance.

Agent	Model	Assert% (A%)				pass@1				$\Delta A\%$
		L0	L1	L2	L3	L0	L1	L2	L3	
Mini-SWE	GPT-5-mini	51.7	46.8	27.1	23.7	25.0	22.2	16.7	4.2	-28.0
OpenHands	GPT-5-mini	65.8	63.2	56.7	52.2	62.5	47.2	39.6	33.3	-13.6
Mini-SWE	Qwen3-Coder-Next	86.4	66.4	52.6	46.1	45.8	25.0	13.5	6.2	-40.2
OpenHands	Qwen3-Coder-Next	73.0	51.7	42.7	27.6	45.8	16.7	16.7	4.2	-45.5
Mini-SWE	Qwen3-235B-A22B	29.6	10.7	9.1	2.3	20.8	1.4	2.1	0.0	-27.3
OpenHands	Qwen3-235B-A22B	26.2	17.7	3.1	0.8	12.5	2.8	0.0	0.0	-25.4
Mini-SWE	MiniMax-M2.5*	88.6	92.5	66.8	58.3	58.3	50.0	33.3	25.0	-30.3
OpenHands	MiniMax-M2.5*	95.6	97.0	87.3	78.6	41.7	41.7	16.7	8.3	-17.0
Mini-SWE	Kimi-K2.5*	85.4	70.9	62.9	53.7	66.7	41.7	8.3	33.3	-31.7
Mini-SWE	GPT-5.2*	78.2	49.3	27.1	48.0	58.3	33.3	0.0	25.0	-30.2

Table 2: Assertion pass rate A% and pass@1 across constraint levels. L0 = baseline; L1-L3 impose increasingly specific structural requirements. \* = evaluated on task subset.

Constraint	Avg $\Delta$ (pp)	pass@1 (%)		
		Model	Mini-SWE	OpenHands
Clean architecture	$-9.1 \pm 1.6$	GPT-5-mini	15.0	48.3
PostgreSQL	$-19.3 \pm 2.5$	GPT-5.2 <sup>†</sup>	50.0	55.0
SQLite	$-14.3 \pm 2.5$	MiniMax-M2.5	46.7	38.3
SQLAlchemy	$-1.5 \pm 2.1$	Qwen3-Coder-Next	16.7	13.3
Sequelize	$-0.6 \pm 2.2$			

(a) Marginal effect of each constraint on A%, via matched-pair differences (+- stderr of the mean).

(b) pass@1 (%) on feature implementation tasks. <sup>†</sup>Single run instead of 3 for GPT-5.2 only.

Table 3: (a) Marginal constraint effects. (b) Feature implementation results.

The worst case, OpenHands + Qwen3-Coder-Next, loses 45 pp (62% of its L0 score), while the most resilient configuration, OpenHands + MiniMax-M2.5, drops 17 pp.

The gap between A% and pass@1 is consistently large: even the strongest L3 configuration (OpenHands + MiniMax-M2.5) reaches 78.6% A% but only 8.3% pass@1. Because a single assertion failure among the entire suite zeroes out a run, pass@1 is noisy for evaluating generation tasks; A% captures partial progress more stably and is therefore the primary metric to look at. This also underscores how LLM agents still lack the cross-file consistency and constraint adherence needed for deployment without manual intervention.

The variance of the drop across levels is substantial, so the precise effect size should be interpreted with caution. However, the central qualitative finding is robust: adding structural constraints consistently worsens performance.

**Scaffold effect.** The choice of agent scaffold has a substantial impact. OpenHands outperforms Mini-SWE-Agent for GPT-5-mini and MiniMax-M2.5 yet underperforms for Qwen3-Coder-Next; however, the constraint-decay trend is consistent across both scaffolds.

**Per-constraint analysis.** To isolate the marginal effect of each constraint, we adopt a matched-pair design: for every constraint  $c$ , we identify all task pairs that differ by exactly  $c$  while keeping framework and all other constraints identical, then compute  $\Delta(A\%) = A\%_{\text{with } c} - A\%_{\text{without } c}$  and average across all pairs and model-agent configurations. Table 3a reports the results. Specifying a database engine (which is fundamental for a production-ready backend) is by far the most impactful constraint. Clean architecture also adds a significant penalty, consistent with the overhead of enforcing layered separation. ORM’s effects are smaller and, for GPT-5.2, the ORM constraint even reduces ambiguity by making the intended data-access pattern more explicit.

**Verifier impact on A%.** All A% scores in Table 2 incorporate structural compliance: when a run violates any applicable verifier, its assertion score is set to zero before averaging. To quantify the impact of this choice, we also compute A% without verifier enforcement. The

		GPT-5-mini		Qwen3-Coder-Next		Qwen3-235B		
		M	O	M	O	M	O	Avg
Web framework	Express	60.0	84.2	71.8	60.8	14.9	16.7	51.4
	Koa	49.2	85.0	66.4	78.5	11.3	13.6	50.7
	Flask	37.8	72.6	74.9	68.9	29.4	12.0	49.3
	aiohhttp	34.7	64.3	48.0	56.4	11.1	15.8	38.4
	Fastify	35.5	55.8	60.2	28.2	4.0	6.5	31.7
	Django	33.7	66.5	32.1	17.4	0.7	2.1	25.4
	FastAPI	21.4	24.7	63.1	28.9	3.6	3.6	24.2
	Hono	6.1	15.9	54.0	24.1	6.9	4.3	18.5

Table 4: Assertion pass rate A% by framework, aggregated across constraint levels. M = Mini-SWE-Agent; O = OpenHands.

two variants differ by at most 2.7 pp on any single configuration, and the average L0→L3 drop changes only from 28 to 30 pp. This confirms that the constraint-decay signal is driven by genuine functional failures rather than verifier artifacts (full comparison in Appendix D).

**Sanity check: feature implementation tasks.** A natural concern is whether constraint decay is an artifact of the synthetic greenfield setting. To test this, we construct 20 feature implementation tasks by ablating feature groups (164–2,604 lines across 2–46 files) from community-maintained RealWorld repositories (actual implementations of the API contract used in the generation tasks) that already embed a layered architecture, PostgreSQL, and an ORM, effectively simulating an L3 condition with implicit constraints. Each agent must read the existing codebase, infer its conventions, and re-implement the missing functionality (full task design in Appendix F). As shown in Table 3b, performance remains low: only GPT-5.2 exceeds 50% pass@1. While we cannot isolate structural constraints as the sole cause of difficulty in this setting, the results confirm that the challenges observed in greenfield generation are not artifacts of asking agents to build repositories from scratch; they persist when agents must infer and respect constraints from an existing codebase.

*Takeaways for RQ1.* Structural constraints cause steep, universal decay: the best models lose 30 pp A% from L0 to L3 on average. Database constraints drive most of this decline.

### 5.2 RQ2: Framework Sensitivity

Since framework is the one constraint present at every level, a natural question is how much the choice of framework matters. All eight frameworks implement the same API contract and share the same test suite, enabling a controlled comparison. Table 4 reports A% by framework for three models, each with both scaffolds, aggregated across constraint levels.

**Framework influence.** Express, Koa and Flask form a clear top tier (avg. 51.4%, 50.7%, 49.3% A%), sharing a minimal, explicit API surface with no implicit conventions. Their ranking shifts by configuration: Koa and Express dominate with OpenHands + GPT-5-mini, while Flask leads with Qwen3-Coder-Next. Django and FastAPI are penalized by implicit configuration: Django’s convention-driven structure and auto-discovery, FastAPI’s type-hint-driven validation. Hono trails despite a comparable API surface to Express, likely because Hono targets edge runtimes and requires a compatibility adapter on Node.js (the runtime used in our study), a setup step that may be underrepresented in training data.

*Takeaways for RQ2.* Framework choice is a major sensitivity axis: lightweight frameworks form a top tier at ~50% avg. A%, while convention-heavy ones trail by 25–32 points.

### 5.3 RQ3: Root Causes of Failure

We perform a failure analysis on unsuccessful runs for Qwen3-Coder-Next (194/240 failed, full 80-task set) and MiniMax-M2.5 (28/48 failed, 16-task subset), both with Mini-SWE-Agent. Following established approaches (Deng et al., 2025), we use GPT-5.2 (temperature

Failure category	Qwen	MMx	Logic-error subcat.	Qwen	MMx
<i>Logic error</i>	70.6	71.4	Framework idiosyncrasy	9.5	50.0
Server startup failure	12.4	21.4	Incorrect query logic	25.5	15.0
Incomplete implementation	9.3	3.6	DB / ORM runtime error	21.2	15.0
Schema / format error	3.6	3.6	Auth misconfiguration	22.6	5.0
Stuck in loop	3.1	–	Business logic defect	11.7	10.0
Constraint violation	1.0	–	State propagation failure	9.5	5.0
<b>Total failed runs</b>	<b>194</b>	<b>28</b>	<b>Total logic errors</b>	<b>137</b>	<b>20</b>

Table 5: Failure taxonomy for Qwen3-Coder-Next and MiniMax-M2.5 with Mini-SWE-Agent. Coarse categories (left) show % of failed runs; subcategories for *logic errors* only.

0) to classify each failure based on the last 20 trajectory turns, behavioral test results, server logs, and static verifier outputs. We define a two-level taxonomy (Table 5): six coarse categories (left), with logic errors only further sub-classified into six root causes (right). We validate the judge on a stratified sample of 50 Qwen3-Coder-Next logic errors against manual labels, obtaining Cohen’s  $\kappa = 0.975$ , aided by the largely unambiguous signal from server logs and test output (details in Appendix B).

**Coarse profile.** Both models share a near-identical distribution: logic errors account for  $\sim 71\%$  of failures, where the server starts and registers correct routes but behaves incorrectly, consistent with the persistent gap between `Assert%` and `pass@1` in §5.1. Server startup failures rank second (12–21%); remaining categories together account for less than 17%.

**Logic-error root causes.** We classify the 157 logic errors into six root-cause categories: *incorrect query logic*: SQL queries execute but return wrong results due to incorrect joins, filters, or dialect-incompatible operators; *DB/ORM runtime error*: query logic is conceptually correct but crashes from ORM API misuse; *auth misconfiguration*: broken token handling or header parsing causes 401 responses; *framework idiosyncrasy*: correct application code is blocked by an unhandled framework-specific default; *business logic defect*: infrastructure operates correctly but domain behavior deviates from the specification; *state propagation failure*: mutations succeed but their effects are not reflected in subsequent reads.

MiniMax-M2.5’s logic errors are dominated by a single *framework idiosyncrasy* (50.0%): Fastify’s rejection of empty-body POST requests, amplified by the smaller subset. Setting this aside, both models converge on a shared failure profile led by data-layer defects. *Incorrect query logic* accounts for 25.5% of Qwen3-Coder-Next’s logic errors and *DB/ORM runtime errors* add another 21.2%; MiniMax-M2.5 shows the same two categories at 15.0% each. Together, these data-layer failures confirm that database constraints impose the steepest penalty. *Auth misconfiguration* is a notable model-specific weakness for Qwen3-Coder-Next (22.6% vs. 5.0%), primarily from incorrect token-prefix parsing. The prevalence of data-layer defects does not contradict the low marginal effect of ORM constraints observed in Table 3a. Because the marginal effect of ORMs is computed relative to a baseline that already includes a database, it merely measures the cost of forcing an ORM in place of raw SQL—not the cost of data-layer interaction itself. The underlying difficulty of correctly interfacing with the data layer is high in both scenarios; it simply manifests as ORM runtime errors when the ORM constraint is active, and as raw SQL errors when it is not.

*Takeaways for RQ3.* Logic errors dominate failures for both models ( $\sim 71\%$ ), with data-layer defects (incorrect query composition and ORM runtime errors) as the leading root causes.

## 6 Conclusion

Our systematic study exposes a phenomenon of *constraint decay* in LLM-based coding agents. While current models excel at unconstrained generation, their performance drops when forced to navigate explicit architectural rules. For end-users, this dichotomy implies that agents are reliable for rapid prototyping but remain unreliable for production-grade backend development. Overcoming this bottleneck requires a shift for agent developers:

moving beyond purely functional benchmarks to actively integrate structural awareness, potentially through retrieval-augmented framework documentation, constraint-oriented planning, or targeted pre-training on convention-heavy codebases.

## References

- Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, Zeyao Ma, Kashun Shum, Xuwu Wang, Jinxi Wei, Jiayi Yang, Jiajun Zhang, Lei Zhang, Zongmeng Zhang, Wenting Zhao, and Fan Zhou. Qwen3-coder-next technical report, 2026. URL <https://arxiv.org/abs/2603.00729>.
- Haorui Chen, Chengze Li, and Jia Li. Featbench: Evaluating coding agents on feature implementation for vibe coding. *arXiv preprint arXiv:2509.22237*, 2025.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.
- Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, et al. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*, 2025.
- Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36:46701–46723, 2023.
- Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=VtmBAGCN7o>.
- Naman Jain, Jaskirat Singh, Manish Shetty, Tianjun Zhang, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environment generation and hybrid verifiers for scaling open-weights SWE agents. In *Second Conference on Language Modeling*, 2025. URL <https://openreview.net/forum?id=7evvwdo3z>.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=VTF8yNQM66>.
- Kimi. Kimi k2.5: Visual agentic intelligence, 2026. URL <https://www.kimi.com/blog/kimi-k2-5>.
- Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. FEA-bench: A benchmark for evaluating repository-level code generation for feature implementation. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 17160–17176, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.839. URL <https://aclanthology.org/2025.acl-long.839/>.
- Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*, 2024.

- Jane Luo, Xin Zhang, Steven Liu, Jie Wu, Yiming Huang, Yangyu Huang, Chengyu Yin, Ying Xin, Jianfeng Liu, Yuefeng Zhan, Hao Sun, Qi Chen, Scarlett Li, and Mao Yang. RPG: A repository planning graph for unified and scalable codebase generation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL <https://openreview.net/forum?id=VAQq3Y8tIF>.
- Robert C. Martin. *Clean Architecture: A Craftsman’s Guide to Software Structure and Design*. Prentice Hall Press, USA, 1st edition, 2017. ISBN 0134494164.
- MiniMax. Minimax m2.5: Built for real-world productivity., 2026. URL <https://www.minimax.io/news/minimax-m25>.
- Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. Swe-lancer: Can frontier llms earn \$1 million from real-world freelance software engineering? In *International Conference on Machine Learning*, pp. 44412–44450. PMLR, 2025.
- Dan North et al. Introducing bdd. *Better Software*, 12(7), 2006.
- OpenAI. Introducing GPT-5.2, 2025. URL <https://openai.com/index/introducing-gpt-5-2/>.
- OpenAI. Introducing GPT-5.4, 2026. URL <https://openai.com/index/introducing-gpt-5-4/>.
- Qwen Team. Qwen3.5: Accelerating productivity with native multimodal agents, February 2026. URL <https://qwen.ai/blog?id=qwen3.5>.
- Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, et al. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents. *arXiv preprint arXiv:2504.08703*, 2025.
- Abhinav Rastogi, Adam Yang, Albert Q. Jiang, Alexander H. Liu, Alexandre Sablayrolles, Amélie Héliou, Amélie Martin, Anmol Agarwal, Andy Ehrenberg, Andy Lo, Antoine Roux, Arthur Darcet, Arthur Mensch, Baptiste Bout, Baptiste Rozière, Baudouin De Moncault, Chris Bamford, Christian Wallenwein, Christophe Renaudin, Clémence Lanfranchi, Clément Denoix, Corentin Barreau, Darius Dabert Devon Mizelle, Diego de las Casas, Elliot Chane-Sane, Emilien Fugier, Emma Bou Hanna, Gabrielle Berrada, Gauthier Delerice, Gauthier Guinet, Georgii Novikov, Graham Neubig, Guillaume Lample, Guillaume Martin, Himanshu Jaju, Jan Ludziejewski, Jason Rute, Jean-Malo Delignon, JeanHadrien Chabran, Joachim Studnia, Joep Barmentlo, Jonas Amar, Josselin Somerville Roberts, Julien Denize, Karan Saxena, Karmesh Yadav, Kartik Khandelwal, Khyathi Raghavi Chandu, Kush Jain, Léo Renard Lavaud, Léonard Blier, Lingxiao Zhao, Louis Martin, Lucile Saulnier, Luyu Gao, Marie Pellat, Mathilde Guillaumin, Mathis Felardos, Matthieu Dinot, Maxime Darrin, Maximilian Augustin, Mickaël Seznec, Neha Gupta, Nikhil Raghuraman, Olivier Duchenne, Patricia Wang, Patrick von Platen, Patryk Saffer, Paul Jacob, Paul Wambergue, Paula Kurylowicz, Philomène Chagniot, Pierre Stock, Pravesh Agrawal, Rémi Delacourt, Roman Soletskyi, Romain Sauvestre, Sagar Vaze, Sanchit Gandhi, Sandeep Subramanian, Shashwat Dalal, Siddharth Gandhi, Soham Ghosh, Srijan Mishra, Sumukh Aithal, Szymon Antoniak, Teven Le Scao, Thibaut Lavril, Thibault Schueller, Thomas Foubert, Thomas Robert, Thomas Wang, Timothée Lacroix, Tom Bewley, Valeriia Nemychnikova, Victor Paltz, Virgile Richard, Wen-Ding Li, William Marshall, Xingyao Wang, Xuanyu Zhang, Yihan Wan, and Yunhao Tang. Devstral: Fine-tuning language models for coding agent applications, 2025. URL <https://arxiv.org/abs/2509.25193>.
- Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. Paper2code: Automating code generation from scientific papers in machine learning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL <https://openreview.net/forum?id=3DcaUTjdKc>.
- SWE-agent Team. mini-swe-agent: The 100 line ai agent that’s actually useful, 2025. URL <https://github.com/SWE-agent/mini-swe-agent>.

Mark Vero, Niels Mündler, Victor Chibotaru, Veselin Raychev, Maximilian Baader, Nikola Jovanović, Jingxuan He, and Martin Vechev. Baxbench: Can llms generate correct and secure backends? In *ICLR 2025 Third Workshop on Deep Learning for Code*, 2024.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=0Jd3ayDDoF>.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Yiqing Xie, Alex Xie, Divyanshu Sheth, Pengfei Liu, Daniel Fried, and Carolyn Rose. RepoST: Scalable repository-level coding environment construction with sandbox testing. In *Second Conference on Language Modeling*, 2025. URL <https://openreview.net/forum?id=2txrMBpw3q>.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Aoyan Li, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, RUI LONG, Ming Ding, and liang xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025. URL <https://openreview.net/forum?id=MhBZzkz4h9>.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, 2023.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch. In *The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=MMwaQEVsAg>.

## APPENDIX

### A Subset Representativeness

Due to cost constraints, three models (MiniMax-M2.5, Kimi-K2.5, GPT-5.2) were evaluated only on a 16-task subset comprising 4 frameworks (aiohttp, Express, FastAPI, Fastify)  $\times$  4

constraint levels (L0-L3), where each framework follows the pipeline: unconstrained → +clean\_architecture → +postgres → +postgres+ORM.

To verify that results on this subset are representative of the full 80-task benchmark, we re-evaluated the six full-set configurations (GPT-5-mini, Qwen3-Coder-Next, Qwen3-235B-A22B, each with both Mini-SWE-Agent and OpenHands; Devstral-Small excluded due to near-zero scores) on the same 16 tasks and compared against their full-set scores. This yields  $N = 24$  paired observations (6 configurations × 4 constraint levels), where each pair consists of (full-set A%, subset A%) for the same configuration and level.

**Assert%** scores on the full set and subset correlate at Pearson  $r = 0.976$  ( $p < 10^{-15}$ ) and Spearman  $\rho = 0.948$  ( $p < 10^{-12}$ ). Critically, cross-model rankings are preserved almost perfectly at every constraint level. The L0→L3 decay direction is also consistent across all configurations: every configuration that shows performance degradation on the full set also degrades on the subset.

These results confirm that the 16-task subset reliably preserves both absolute performance rankings and constraint-decay trends, supporting the validity of subset-only evaluations for the cost-constrained models reported in Table 6.

Agent	Model	Full-set A%				Subset A%				$\Delta$ (pp)			
		L0	L1	L2	L3	L0	L1	L2	L3	L0	L1	L2	L3
Mini-SWE	GPT-5-mini	51.7	46.8	27.1	23.7	67.6	48.9	27.5	27.2	-15.9	-2.1	-0.4	-3.5
OpenHands	GPT-5-mini	65.8	63.2	56.7	52.2	81.6	60.6	53.7	47.1	-15.8	+2.6	+2.9	+5.1
Mini-SWE	Qwen3-Coder-Next	86.4	66.4	52.6	46.1	87.7	66.4	54.2	52.1	-1.4	0.0	-1.6	-6.0
OpenHands	Qwen3-Coder-Next	73.0	51.7	42.7	27.6	76.7	57.6	38.3	16.6	-3.6	-5.9	+4.3	+11.0
Mini-SWE	Qwen3-235B	29.6	10.7	9.1	2.3	25.9	11.3	5.3	1.4	+3.7	-0.6	+3.7	+0.8
OpenHands	Qwen3-235B	26.2	17.7	3.1	0.8	32.1	21.6	2.0	0.7	-5.9	-4.0	+1.1	0.0
MAE										7.7	2.5	2.3	4.4

Table 6: Subset representativeness: A% on the full 80-task set vs. the 16-task subset for each of the six validation configurations (Devstral-Small excluded).  $\Delta$  = full-subset. MAE = mean absolute difference. Pearson  $r = 0.976$ , Spearman  $\rho = 0.948$  ( $N = 24$ ,  $p < 10^{-12}$ ).

## B LLM-as-Judge Validation

To assess the reliability of the GPT-5.2 judge used for logic-error sub-categorization (§5.3), we drew a stratified random sample of 50 logic errors from the Qwen3-Coder-Next failure set ( $n=137$ ), preserving population proportions across the six subcategories. A human (one of the authors) independently labeled each entry using the same taxonomy and evidence (last 20 trajectory turns, test-failure summary, server logs), without viewing the judge’s label.

Table 7 reports the results. Overall accuracy is 98% (49/50) with Cohen’s  $\kappa = 0.975$ . Four of the six categories achieve perfect precision, recall, and F1. The near-perfect agreement is largely attributable to the rich signal in server logs and test outputs, which make the root cause of most logic errors unambiguous.

## C Verifier Functions

The three verifier functions introduced in §3.3 operate on the unified diff (patch) produced by each agent run. They parse only the added lines of the patch (i.e., the source code the agent actually wrote) and apply static pattern matching to determine structural compliance. No execution or dynamic analysis is performed.

Subcategory	Prec.	Rec.	F1	Support
auth_misconfiguration	100.0	100.0	100.0	11
incorrect_query_logic	92.3	100.0	96.0	12
state_propagation_failure	100.0	100.0	100.0	5
database_runtime_error	100.0	100.0	100.0	11
framework_idiosyncrasy	100.0	80.0	88.9	5
business_logic_defect	100.0	100.0	100.0	6
<b>Macro avg</b>	98.7	96.7	97.5	50

Table 7: Per-category precision, recall, and F1 (%) of the GPT-5.2 judge against human annotations on the 50-sample validation set. Cohen’s  $\kappa = 0.975$ ; overall accuracy = 98%.

### C.1 Architecture Verifier

This verifier enforces the Clean Architecture constraint through two checks:

**Layer presence.** The prompt prescribes four layers, each in its own directory: routes/handlers (rank 3), services/use cases (rank 2), repositories/data access (rank 1), and models/entities (rank 0). Agents use diverse naming conventions, so the verifier maps each directory component against a set of aliases (e.g., routes, handlers, routers, views all map to the routes layer). A solution is considered layered if at least 3 of the 4 canonical layers are present as distinct directories.

**Dependency direction.** For every source file classified into a layer, the verifier extracts all import statements (Python `import/from . . . import` and JavaScript `require()/import . . . from`) and resolves each import path to a target layer using the same alias mapping. A dependency violation is recorded when a file in a lower-rank layer imports from a higher-rank layer (e.g., a repository importing from a route handler).

A solution is compliant if and only if both checks pass: sufficient layer presence and no upward dependency violations.

### C.2 Database Verifier

This verifier checks that the agent used the prescribed database engine (SQLite or PostgreSQL) and did not use any alternative. It scans all added source lines (excluding lock files, which list optional peer dependencies for all supported dialects) for two sets of regex evidence patterns:

**SQLite evidence.** `sqlite3/aiosqlite imports`, `sqlite:/// connection strings`, Django’s `sqlite3 backend`, Node packages (`sqlite3`, `better-sqlite3`), and Sequelize dialect: `'sqlite'`.

**PostgreSQL evidence.** `psycopg2/asyncpg imports`, `postgresql:// connection strings`, SQLAlchemy’s `PostgreSQL dialect imports`, Django’s `postgresql backend`, Node packages (`pg`, `pg-promise`), Sequelize dialect: `'postgres'`, and host references to the Docker service name `postgres`.

A solution is compliant when at least one evidence pattern for the expected database is found and no evidence of the alternative database is present. A special case handles Django projects that retain the default SQLite backend in `settings.py` even when PostgreSQL is the active configuration (the second `DATABASES` assignment overrides the first).

### C.3 ORM Verifier

This verifier checks for evidence of the prescribed ORM (SQLAlchemy for Python tasks, Sequelize for Node tasks). It scans added lines for:

**SQLAlchemy.** `import sqlalchemy / from sqlalchemy import`, or any `sqlalchemy` mention in dependency files.

**Sequelize.** `require('sequelize') / import ... from 'sequelize', new Sequelize()` constructor calls, or "sequelize": entries in `package.json`.

A solution is compliant if at least one evidence pattern for the expected ORM is found. Unlike the database verifier, the presence of an alternative ORM is not penalized: some agents import utility libraries that happen to reference other ORMs without actually using them for data access.

#### C.4 Limitations

All three verifiers are deliberately conservative in their design:

**Static-only.** The verifiers analyze source text, not runtime behavior. An agent could satisfy the verifier’s patterns while violating constraints at runtime (e.g., importing SQLAlchemy but executing raw SQL). This means false positives are possible, but as discussed in Appendix D, they can only weaken the observed decay signal.

**Pattern-based.** Regex matching cannot capture all possible naming conventions or import patterns. We mitigate this with broad alias sets and case-insensitive matching, and validate empirically that the false-reject rate is  $\leq 0.7\%$  (§5.1).

### D Verifier Impact on Reported Metrics

A potential concern with our evaluation is that the static verifier functions (architecture, database, ORM) could introduce measurement artifacts: false negatives (incorrectly rejecting compliant runs, zeroing their A% contribution) would artificially inflate the constraint-decay signal, while false positives (incorrectly accepting non-compliant runs) would weaken it by allowing structurally incorrect implementations to retain their A% scores.

To quantify the actual impact, we compute two variants of A%: (i) with verifier enforcement (Table 2), where a run that violates any applicable verifier has its assertion score set to zero before averaging; and (ii) without verifier enforcement, using raw behavioral-test scores regardless of structural compliance. Table 8 reports both variants side-by-side.

**False negatives.** The maximum difference between the two A% variants across all configurations and constraint levels is 2.7 pp (OpenHands + Qwen3-Coder-Next at L3). The average L0→L3 decay changes from 28 pp (without enforcement) to 29 pp (with enforcement). This near-identity confirms that verifier false negatives have negligible impact: the overwhelming majority of test-passing runs also satisfy all structural constraints, and the few that do not ( $\leq 0.7\%$  of constrained test-passing runs) barely move the aggregate statistics.

**False positives.** If verifiers were to miss genuine constraint violations (false positives), the effect would be to preserve the A% of non-compliant runs. This can only weaken the observed decay signal, making our reported constraint decay a conservative lower bound on the true effect. In other words, false positives cannot inflate the decay, they can only mask it.

Together, these observations confirm that the constraint-decay findings are robust to verifier imperfections in both directions.

### E Generation Task Design

Each generation task instructs an LLM agent to produce, from scratch, a fully functional REST API server that conforms to the RealWorld (Conduit) OpenAPI specification. Tasks are assembled from a modular prompt template whose blocks are conditionally included depending on the target constraint level.

#### E.1 Prompt Template

The master template has the following structure (optional blocks are marked with  $\langle \cdot \rangle$ ):

Agent	Model	L0			L1			L2			L3		
		A%	Raw	$\Delta$	A%	Raw	$\Delta$	A%	Raw	$\Delta$	A%	Raw	$\Delta$
Mini-SWE	GPT-5-mini	51.7	51.7	0.0	46.8	46.8	0.0	27.1	27.7	0.6	23.7	25.1	1.4
OpenHands	GPT-5-mini	65.8	65.8	0.0	63.2	63.2	0.0	56.7	57.7	1.0	52.2	54.7	2.5
Mini-SWE	Qwen3-Coder-Next	86.4	86.4	0.0	66.4	66.4	0.0	52.6	54.7	2.1	46.1	48.8	2.7
OpenHands	Qwen3-Coder-Next	73.0	73.0	0.0	51.7	51.8	0.1	42.7	43.9	1.2	27.6	30.0	2.4
Mini-SWE	Qwen3-235B	29.6	29.6	0.0	10.7	10.7	0.0	9.1	9.1	0.0	2.3	2.9	0.7
OpenHands	Qwen3-235B	26.2	26.2	0.0	17.7	17.7	0.0	3.1	3.1	0.0	0.8	0.8	0.0
Mini-SWE	Devstral-Small	6.8	6.8	0.0	4.7	4.7	0.0	2.3	3.6	1.3	2.9	3.0	0.1
Mini-SWE	MiniMax-M2.5*	88.6	88.6	0.0	92.5	92.5	0.0	66.8	66.8	0.0	58.3	58.3	0.0
OpenHands	MiniMax-M2.5*	95.6	95.6	0.0	97.0	97.0	0.0	87.3	87.3	0.0	78.6	78.6	0.0
Mini-SWE	Kimi-K2.5*	85.4	85.4	0.0	70.9	72.5	1.6	62.9	62.9	0.0	53.7	53.7	0.0
Mini-SWE	GPT-5.2*	78.2	78.2	0.0	49.3	49.3	0.0	27.1	27.1	0.0	48.0	48.0	0.0

Table 8: A% with and without verifier enforcement. “A%” zeros out assertion scores for runs that violate structural constraints; “Raw A%” uses behavioral-test scores only.  $\Delta = \text{Raw A\%} - \text{A\%}$ . Full-set models use all 80 tasks; \* = 16-task subset.

```

{OpenAPI specification}
Generate a complete {runtime} REST API server compliant with the given
specification.
## Requirements
-- You MUST use the {framework} framework.
-- You MUST work in the current directory.
<-- You MUST follow the architecture described below.>
<-- You MUST use the database as described below.>
<-- You MUST use the {ORM} ORM for handling the database.>
## Architecture
(Clean Architecture pattern: four layers, dependency rule, directory layout)
## Database
(SQLite auto-schema or PostgreSQL with connection credentials)
## Mandatory Files
(requirements.txt/package.json + run.sh)
## Server Configuration
-- The server MUST listen on port {port}.
-- The server MUST expose GET /api/health-check returning 200.
-- All routes MUST be prefixed with /api.
## Evaluation Pipeline
(dependency install → run.sh → health-check poll → Postman test suite)

```

The {runtime} placeholder is filled with Python 3.12 or Node.js; {framework} with one of the eight target frameworks; {port} with 8080 for Python or 3000 for Node; and {ORM} with SQLAlchemy or Sequelize. The OpenAPI specification at the top is a standard OpenAPI 3.0 YAML document (~500 lines) defining all Conduit endpoints, schemas, and authentication requirements.

## E.2 Constraint Blocks

The optional blocks correspond to the three constraint dimensions introduced in §3.2:

**Architecture.** A paragraph prescribing the Clean Architecture pattern: four layers (routes/handlers, services/use cases, models/entities, repository/data access) with a strict top-down dependency rule (each layer may only import from the layer below) and a directory-per-layer layout.

**Database.** Either a SQLite block (auto-create schema on startup) or a PostgreSQL block (auto-create schema on startup, with given fixed credentials: user/password@postgres:5432/conduit).

**ORM.** A single sentence requiring SQLAlchemy (Python) or Sequelize (Node.js).

Constraint levels L0–L3 are formed by including an increasing number of these blocks: L0 includes none (baseline); L1 includes one (e.g., architecture only, or database only); L2 includes two (e.g., architecture + database); L3 includes all three. For each of the 8 frameworks we generate all 10 valid constraint combinations, yielding 80 distinct generation tasks.

### E.3 Prompt Example

Figure 2 shows an abridged prompt for a task with all constraints active.

```

openapi: "3.0.1"
info:
  title: "Conduit API"
  version: "1.0.0"
  paths:
    /api/users/login:
      post: ... (19 endpoints, ~500 lines total, omitted for brevity)

Generate a complete Python 3.12 REST API server compliant with the given specification.

## Requirements
- You MUST use the flask framework.
- You MUST work in the current directory.
- You MUST follow the architecture described below.
- You MUST use the database as described below.
- You MUST use the SQLAlchemy Python ORM for handling the database.

## Architecture
You MUST follow the Clean Architecture pattern. Organize your code into these layers:
- Routes/Handlers layer: HTTP request handling, input parsing, response formatting. No business logic.
- Services/Use Cases layer: Business logic and orchestration. Framework-agnostic.
- Models/Entities layer: Data structures and domain objects.
- Repository/Data Access layer: All data storage operations. Accessed only through the Services layer.
Each layer must only depend on the layer below it. Keep each layer in its own directory.

## Database
Use PostgreSQL as the database. The database schema MUST be created automatically on server startup.
An instance of Postgres 16 is already running on port 5432.
Connection: username=user, password=password, db=conduit, host=postgres.

## Mandatory Files
1. requirements.txt – all pip dependencies.
2. run.sh – a shell script that starts the server.

## Server Configuration
- The server MUST listen on port 8080.
- The server MUST expose a health-check endpoint at GET /api/health-check returning HTTP 200.
- All API routes MUST be prefixed with /api.

## Evaluation Pipeline
1. uv pip install --system -r requirements.txt
2. chmod +x run.sh
3. ./run.sh (starts your server in background)
4. Health-check polls GET http://localhost:8080/api/health-check until 200.
5. A Postman test suite runs against http://localhost:8080/api.

```

Figure 2: Generation prompt for an L3 task. The full OpenAPI specification (~500 lines) is prepended verbatim but omitted here for space.

## F Feature Implementation Task Design

Feature implementation tasks evaluate an agent’s ability to add missing functionality to an existing, partially stripped codebase. Unlike generation tasks, the agent starts with a working repository and must read, understand, and extend it: simulating the common developer scenario of implementing a new feature in an established project.

### F.1 Construction by Codebase Ablation

Each task is derived from a real open-source RealWorld (Conduit) implementation. Since the Conduit API is organized around well-defined feature groups (articles, comments, tags, profiles, users), we can cleanly remove an entire feature vertical (routes, controllers/handlers, services, data-access code, and model definitions) while leaving the rest of the codebase intact and functional. Concretely, for each repository we:

- (i) Select one or more feature domains to ablate (e.g., “comments,” or “articles + comments + tags”).
- (ii) Apply a `git diff` patch that removes all code belonging to those domains.
- (iii) Verify that the stripped repository still compiles, starts, and partially passes the test suite on the *remaining* endpoints.
- (iv) Record the resulting task: the agent must restore full API compliance by re-implementing the removed features.

Crucially, the number of files and lines affected by each task is not arbitrarily chosen: it is a direct consequence of how much code a given feature domain occupies across the repository’s layered architecture. A single-domain removal (e.g., comments) touches fewer files than a multi-domain removal (e.g., articles + comments + tags) because the latter spans more entities, more handlers, and more service logic. This mirrors the real-world situation in which the scope of a feature implementation is determined by the feature itself.

### F.2 Task Inventory and Statistics

We construct 20 tasks across five framework repositories (4 per framework), covering three ablation scopes summarized in Table 9.

Ablation scope	Tasks	Lines removed	Files
Single domain (comments)	5	176–639	4–15
Single domain (profiles)	5	161–407	2–14
Multi-domain (articles + tags ± comments)	10	726–2,604	5–46
<b>All 20 tasks</b>	<b>20</b>	<b>161–2,604</b>	<b>2–46</b>

Table 9: Feature implementation tasks grouped by ablation scope. The wide range of lines and files reflects natural variation in how each repository organizes code across layers.

Table 10 lists the five repositories and their provenance.

### F.3 Prompt and Evaluation

Each task is stored as a JSON file containing: (i) the agent prompt (the full OpenAPI specification followed by a single instruction sentence), (ii) a unified diff patch that strips the target feature before the agent starts, (iii) repository metadata (URL, pinned commit, setup and run commands), and (iv) task metadata (lines removed, files affected).

The evaluation pipeline clones the repository at the pinned commit, applies the ablation patch, runs the agent, then executes the same Postman test suite used for generation tasks. Because the ablation preserves all non-target features, the stripped repository already passes

Framework	Repository	Tasks
Express	realworld-express-prisma	4
Flask	realworld-flask	4
FastAPI	fastapi-realworld-backend	4
aiohhttp	realworld-aiohhttp	4
Hono.js	realworld-honojs	4

Table 10: Source repositories for feature implementation tasks. All implement the full RealWorld Conduit API with PostgreSQL, an ORM (Prisma or SQLAlchemy), and a layered architecture.

all tests for the retained endpoints; the agent’s task is to bring the remaining (currently failing) tests back to passing.

Figure 3 shows the abridged prompt for an Express task.

```

openapi: "3.0.1"
info:
  title: "Conduit API"
  version: "1.0.0"
paths:
  /api/users/login:
    post: ... (19 endpoints, ~500 lines total, omitted for brevity)

Complete the existing repository and make it compliant to this OpenAPI specification by adding
missing comments functionalities

```

Figure 3: Example feature implementation prompt. The agent receives the full OpenAPI specification followed by a single instruction. The repository has been pre-stripped of 245 lines across 10 files (the entire comments vertical: route registrations, controller, Prisma schema entries, and validation middleware).

## G Behavioral Test Suite

All task variants share a single specification compliant HTTP behavioral test suite, defined as a Postman<sup>3</sup> API testing framework collection and executed via the Newman<sup>4</sup> engine. The collection encodes a stateful integration-test scenario: requests run in a fixed order and later requests depend on state created by earlier ones (e.g., the authentication token, article slug, and comment identifier produced by prior steps).

### G.1 Coverage

The suite contains 32 requests targeting 19 unique API endpoints (the full spec. surface) and 291 assertions. Requests are organized into five sequential folders described in Table 11.

### G.2 Assertion Taxonomy

Each assertion belongs to one of four categories:

**Property presence.** Verifies that the JSON response contains expected keys (e.g., `user.email`, `article.slug`).

**Type validation.** Checks that the JSON response attribute values have the correct type (e.g., `articles` is an array, `favoritesCount` is an integer).

<sup>3</sup><https://www.postman.com/>

<sup>4</sup><https://github.com/postmanlabs/newman>

Folder	Req.	Assert.	Scope
Auth	5	30	Register, login, token, profile update
Articles	4	20	Public listing (by author, tag, favorites)
Articles, Favorites, Comments	18	212	Full CRUD + favorites + comments lifecycle
Profiles	4	26	Follow / unfollow social graph
Tags	1	3	Tag listing
<b>Total</b>	<b>32</b>	<b>291</b>	

Table 11: Test suite folder breakdown. The third folder accounts for 73% of all assertions because it exercises the full article lifecycle including nested resources.

**Status-code checks.** Verifies the HTTP response code (e.g., 200 OK, 201 Created).

**State-transition checks.** Verifies that server-side state mutated correctly (e.g., favorited flips from false to true after the relative POST request).

## H Token Consumption

Table 12 reports input and output token consumption aggregated across all runs of the full evaluation, broken down by constraint level. Table 13 provides the same breakdown per agent-model pair.

Level	Avg Input	Avg Output	Total Input	Total Output
L0	1,012,477	12,925	229,832,290	2,933,917
L1	2,014,935	20,660	1,172,691,886	12,024,383
L2	2,663,374	23,344	2,024,163,960	17,741,580
L3	3,097,662	26,221	1,260,748,542	10,671,896
Total	-	-	4,687,436,678	43,371,776

Table 12: Global token consumption by constraint level, aggregated across all agent-model configurations.

Agent	Model	Avg Input Tokens				Avg Output Tokens			
		L0	L1	L2	L3	L0	L1	L2	L3
Mini-SWE	GPT-5-mini	19K	50K	69K	124K	4K	9K	10K	12K
Mini-SWE	GPT-5.2*	60K	163K	310K	237K	7K	22K	28K	23K
Mini-SWE	Qwen3-Coder-Next	1,676K	3,512K	5,138K	7,153K	19K	44K	44K	53K
Mini-SWE	Qwen3-235B	107K	167K	324K	415K	6K	10K	12K	13K
Mini-SWE	MiniMax-M2.5*	262K	1,121K	1,405K	1,261K	8K	19K	25K	21K
Mini-SWE	Kimi-K2.5*	150K	829K	1,144K	1,115K	6K	16K	18K	18K
OpenHands	GPT-5-mini	304K	519K	908K	1,732K	21K	24K	27K	35K
OpenHands	Qwen3-Coder-Next	3,774K	6,560K	7,310K	7,115K	23K	31K	33K	32K
OpenHands	Qwen3-235B	702K	1,267K	1,373K	1,474K	11K	18K	20K	20K
OpenHands	MiniMax-M2.5*	1,132K	1,923K	2,552K	2,914K	10K	18K	23K	24K

Table 13: Average input and output tokens per run by constraint level and agent-model pair (K =  $\times 10^3$ , rounded). \* = 16-task subset.

## I Execution Pipeline

This section details the end-to-end infrastructure that executes each task. The pipeline is fully containerized and comprises two sequential phases: (i) build (agent execution) and (ii) evaluate (behavioral testing), each running in a pristine Docker environment to prevent state leakage between runs.

## I.1 Infrastructure Overview

Each framework (e.g., Flask, Express) has a dedicated Docker Compose project defining three services on a shared network:

**conduit.** The application container. Built from a multi-stage Dockerfile that (i) installs two Python virtual environments for agent execution (one for Mini-SWE-Agent, one for OpenHands), and (ii) clones the reference repository at a pinned commit with all framework dependencies pre-installed. For generation tasks, the repository starts empty (only a `gitkeep` placeholder); for feature implementation tasks, it contains the full codebase.

**postgres.** A PostgreSQL 16 instance with fixed credentials (`user/password@postgres:5432/conduit`).

**newman.** A Node.js container with the Newman suite executor pre-installed, used exclusively during the evaluation phase.

## I.2 Build Phase (Agent Execution)

The build phase proceeds as follows:

- (i) **Image build.** `docker compose build conduit` compiles the framework-specific image.
- (ii) **Container launch.** The conduit container is started in detached mode with two bind-mounted volumes: `/agents` (agent scripts and model cost registry) and `/trajectories` (output directory for the agent’s conversation trace).
- (iii) **Task preparation.** The task JSON is copied into the container. For feature implementation tasks, the ablation patch is applied via `git apply` to strip the target feature. The Git history is then re-initialized and the baseline commit SHA is recorded.
- (iv) **Setup commands.** Any task-specific setup commands run inside the container (e.g., `npm run migrate:develop` for Express/Prisma tasks, or `alembic upgrade head` for SQLAlchemy tasks).
- (v) **Agent invocation.** The agent script is executed with environment variables specifying the LLM model, API key, task prompt, and a system guidance message containing server startup instructions. For example, Mini-SWE-Agent runs as:  

```
/mini-env/bin/python /agents/mini_swe_sdk.py
```

The agent can read, write, and execute code within the `/repository` folder.
- (vi) **Patch extraction.** After the agent finishes, a `git diff` against the baseline SHA captures all changes as a unified diff patch, excluding generated artifacts (`node_modules`, `__pycache__`, `*.db`, lock files). This patch is the sole artifact carried to the evaluation phase.
- (vii) **Cleanup.** The container is stopped and removed; all volumes are destroyed.

To avoid potential agentic loops, a maximum iteration limit is set for both agents: 300 turns for Mini-SWE-Agent and 200 turns for OpenHands.

## I.3 Evaluate Phase (Behavioral Testing)

The evaluation phase applies the agent’s patch to a *pristine* container, ensuring that any side effects from the build phase (e.g., leftover processes, corrupted state) do not influence the result.

- (i) **Pristine container.** A new conduit container is started from the same image. For feature implementation tasks, the ablation patch is applied first (reproducing the stripped baseline), followed by the agent’s patch.
- (ii) **Setup and server start.** Setup commands run again in the clean environment. The server is started in the background via `run.sh`, with `stdout/stderr` redirected to a log file.

- (iii) **Health-check polling.** The pipeline polls the health-check endpoint (GET `http://conduit:{port}/api/health-check`) up to 5 times at 5-second intervals (120-second total timeout). If the server never responds, the evaluation continues to record a baseline measurement (typically 0% assertions passed).
- (iv) **Newman test execution.** The Postman collection is executed via Newman against the running server:

```
newman run Conduit.postman_collection.json \  
--global-var APIURL=http://conduit:{port}/api \  
--reporters cli,html,csv
```
- (v) **Artifact collection.** Server logs, Newman reports (HTML + CSV), and agent patch are saved to the results directory.
- (vi) **Verifier execution.** The three static verifiers (§C) are run on the agent patch to assess structural compliance.

#### I.4 Walkthrough: Generation Task

Consider the L3 task `flask-openapi-clean_architecture-postgres-sqlalchemy`.

- (i) The conduit container starts with an empty `/repository` directory (no pre-existing code).
- (ii) The agent receives the full prompt (Appendix E, Figure 2) and generates a Flask application from scratch: `models`, `routes`, `services`, `repositories`, `configuration`, `requirements.txt`, and `run.sh`.
- (iii) The build phase extracts a patch containing all created files (~500–2000 added lines).
- (iv) The evaluate phase applies this patch to a fresh empty container, runs `uv pip install -r requirements.txt`, starts the server on port 8080, and executes the Postman suite.
- (v) Verifiers check: (a)  $\geq 3$  architectural layers with no upward imports, (b) PostgreSQL evidence present and no SQLite evidence, (c) SQLAlchemy evidence present.

#### I.5 Walkthrough: Feature Implementation Task

Consider the Express task that ablates the comments feature (245 lines across 10 files).

- (i) The conduit container starts with the full Express/Prisma repository at the pinned commit. The ablation patch is applied, removing all comment-related code (routes, controller, Prisma schema fields, validation middleware). After ablation, the server still starts and passes all non-comment tests.
- (ii) The agent receives the prompt (Appendix F, Figure 3) and must re-implement the missing comments functionality by reading the existing codebase and extending it.
- (iii) The build phase extracts a patch containing only the agent’s additions.
- (iv) The evaluate phase starts from the pinned commit, applies the ablation patch, then applies the agent’s patch on top, sets up the server execution (e.g., runs database migrations), starts the server on port 3000, and executes the Postman suite.

In this case no verifiers are applied since the constraints are already implicitly stated in the codebase.

## J Revised Agent Scaffolds

Both agent scaffolds required targeted modifications to operate in our greenfield generation setting and to ensure fully automated evaluation.

**Mini-SWE-Agent.** Mini-SWE-Agent is designed for issue-solving on existing repositories. To repurpose it for greenfield generation, we replaced the default “fix the issue” instance prompt with a generative workflow. The injected instructions direct the agent through four steps:

1. Analyze the codebase by finding and reading relevant files
2. Edit/Create the source code to solve your task
3. Test edge cases to ensure your solution is robust
4. Submit your changes and finish your work by issuing:  
echo COMPLETE\_TASK\_AND\_SUBMIT\_FINAL\_OUTPUT

No other changes were made to the scaffold: the system prompt, action format, observation template, and execution environment remain identical to the upstream configuration.

**OpenHands.** OpenHands provides a built-in finish tool that the agent is expected to call when work is complete. In practice, however, several models prematurely halt execution by emitting a natural-language summary message instead of invoking the finish tool. This triggers the FINISHED status without the agent having completed the task, as OpenHands interprets the end of a `conversation.run()` cycle with a final agent message as a request for user input.

To mitigate this, we wrap the conversation loop with a re-prompting mechanism (up to 6 turns). After each `conversation.run()` call, we inspect the last event: if it is an agent message rather than a tool invocation, we send a follow-up prompt instructing the agent to either continue working or call the finish tool explicitly. The loop terminates only when the agent invokes `finish` or the maximum number of re-prompting turns is exhausted. This ensures that models prone to premature halting (Ding et al., 2025) are given the opportunity to resume work, yielding complete evaluation trajectories without manual intervention.