LoLBench: Evaluating Coding Agents with Long-Horizon Proposals on Large Software Systems

Anonymous authors
Paper under double-blind review

Abstract

Modern coding agents can deliver increasingly large repository-level changes, and recent benchmarks reflect this by emphasizing long-horizon tasks with large reference implementations. Many benchmarks evaluate coding agents' implementation capability to produce correct code edits from detailed specifications. However, practical modular development tasks also require the perception capability of grounding user intent and high-level design to derive a specification. We introduce LoLBench to evaluate both capabilities through the entire proposal-to-implementation process on large software systems. It is a multilingual benchmark of 100 tasks across 29 software systems in five domains. Each task provides a human-written enhancement proposal with user intent and high-level design. On average, proposals contain about 5,000 words, software systems contain 2.4 million source lines of code (LoC), and implementation pull requests (PRs) change approximately 5,500 LoC. Across 28 agents we evaluated, the best agent resolves only 14% of tasks and achieves a 52.7% Fail-to-Pass (F2P) pass rate. Failure analysis identifies incomplete code localization as a major bottleneck, while providing reference-derived file trees alongside API specifications improves resolved rates by 16–22 percentage points (2.4–17×), reaching at most 34%. These results show that both perception and implementation remain central challenges for coding agents in practical modular development on large software systems.

1 Introduction

Large language models (LLMs) and coding agents have advanced rapidly in software development. Equipped with tools for planning, exploration, editing, and testing, modern agents can resolve increasingly complex issues and implement large features. Correspondingly, recent benchmarks have expanded evaluation scale through longer specifications and reference solutions (Jimenez et al., 2024; Deng et al., 2025; Huang et al., 2026; Li et al., 2025; Zhou et al., 2026). These benchmarks reveal important limitations in agents' ability to carry out large implementations.

However, software development in practice does not begin with detailed specifications. Instead, users and maintainers often begin by expressing the intent and high-level design. Developers must then ground them in existing software systems and decide implementation plans. Many coding benchmarks reduce this uncertainty by providing detailed specifications that contain target interfaces and implementation guidance, or bug reports with local problem scopes. They therefore emphasize a critical but comparatively later stage of software development: producing correct code edits based on clarified specifications or bug reports. This raises a complementary question: *Can coding agents deliver correct implementations based directly on user intent and high-level design?*

This intent-to-implementation process is particularly visible in modular development on large software systems. Projects such as CPython, OpenJDK, Apache Kafka, and Kubernetes use enhancement proposals (EPs) to discuss and govern substantial changes. These proposals express the intent of users and maintainers, including desired functionalities, constraints, and design rationale, together with high-level design decisions. Developers must determine how the proposed design fits into the existing architecture, which modules and interfaces must change, and how to integrate and validate the changes. For example, Python Enhancement Proposal 617 (van Rossum et al., 2020) proposes

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

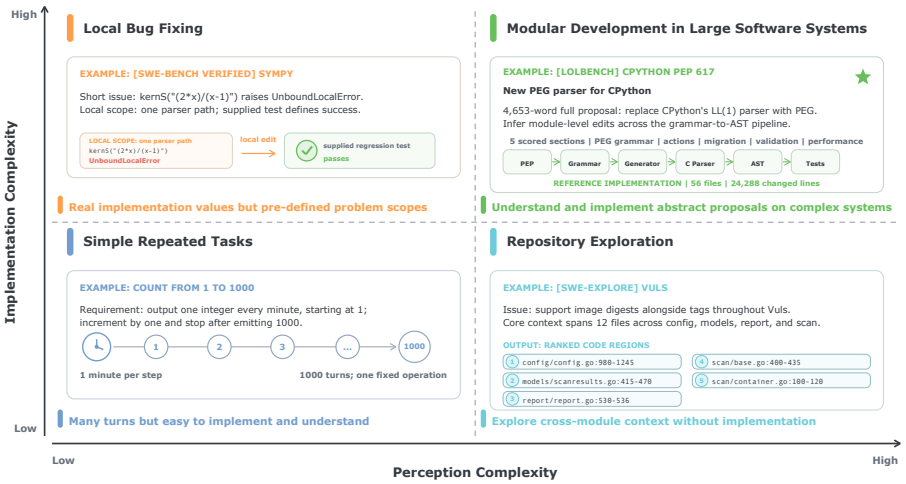


Figure 1: Illustration of tasks with different levels of perception and implementation complexity. Perception complexity captures the difficulty of grounding user intent and high-level design in an existing software system, whereas implementation complexity captures the difficulty of producing correct and regression-free code changes from that understanding.

replacing CPython's LL(1)-based parser with a PEG-based parser. Realizing this proposal nevertheless requires developers to determine how the new parser should integrate with CPython's existing components without breaking other functionalities.

Modular development involves two sources of complexity: 1) **perception complexity**, which captures the difficulty of grounding user intent and high-level design in a large software system to derive a detailed implementation specification. This requires interpreting the proposal, understanding cross-module dependencies, and determining the affected code. 2) **implementation complexity**, which captures the difficulty of translating that specification into functionally correct and regression-free code changes on an existing large software system instead of from scratch.

Fig. 1 illustrates how these two complexities distinguish different tasks. Simple repeated tasks have low perception and implementation complexity. Although they may require many steps, each step is easy to understand and execute. Repository-exploration tasks have substantial perception complexity with low implementation complexity. For example, SWE-Explore (Zhang et al., 2026) provides an agent with an issue and a codebase and asks it to identify and rank the relevant code regions under a fixed context budget. Conversely, a local bug-fixing task may require a technically difficult repair but have a clear problem scope, giving it high implementation complexity but low perception complexity. Modular development in large software systems occupies the most challenging region. An agent must first understand an abstract intent in a large software system, then carry that understanding through a large implementation while maintaining compatibility with the rest of the system. These tasks combine high perception complexity with high implementation complexity.

To evaluate current agents in these tasks, we introduce LOLBENCH, a multilingual benchmark containing 100 modular development tasks across 29 large software systems in five domains. Each task in LOLBENCH includes a human-written enhancement proposal from open-source communities. On average, a proposal contains approximately 5,000 words, the corresponding software system contains 2.4 million source LoC, and its implementation PR changes approximately 5,500 LoC. Agents must interpret the proposal, determine how its intended functionality fits into the existing system, and produce regression-free changes in an offline Docker environment. We evaluate implementations using system-level and end-to-end Fail-to-Pass (F2P) tests through public interfaces, allowing structural divergence from the reference implementation, along with Pass-to-Pass (P2P) regression tests. We further augment the F2P suites through proposal mutation and coverage-guided test generation, and map F2P tests to implementable sections of EPs to enable fine-grained evaluation of partial progress.

We evaluate six powerful LLMs on five scaffolds on LoLBench. The highest resolved rate is only 14%, achieved by Opus 5 on Claude Code and mini-SWE-agent. Claude Code with Opus 5 achieves the highest completed rate of 26.3% and an F2P pass rate of 52.7%, showing that even the strongest agent implements only a fraction of the proposal. Based on reference solutions from human developers, our failure analysis attributes the largest failures to incomplete code localization. When we additionally give three representative agents reference-derived file trees and API specifications, their resolved rates increase by 16–22 percentage points (2.4–17×), reaching at most 34%. These results highlight the importance of perception and implementation capability in modular development tasks on large software systems.

We summarize our key contributions as follows:

- We introduce modular development tasks on large software systems with two dimensions of complexity: perception complexity, which concerns grounding user intent and high-level design in an existing software system, and implementation complexity, which concerns turning that understanding into verified code changes.
- We construct LoLBench, a multilingual benchmark that consists of 100 modular development tasks from enhancement proposals containing user intent and high-level design, across 29 large software systems in five domains. LoLBench also includes augmented system-level tests to support flexible implementation and fine-grained evaluation.
- We evaluate six frontier LLMs on five popular scaffolds on LoLBench and demonstrate substantial challenges in both perception and implementation for current coding agents.

2 Related Work

2.1 Coding Benchmarks

Early benchmarks such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), xCodeEval (Khan et al., 2023), CrossCodeEval (Ding et al., 2023) and DevEval (Li et al., 2024) evaluate code generation from self-contained problems. NL2Repo-Bench (Ding et al., 2025) extends evaluation to repository generation, but does not require integration into an existing software system. SWE-bench (Jimenez et al., 2024) introduces real-world repository-level issue resolution, followed by SWE-bench Verified (OpenAI, 2024), Multi-SWE-bench (Zan et al., 2025), and the longer-horizon SWE-bench Pro (Deng et al., 2025). Recent benchmarks further increase implementation complexity through original engineering tasks (Huang et al., 2026), feature implementation (Li et al., 2025; Zhou et al., 2026), and terminal environments (Merrill et al., 2026). These tasks may require repository exploration, but primarily evaluate the correctness of the resulting implementation. Conversely, SWE-Explore (Zhang et al., 2026) isolates repository exploration by asking agents to rank relevant code regions without implementing the requested change. LoLBench complements these benchmarks by jointly evaluating perception and implementation. Its tasks use human-written enhancement proposals that predate their reference implementations. Agents must ground the expressed intent and high-level design in an existing large software system, identify necessary cross-module changes, and produce a functionally correct and regression-free implementation.

2.2 Coding Agents

ReAct (Yao et al., 2022) establishes the paradigm of interleaving reasoning with actions in an external environment. Software-engineering agents such as SWE-agent (Yang et al., 2024), AutoCodeRover (Zhang et al., 2024), live-swe-agent (Xia et al., 2025), and mini-SWE-agent (Lieret & Jimenez, 2025) combine repository exploration, editing, testing, and repair. General-purpose scaffolds include OpenHands (Wang et al., 2025), OpenCode (Anomaly, 2026), and Pi (Earendil Inc., 2026), alongside commercial systems such as Claude Code (Anthropic, 2026a) and Codex (OpenAI, 2026a). We evaluate representative models and scaffolds on the substantially broader context required by modular development in large software systems.

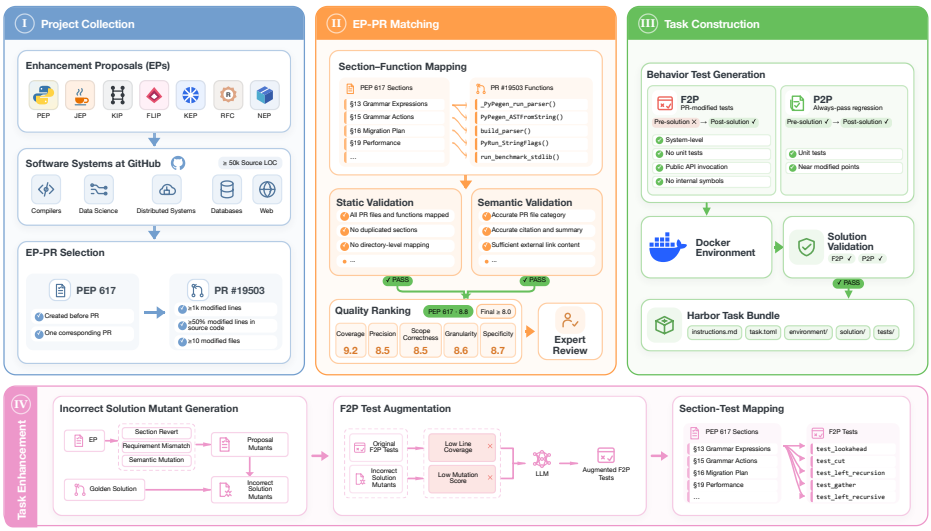


Figure 2: Construction of LoLBench. We collect enhancement proposals and their corresponding implementations, align proposal sections with implementation changes, construct executable tasks, and augment behavioral tests with incorrect solution mutants and coverage feedback.

3 Benchmark Construction

We implement a four-phase construction process to collect high-quality modular development tasks for LoLBench, as shown in Fig. 2.

3.1 Project Collection

We use *enhancement proposal* (EP) to denote a community-authored artifact that records intended functionality and high-level design before implementation, including formal proposals, RFCs, design documents, and feature requests. For each completed EP, we identify its software system and corresponding implementation PR. We retain an EP–PR pair when: 1) the software has at least 50k source LoC, the EP contains at least one implementable section, and the PR changes at least 1k LoC across at least 10 files, with source code accounting for over half of the changed lines; 2) the EP predates the PR and corresponds to exactly one merged PR, with no competing pending implementation; and 3) the PR contains tests that check the proposed behavior.

3.2 EP-PR Matching

We align each EP with its reference implementation to verify that their functional scopes match. We first classify proposal sections as *implementable* when they describe externally observable functionality realized by the PR, or *contextual* when they provide background, rationale or process information. We then map changed files, classes, and functions to the implementable sections they directly implement. We validate each mapping using 28 static rules for structural properties such as completeness, uniqueness, and mapping direction, and eight semantic rules for scope and attribution correctness. An LLM judge subsequently scores each candidate along five dimensions: requirement coverage, mapping precision, scope correctness, granularity, and requirement specificity. We retain candidates with a weighted score of at least 8.0. Appendix B provides details of this phase.

3.3 Task Construction

For each remaining candidate, we build a task for it based on the Harbor framework (Harbor Framework Team, 2026). Each task provides an agent with the EP and the codebase before implementation. The non-test portion of the corresponding PR forms the reference solution but is hidden from the agent. We evaluate the correctness of solutions based on F2P and P2P tests. An F2P test must

Table 1: Comparison with recent coding benchmarks. The last four columns report mean (median) values over tasks. "Repo Size" counts source lines excluding comments and blanks. "Solution Size" counts inserted and deleted lines across the implementation PR. "Modified" counts inserted, deleted, and changed files/classes/functions in solutions. "Class Mentioned" is the percentage of target class names that appear in the task description. Lower values indicate less explicit implementation guidance.

Benchmark	#Tasks	Lang.	Type	Repo Size (k LoC)	Solution Size (LoC)	Modified (F/C/Fn)	Target Class Mentioned (%)
SWE-bench Verified	500	Python	Issue	253.6 (277.6)	38 (25)	2.6/1.1/6.7 (2/0/5)	15.9 (0.0)
SWE-bench Pro	731	Multiple	Issue	270.0 (107.6)	300 (173)	7.2/2.4/14.5 (5/1/11)	13.7 (0.0)
Multi-SWE-bench	2,132	Multiple	Issue	205.8 (112.2)	224 (64)	6.6/1.8/11.6 (3/0/5)	14.0 (0.0)
FEA-Bench	1,401	Python	Issue	174.7 (89.4)	222 (155)	5.0/2.8/16.3 (4/2/12)	15.5 (0.0)
DeepSWE	113	Multiple	Issue	66.7 (29.8)	1,670 (1,492)	11.6/11.9/80.7 (8/6/72)	31.8 (6.7)
FeatureBench	200	Python	Specification	490.6 (460.8)	2,050 (1,391)	15.4/11.3/78.3 (10/9/54)	38.7 (30.9)
LoLBench	**100**	**Multiple**	**Proposal**	**2,393.6 (1,545.1)**	**5,499 (2,774)**	**64.9/40.1/195.9 (39/21.5/117.5)**	**7.3 (2.8)**

fail or error before implementation and pass after applying the reference implementation. To allow semantically equivalent implementations with different structures from the reference, F2P tests check system-level or end-to-end behavior through public interfaces rather than private functions. A P2P test must pass both before and after the reference implementation and serves as a regression guard, and it may range from unit to system level. We package the repository, build dependencies, test runner, and reference solution in an offline Docker environment. We retain a task only when the reference solution applies successfully, all F2P tests pass after the solution, and all P2P tests pass both before and after it. Appendix H provides a task example.

3.4 Task Enhancement

Because system-level tests are limited in the wild, the original F2P tests may evaluate only part of the functionality in the EP and implementation. We therefore augment each F2P suite along two dimensions: behavioral distinguishability and solution coverage. For behavioral distinguishability, we construct incorrect solution mutants using three proposal-level transformations: *Section Revert*, which removes an implementable capability; *Requirement Mismatch*, which alters a required value, condition, or boundary; and *Semantic Mutation*, which changes the required structure or execution flow. We then generate new F2P tests to kill incorrect solution mutants that the original F2P tests cannot kill. We retain a generated F2P test only if it kills its target mutant and satisfies the same public-behavior restrictions. For solution coverage, we identify tasks whose original F2P suite has line coverage below 50%, and generate new F2P tests to improve their coverage. Appendix C.2 details the F2P test augmentation methods. Finally, we map each F2P test to the implementable sections in EPs whose required behavior it asserts, based on the behavior relationships confirmed in the semantic review and the implementation coverage. These mappings enable fine-grained evaluation by supporting section-level assessment of partial progress.

3.5 Benchmark Statistics

We present the comparison between LoLBench and recent coding benchmarks in Table 1. LoLBench contains 100 tasks from 29 software systems in five domains and eight programming languages. On average, its proposals contain approximately 5,000 words, its repositories contain 2.4 million source LoC, and its implementation PRs change 5,499 LoC, 65 files, 40 classes, and 196 functions. The repository scale and broad cross-module changes indicate significantly higher implementation complexity compared with other benchmarks. Furthermore, only 7.3% of target class names appear in the proposals, substantially less than in other benchmarks with large implementations. The limited implementation guidance and large abstract proposals demonstrate the substantial perception complexity of LoLBench.

Table 2: Main results of 28 agents on LOLBENCH across 100 tasks. "Resolved", "Completed", "F2P Pass", and "P2P Pass" report percentages. We average "turns", "time", "input tokens", "generated tokens", and "cost" over tasks. "Time" reports average wall-clock minutes per task. "Input tokens" includes cached input and is reported in millions. "Generated tokens" sums output and reasoning tokens. All models use their second-highest reasoning effort. Bold and underline denote the best and second-best performance values within each scaffold, respectively. Claude Code with GPT-5.6 Sol and Codex with Opus 5 are unavailable due to provider issues.

Scaffold	Model	Resolved (%)	Completed (%)	F2P Pass (%)	P2P Pass (%)	Avg. Turns	Avg. Time (min)	Input Tokens (m)	Generated Tokens (k)	Avg. Cost ($)
CLAUDE CODE	DeepSeek V4 Flash	0.0	4.8	15.0	71.8	176.9	64.4	27.6	115.6	1.4
	Opus 5	**14.0**	**26.3**	**52.7**	73.0	165.8	57.7	17.7	151.5	15.3
	Kimi K3	4.0	10.1	32.5	76.1	187.9	110.7	36.3	149.9	16.9
	GLM-5.2	2.0	4.5	20.6	**78.3**	172.4	55.4	21.8	104.3	7.7
	MiniMax M3	3.0	4.0	12.2	73.5	343.4	30.2	44.6	66.0	4.2
CODEX	DeepSeek V4 Flash	0.0	3.3	18.1	72.1	197.5	56.2	22.1	125.3	0.6
	Kimi K3	2.0	8.8	**32.9**	**79.3**	188.3	83.5	22.2	119.3	11.5
	GPT-5.6 Sol	**3.0**	**13.6**	28.6	76.0	135.2	24.9	17.5	46.4	11.7
	GLM-5.2	1.0	4.8	32.0	78.8	204.2	48.8	22.8	136.3	6.3
	MiniMax M3	1.0	1.5	12.6	71.5	532.4	44.9	63.1	96.7	4.0
OPENCODE	DeepSeek V4 Flash	1.0	4.0	18.3	73.6	177.2	44.4	24.1	95.8	0.5
	Opus 5	**10.0**	**16.2**	**37.6**	65.8	148.2	77.6	30.3	140.0	18.6
	Kimi K3	3.0	8.3	32.1	76.6	171.4	82.6	29.7	114.3	13.8
	GPT-5.6 Sol	5.0	8.3	15.8	43.4	53.0	82.7	8.7	28.3	5.2
	GLM-5.2	2.0	5.1	18.2	**80.1**	193.4	54.9	22.7	80.2	6.7
	MiniMax M3	0.0	1.8	12.0	72.9	373.0	37.1	54.9	71.8	3.5
PI	DeepSeek V4 Flash	1.0	3.3	28.0	73.1	182.7	51.1	23.9	105.5	0.5
	Opus 5	**13.0**	**21.0**	**44.5**	80.2	204.4	186.0	34.1	714.0	39.7
	Kimi K3	1.0	8.6	33.5	**81.1**	175.8	94.0	29.3	124.8	13.9
	GPT-5.6 Sol	3.0	7.1	22.9	80.2	72.4	18.4	5.7	26.8	4.4
	GLM-5.2	2.0	5.6	17.0	69.9	218.6	52.2	26.6	90.2	9.3
	MiniMax M3	0.0	3.0	13.7	69.9	415.5	56.6	51.3	76.5	3.3
MINI-SWE-AGENT	DeepSeek V4 Flash	0.0	3.5	14.9	69.7	212.9	57.3	27.4	105.7	0.6
	Opus 5	**14.0**	**20.5**	**44.7**	73.9	211.1	185.5	40.7	752.0	43.6
	Kimi K3	5.0	11.1	34.0	**80.3**	240.0	108.4	51.5	178.1	22.4
	GPT-5.6 Sol	5.0	9.1	29.4	78.0	72.4	43.8	10.0	100.7	9.3
	GLM-5.2	0.0	2.5	16.9	73.9	246.3	40.7	27.8	79.8	9.0
	MiniMax M3	1.0	2.3	11.1	70.9	473.5	54.9	66.0	84.1	4.2

4 EVALUATION

4.1 EXPERIMENT SETUP

Models and scaffolds. We evaluate six recent models: DeepSeek V4 Flash (DeepSeek, 2026), Opus 5 (Anthropic, 2026c), Kimi K3 (Moonshot AI, 2026), GPT-5.6 Sol (OpenAI, 2026b), GLM-5.2 (Z.ai, 2026), and MiniMax M3 (MiniMax, 2026). We access all models through OpenRouter (OpenRouter, 2026) and use the second-highest reasoning effort for each model. We pair them with five scaffolds: Claude Code (Anthropic, 2026a), Codex (OpenAI, 2026a), OpenCode (Anomaly, 2026), Pi (Earendil Inc., 2026), and mini-SWE-agent (Lieret & Jimenez, 2025). Codex with Opus 5 and Claude Code with GPT-5.6 Sol are unavailable because of provider issues, leaving 28 agents.

Environment. We implement LOLBENCH with Harbor (Harbor Framework Team, 2026), which executes each task in a sandbox. We disable public network access to prevent cheating. Each attempt has a six-hour wall-clock limit and is terminated after one hour without observable progress. We give each agent one attempt at each task, except when provider or infrastructure failures require a retry.

Metrics. *Resolved* is the percentage of tasks accepted by the full task verifier. *Completed* is the percentage of all completed implementable sections over all implementable sections: a section is completed only when all F2P tests mapped to it pass. *F2P Pass* and *P2P Pass* are micro-averaged over all tests. We also report mean turns, wall-clock time, input tokens, generated tokens, and cost.

4.2 EFFECTIVENESS ANALYSIS

Modular development remains challenging. Table 2 reports results for all 28 agents. The best agents, Claude Code and mini-SWE-agent with Opus 5, resolve only 14% of the tasks. Across

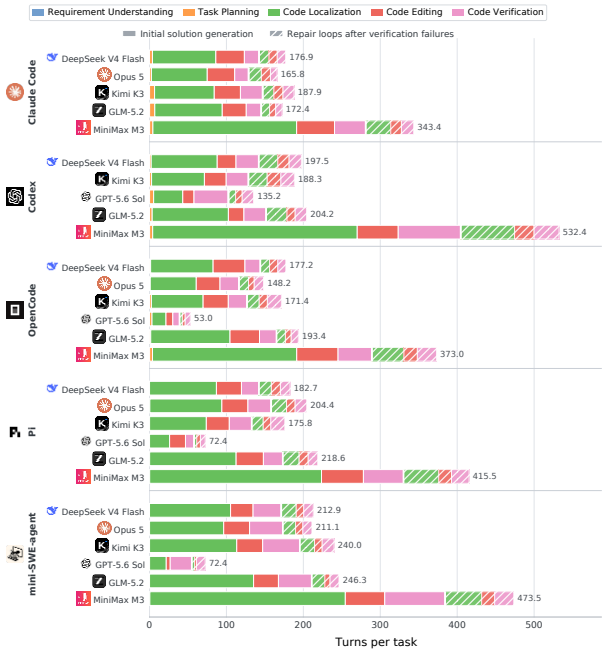


Figure 3: Average agent turns per task across five phases. Solid segments denote initial solution generation, and hatched segments denote repair loops after failed verification.

agents, the mean and median resolved rates are 3.4% and 2.0%. Fine-grained progress is also limited: Claude Code with Opus 5 attains the highest completed rate of 26.3%, whereas the mean and median are 8.0% and 5.3%. Thus, the low resolved rates are accompanied by limited section-level progress, rather than arising only from the strict all-or-nothing resolved rates.

Model choice can substantially affect effectiveness. We observe that Opus 5 achieves the highest resolved, completed, and F2P pass rates under every scaffold that supports it. We further analyze the four models available with all scaffolds and measure performance variability using relative standard deviation (RSD). Holding the scaffold fixed, the RSD across models averages 62.0% for Completed rate and 44.5% for F2P Pass rate. Holding the model fixed, the RSD across scaffolds averages 24.0% and 17.3%, respectively. Relative variability across models is about 2.6 times that across scaffolds for both metrics, indicating that agent effectiveness is highly sensitive to model choice.

4.3 Efficiency Analysis

Scaffold choice can substantially affect efficiency. With Opus 5, both Claude Code and mini-SWE-agent resolve 14 tasks, but mini-SWE-agent uses 3.2× the wall-clock time, 5.0× the generated tokens, and 2.8× the cost of Claude Code. Comparing the three open-source scaffolds, we find that OpenCode uses fewer turns, generated tokens, and less time than Pi and mini-SWE-agent.

Localization dominates agent activities. We assign agent turns to requirement understanding, task planning, code localization, code editing, or code verification, and we record repair-loop membership separately. As shown in Fig. 3, code localization is the largest phase for 26 of 28 agents during initial solution generation and for 21 during repair. Aggregated over the five phases, it accounts for 60.4% of initial-generation turns and 44.2% of repair turns. These results identify repository exploration and localization as the dominant activity for current agents in modular development tasks with high perception complexity. Appendix E adds analysis for phase distribution on token usage.

4.4 Failure Analysis

Method. We perform a failure analysis on the trajectories of failed tasks for all 28 agents. For each failed task, we attribute its failure to several failure modes in different phases mentioned in

Table 3: The heat map of failure attribution across the 28 agents on LOLBENCH. "Agent Failure" indicates failed tasks without analyzable trajectories, and "Solution Failure" indicates failed tasks with analyzable trajectories. The 49 agent failures for OpenCode with GPT-5.6 Sol are primarily due to timeouts before submission. The last seven columns show failure attribution across seven phases, and they sum to the "Solution Failure" column. Darker backgrounds indicate a larger share of total failure attribution for each agent's solution failures.

Scaffold	Model	Agent Failure	Solution Failure	Requirement Understanding	Task Planning	Code Localization	Code Editing	Code Verification	Self-Repair	Tool Use
CLAUDE CODE	DeepSeek V4 Flash	0	100	11.2	10.5	27.8	25.6	13.8	11.0	0.1
	Opus 5	6	80	6.9	6.2	25.1	18.1	13.8	9.9	0.0
	Kimi K3	5	91	5.1	4.4	26.4	25.9	9.8	18.8	0.6
	GLM-5.2	0	98	8.2	7.7	25.9	31.7	12.9	11.6	0.0
	MiniMax M3	1	96	9.4	9.2	24.2	29.3	12.2	11.5	0.2
CODEX	DeepSeek V4 Flash	0	100	4.5	3.9	39.9	24.6	18.3	8.5	0.3
	Kimi K3	0	98	2.1	1.7	41.7	23.8	16.8	11.4	0.5
	GPT-5.6 Sol	1	96	1.2	1.0	35.5	20.5	19.8	18.0	0.0
	GLM-5.2	0	99	3.3	2.9	42.1	25.2	16.3	9.2	0.0
	MiniMax M3	0	99	6.3	6.4	36.1	23.5	15.8	9.4	1.5
OPENCODE	DeepSeek V4 Flash	1	98	8.2	8.5	25.6	30.7	12.8	11.8	0.4
	Opus 5	11	79	5.7	5.1	27.1	16.9	9.5	14.7	0.0
	Kimi K3	1	96	5.6	6.1	26.5	25.5	13.7	18.1	0.5
	GPT-5.6 Sol	49	46	0.8	0.6	13.2	18.4	6.5	6.5	0.0
	GLM-5.2	0	98	7.8	7.5	26.5	32.2	12.6	11.4	0.0
	MiniMax M3	1	99	8.3	7.8	25.6	33.0	11.2	13.1	0.0
PI	DeepSeek V4 Flash	3	96	10.2	10.4	32.6	24.1	10.5	7.6	0.6
	Opus 5	7	80	7.1	6.5	30.7	17.6	8.5	8.2	1.4
	Kimi K3	1	98	4.2	4.3	38.4	28.7	10.7	11.1	0.6
	GPT-5.6 Sol	2	95	1.4	0.7	29.5	39.9	11.7	11.7	0.1
	GLM-5.2	1	97	10.4	10.1	27.3	29.9	10.7	8.2	0.4
	MiniMax M3	1	99	10.4	9.8	24.5	34.2	8.5	10.4	1.2
MINI-SWE-AGENT	DeepSeek V4 Flash	1	99	3.0	1.7	38.6	25.7	24.9	5.1	0.0
	Opus 5	12	74	0.6	0.6	30.6	14.2	16.6	10.8	0.6
	Kimi K3	1	94	1.1	1.1	37.5	18.5	23.0	11.7	1.1
	GPT-5.6 Sol	0	95	1.5	1.5	35.9	24.0	13.7	18.2	0.2
	GLM-5.2	1	99	4.1	3.2	39.0	23.4	20.3	8.8	0.2
	MiniMax M3	0	99	7.6	7.1	31.4	23.6	17.0	11.2	1.1

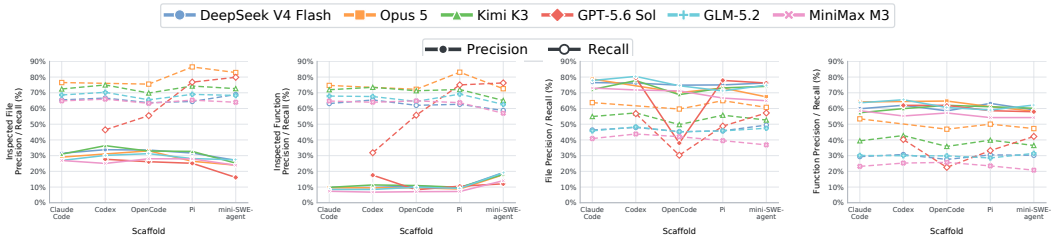
Fig. 3. We add self-repair and tool use as extra phases to separate failures that occur outside initial solution generation. We use pre-defined rules to detect observable localization, editing, verification, repair, and tool-use signals, and an outcome-blinded LLM judge to detect semantic requirement understanding and planning issues. For solution failures with multiple modes, we normalize confidence scores, so each solution failure contributes one unit of attribution. Table 3 reports phase-level attribution, and Appendix F provides the taxonomy and per-phase failure mode results.

Incomplete cross-module context is the dominant failure mode. Averaged over the 28 agents, code localization accounts for the largest 30.9 units of failure attribution, followed by code editing at 25.3 units. These values are fractional attribution weights aggregated across solution failures. Table 13 further shows that nearly all localization attribution, averaging 30.0 units, comes from `cross_module_context_missing`: the submitted solution reaches some target files and modules but also misses some. The phase attribution also varies with the scaffold. Among the four models shared by all scaffolds, Codex assigns 11.8–16.2 more attribution units to code localization than Claude Code, while mini-SWE-agent assigns 7.2–13.1 more. Code localization is the largest phase for 19 agents, while editing is the largest for the remaining nine. This shows that modular development tasks on large software systems require comprehensive perception and implementation capabilities.

4.5 PERCEPTION CAPABILITY ANALYSIS

To further evaluate the perception capability of current agents in modular development tasks, we compare the source files and functions each agent inspects or modifies with those modified by the reference implementation. We exclude newly added entities because they have no pre-existing location to recover. Figure 4 reports task-level macro averages.

Current agents struggle to pinpoint all related files and functions. Figure 4 shows incomplete code localization coverage at both the inspection and modification stages. Across all 28 agents, inspection recall is 46.4–86.4% for files and 31.9–83.1% for functions, indicating that some reference



(a) Inspected files. (b) Inspected functions. (c) Modified files. (d) Modified functions.

Figure 4: Macro-averaged precision and recall for code localization based on inspected (left) and modified (right) files and functions.

Table 4: Results of three agents on LoLBench with file trees and API specification in EPs. All columns are calculated the same way in Table 2. Parenthesized values report signed absolute changes computed by subtracting the baseline values in Table 2 from the values here.

Agent	Resolved (%)	Completed (%)	F2P Pass (%)	P2P Pass (%)	Avg. Turns	Avg. Time (min)	Generated Tokens (k)	Avg. Cost ($)
Opus 5 + Claude Code	**34.0** (+20.0)	**42.9** (+16.6)	**72.7** (+20.0)	81.1 (+8.1)	168.0 (+2.2)	50.8 (-6.9)	141.5 (-10.0)	15.2 (-0.1)
GPT-5.6 Sol + Codex	25.0 (+22.0)	32.8 (+19.2)	56.2 (+27.6)	**85.7** (+9.7)	123.8 (-11.4)	27.4 (+2.5)	47.8 (+1.4)	10.8 (-0.9)
DeepSeek V4 Flash + OpenCode	17.0 (+16.0)	17.7 (+13.7)	31.6 (+13.3)	69.4 (-4.2)	232.0 (+54.8)	55.1 (+10.7)	128.7 (+32.9)	0.9 (+0.4)

edit locations remain uninspected. Inspection precision is only 16.1–36.3% for files and 6.8–19.2% for functions, reflecting broad exploration beyond the entities modified by the reference implementation. Agents select some reference edit locations more precisely when making changes: across the 27 agents other than OpenCode with GPT-5.6 Sol, modification precision is 64.9–80.4% for files and 54.2–65.6% for functions. However, modification recall remains limited to 36.9–65.0% for files and 20.8–53.4% for functions. These results show that broad repository exploration does not ensure comprehensive coverage of edit locations when implementing abstract EPs.

Current agents perform substantially better on specifications than on abstract proposals. We follow the instruction format in NL2Repo-Bench (Ding et al., 2025) and reformulate EPs as specifications by appending a file tree and API specification derived from the reference solution. We evaluate three selected agents on the specifications and report the paired results in Table 4. Compared with EPs, the specifications increase resolved rate by 16–22 percentage points, completed rate by 13.7–19.2 points, and F2P pass rate by 13.3–27.6 points. For Claude Code with Opus 5, F2P pass rate rises from 52.7% to 72.7%. These paired improvements show that explicit implementation guidance in specifications can remove much of the difficulty, while agents need perception capability to convert abstract EPs into detailed specifications.

5 Conclusion and Future Work

We introduce LoLBench, a multilingual benchmark of 100 modular development tasks across 29 large software systems. Built from human-written enhancement proposals, LoLBench evaluates the full process of grounding user intent and high-level design in an existing codebase and producing verified implementations. Its tasks combine perception complexity with implementation complexity. Across 28 agents, the highest resolved rate is only 14%. Our analyses identify incomplete cross-module context as the major failure mode. Providing reference-derived implementation guidance improves resolved rates by 16–22 percentage points across three agents, highlighting the challenge of translating proposals into concrete changes on large software systems. Future work should focus on improving the perception capability of coding agents to handle practical modular development tasks on large software systems.

AI USE STATEMENT

In this work, we used generative AI tools to implement failure-analysis methods, quality-check the proposed dataset, and support qualitative data analysis. We used LLM judges and LLM-based approaches to help generate dataset artifacts. We did not use generative AI tools for designing or providing feedback on research methodology or experiments. Developing theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, and proposing or refining hypotheses are not applicable to this work. Additionally, we used generative AI tools for polishing the text and figures in the paper. We have reviewed all AI-assisted work. For example, we manually checked the outputs of LLM judges used in the dataset building and evaluation, as well as the correctness of the failure-analysis method and data analysis. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The LOLBENCH benchmark introduced in this paper is available at `https://huggingface.co/datasets/lolbench26/LoLBench`. The appendix provides further details of the data construction and result analyses. Appendix A lists the statistics of all the software systems selected in LOLBENCH. Appendices B and C document EP–PR validation and quality ranking, solution mutant generation, and F2P test augmentation, including the rules and prompts used. Appendices D and E provide per-language results and phase-level token usage analyses, while Appendix F specifies the failure taxonomy, detection rules, and attribution procedure used in our failure analysis.

REFERENCES

Anomaly. Opencode, 2026. URL `https://opencode.ai/`. Accessed: Sep 2, 2026.

Anthropic. Claude code, 2026a. URL `https://claude.com/product/claude-code`. Accessed: Sep 2, 2026.

Anthropic. Introducing claude opus 4.8, 2026b. URL `https://www.anthropic.com/news/claude-opus-4-8`. Accessed: Sep 18, 2026.

Anthropic. Introducing claude opus 5, 2026c. URL `https://www.anthropic.com/news/claude-opus-5`. Accessed: Sep 2, 2026.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. doi: 10.48550/ARXIV.2108.07732. URL `https://arxiv.org/abs/2108.07732`.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. doi: 10.48550/ARXIV.2107.03374. URL `https://arxiv.org/abs/2107.03374`.

DeepSeek. Deepseek-v4-flash update, 2026. URL `https://api-docs.deepseek.com/updates/#deepseek-v4-flash-update`. Accessed: Sep 18, 2026.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can AI agents solve long-horizon software engineering tasks? *CoRR*, abs/2509.16941, 2025. doi: 10.48550/ARXIV.2509.16941. URL `https://doi.org/10.48550/arXiv.2509.16941`.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, Weiran Shi, Zaiyuan Wang, Daoguang Zan, Chenchen Zhang, Xiaoxu Zhang, Qizhi Chen, Xianfu Cheng, Bo Deng, Qingshui Gu, Kai Hua, Juntao Lin, Pai Liu, Mingchen Li, Xuanguang Pan, Zifan Peng, Yujia Qin, Yong Shan, Zhewen Tan, Weihao

Xie, Zihan Wang, Yishuo Yuan, Jiayu Zhang, Enduo Zhao, Yunfei Zhao, He Zhu, Chenyang Zou, Ming Ding, Jianpeng Jiao, Jiaheng Liu, Minghao Liu, Qian Liu, Chongyao Tao, Jian Yang, Tong Yang, Zhaoxiang Zhang, Xinjie Chen, Wenhao Huang, and Ge Zhang. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents. *CoRR*, abs/2512.12730, 2025. doi: 10.48550/ARXIV.2512.12730. URL https://doi.org/10.48550/arXiv.2512.12730.

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.

Earendil Inc. Pi, 2026. URL https://pi.dev/. Accessed: Sep 2, 2026.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, 2026. URL https://doi.org/10.5281/zenodo.20953922.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In Joaquin Vanschoren and Sai-Kit Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks. *CoRR*, abs/2607.07946, 2026. doi: 10.48550/ARXIV.2607.07946. URL https://doi.org/10.48550/arXiv.2607.07946.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *International Conference on Learning Representations*, 2024.

Mohammad Abdullah Matin Khan, M. Saiful Bari, Xuan Long Do, Weishi Wang, Md. Rizwan Parvez, and Shafiq R. Joty. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *CoRR*, abs/2303.03004, 2023. doi: 10.48550/ARXIV.2303.03004. URL https://doi.org/10.48550/arXiv.2303.03004.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, et al. DevEval: A Manually-Annotated Code Generation Benchmark Aligned with Real-World Code Repositories. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 3603–3614, 2024.

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 *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 17160–17176, 2025.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://www.science.org/doi/abs/10.1126/science.abq1158.

Kilian Lieret and Carlos E. Jimenez. mini-SWE-agent. GitHub software repository, 2025. URL https://github.com/SWE-agent/mini-swe-agent. Accessed: Sep 22, 2026.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. In *International Conference on Learning Representations*, 2026.

MiniMax. MiniMax M3: Frontier Coding, 1M Context, Native Multimodality — All in One Model, 2026. URL https://www.minimax.io/blog/minimax-m3. Accessed: Sep 2, 2026.

Moonshot AI. Kimi K3: Open Frontier Intelligence, 2026. URL https://www.kimi.ai/blog/kimi-k3. Accessed: Sep 2, 2026.

OpenAI. Swe-bench verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified/. Accessed: Sep 2, 2026.

OpenAI. Codex, 2026a. URL https://openai.com/codex/. Accessed: Sep 2, 2026.

OpenAI. Previewing GPT-5.6 Sol: a next-generation model, 2026b. URL https://openai.com/index/previewing-gpt-5-6-sol/. Accessed: Sep 2, 2026.

OpenAI. GPT-5.6 Luna model, 2026c. URL https://developers.openai.com/api/docs/models/gpt-5.6-luna. Accessed: Sep 22, 2026.

OpenRouter. OpenRouter API, 2026. URL https://openrouter.ai. Accessed: Sep 2, 2026.

Guido van Rossum, Pablo Galindo Salgado, and Lysandros Nikolaou. PEP 617 – New PEG parser for CPython, 2020. URL https://peps.python.org/pep-0617/. Accessed: Sep 2, 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *International Conference on Learning Representations*, 2025.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-SWE-agent: Can Software Engineering Agents Self-Evolve on the Fly? *arXiv preprint arXiv:2511.13646*, 2025.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629*, 2022.

Z.ai. Glm-5.2: Built for long-horizon tasks, 2026. URL https://z.ai/blog/glm-5.2. Accessed: Sep 2, 2026.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Aoyan Li, Lu Chen, Xiaojian Zhong, et al. Multi-SWE-bench: A Multilingual Benchmark for Issue Resolving. *Advances in Neural Information Processing Systems*, 38, 2025.

Shaoqiu Zhang, Yuhang Wang, Jialiang Liang, Yuling Shi, Wenhao Zeng, Maoquan Wang, Shilin He, Ningyuan Xu, Siyu Ye, Kai Cai, et al. Swe-explore: Benchmarking how coding agents explore repositories. *arXiv preprint arXiv:2606.07297*, 2026.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604, 2024.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, et al. FeatureBench: Benchmarking Agentic Coding for Complex Feature Development. *arXiv preprint arXiv:2602.10975*, 2026.

Table 5: Statistics of projects in LoLBench. "Language" is assigned at the project level and applies to every task from that project. "Repo Size" is the mean source code lines excluding comments and blank lines. "GitHub Stars" are as of September 18, 2026, and "Maintenance Days" is the number of days elapsed from each repository's creation on GitHub to September 18, 2026.

Domain	Project	Language	#Tasks	Repo Size (kLoC)	GitHub Stars	Maintenance Days
Compilers	CPython	Python	18	1,344.4	77,201	3,507
	Cargo	Rust	1	230.1	15,489	4,581
	Mypy	Python	1	213.2	20,644	5,033
	OpenJDK	Java	14	7,744.6	23,358	2,923
	PHP	C	4	2,615.7	40,393	5,573
	Roslyn	C#	1	5,981.0	20,672	4,268
	Ruff	Rust	1	297.9	49,679	1,501
	Rust	Rust	4	2,390.0	118,921	5,938
	TypeScript	TypeScript	4	2,379.0	111,099	4,476
Data Science	Apache Arrow	C++	2	906.2	17,136	3,866
	Apache DataFusion	Rust	2	475.2	9,323	1,980
	Apache Flink	Java	13	1,824.0	26,342	4,486
	Apache Iceberg	Java	1	635.7	9,248	2,860
	Apache Kafka	Java	7	778.2	33,749	5,513
	NumPy	Python	2	335.2	32,765	5,849
	Pandas	Python	1	421.1	49,734	5,869
	Scikit-learn	Python	1	218.2	67,288	5,876
Distributed Systems	cert-manager	Go	1	166.0	14,081	3,404
	gRPC-Go	Go	1	208.8	23,069	4,302
	Kubernetes	Go	7	3,805.3	127,805	4,487
	Kueue	Go	1	1,947.1	2,984	1,675
	OpenTofu	Go	4	438.7	30,212	1,129
	Prometheus	Go	3	250.7	66,114	5,046
Databases	DuckDB	C++	1	1,080.6	41,492	3,006
	Presto	Java	1	897.8	16,739	5,153
	Vitess	Go	1	1,043.9	21,344	4,831
Web	ASP.NET Core	C#	1	1,821.5	38,445	4,574
	Django	Python	1	246.3	91,134	5,256
	FastAPI	Python	1	52.5	102,419	2,841

A Statistics of Projects in LoLBench

Table 5 details the project composition of the 100 tasks in LoLBench. The benchmark spans 29 projects in five domains: compilers contribute 48 tasks, data science systems 29, distributed systems 17, and databases and web frameworks three each. It also covers eight programming languages, with 36 Java, 25 Python, 18 Go, 8 Rust, 3 C++, 4 C, 4 TypeScript, and 2 C# tasks.

The projects we selected are well-maintained large software systems. Based on the per-project means, all repositories exceed 50k source LoC, and 12 exceed one million. Mean repository size ranges from 52.5k source LoC for FastAPI to 7.7 million for OpenJDK. As of September 18, 2026, their GitHub stars range from 2,984 to 127,805, and their elapsed times since repository creation on GitHub range from 1,129 to 5,938 days. This indicates that these software systems have a significant impact on the open-source community and have been maintained for a long time. Therefore, LoLBench combines diverse languages and architectural styles while consistently requiring agents to navigate complex software systems.

B Details of EP–PR Matching

This section specifies the quality-control pipeline used in the EP–PR matching phase in Sec. 3. For each candidate, the mapping artifact contains (1) a Section Classification Summary (SCS) covering every EP section, (2) a PR File Summary (PFS), (3) one body block for each mapped implementable section, with separate tables for direct modifications and associated support changes, and (4) tables for unmapped EP sections and PR files. Source-code rows are additionally resolved to changed classes and functions. A candidate must pass the static and semantic validation gates before it is quality-ranked.

B.1 STATIC VALIDATION

The 28 static rules are deterministic checks over the EP, the PR, and the mapping artifact. They are:

1. **PR-file completeness.** Every changed PR file is assigned to exactly one role: directly mapped, associated, or unmapped. The three-way total equals the PR file count.
2. **File validity.** Every file named by the mapping occurs in the PR diff.
3. **EP-section completeness.** Every Markdown or reStructuredText heading in the EP occurs in the SCS and is placed consistently: mapped body block, unmapped implementable table, or SCS-only contextual entry.
4. **Controlled vocabularies.** Section, PR-file, and unmapped-reason labels belong to their predefined vocabularies, and the relevant tables use the required schemas.
5. **SCS consistency.** The SCS has the required columns, implementable sections use implementation/evaluation categories, non-implementable sections use knowledge/contextual/process categories, and its mapping flags agree with the body blocks.
6. **Document order.** Mapped sections are numbered and occur in the same order as in the EP.
7. **Per-section proportions.** Each body block reports direct, associated, and accounted file counts and percentages that agree with its tables.
8. **Global arithmetic.** The reported EP-section coverage and PR file coverage are arithmetically correct.
9. **Test-plan coverage.** When the EP and PR contain test sections and test files, the sections are implementable evaluation sections, and the test files are assigned to them.
10. **Implementability–category agreement.** The implementability flag and section category form a valid pair.
11. **PFS consistency.** The PFS has the required columns, direct and associated roles are mutually exclusive, and every PFS assignment agrees with the body or unmapped-file table.
12. **Unmapped-section consistency.** The unmapped-section table contains exactly the SCS rows marked implementable but unmapped, with the same identifiers and a permitted reason.
13. **No redundant section lists.** Contextual and process sections appear only in the SCS, rather than in additional stand-alone lists.
14. **Multi-section accounting.** If one file contributes to several sections, the PFS lists every owning or associated section.
15. **Statistics consistency.** Regenerated per-instance statistics agree with the mapping, including category totals and coverage values.
16. **Unique SCS entries.** SCS identifiers and titles are unique, and code fragments are not mistaken for section headings.
17. **Unique PFS entries.** Each PR file has exactly one PFS row.
18. **Sequential numbering.** SCS and PFS rows are consecutively numbered, and mapped body blocks are in ascending order.
19. **SCS–body correspondence.** Each body block corresponds to one SCS row marked implementable and mapped, and conversely.
20. **File-level entries.** Mapping tables name individual files, not directories, globs, or aggregate paths.
21. **Unmapped-count arithmetic.** Unmapped EP sections and PR files equal their respective totals minus mapped/accounted items.
22. **Verbatim evidence.** Every mapped body block quotes text that occurs verbatim in the appropriate EP section and includes a concise requirement summary.
23. **No synthetic sections.** Every SCS title is backed by an actual EP heading.
24. **Modification-summary coverage.** Every directly mapped file has exactly one corresponding summary bullet, with no extra bullets.
25. **Non-overlapping quotations.** A section quotation neither absorbs a mapped child section nor overlaps the quotation of another body block.
26. **Self-contained EP.** The EP contains substantive requirement text rather than only a link to an external design document, and ambiguous cases are forwarded to semantic rule S8.

27. **Touched-scope coverage.** For each source file, every changed class/function pair recovered from the pre- and post-PR syntax trees occurs in at least one direct or associated row. Pure file-level changes are exempt.
28. **Unique scope ownership.** Each changed (file, class, function) tuple has one canonical assignment, even when its file contributes to multiple EP sections.

The controlled vocabularies in rule 4 are: {knowledge, implementation, evaluation, contextual, process} for EP sections, {source, test, test-data, data, build, documentation, generated, vendor} for PR files, and {deferred, pre-existing, out-of-scope, documentation-only, runtime-behavior, external-dependency, test-only, insufficient-context} for an unmapped implementable section.

Rules 27–28 use syntax-tree scopes from both sides of each diff: added lines are attributed using the post-PR tree and deleted lines using the pre-PR tree. This avoids treating a multi-purpose file as an indivisible unit while still requiring every changed code entity to have a unique semantic owner.

B.2 SEMANTIC VALIDATION

Static checks cannot decide whether an EP section matches a code change in the PR. We therefore run eight focused LLM audits. Each audit receives the complete EP and mapping, as well as the PR patches and the extracted class/function scopes. We use separate prompts to reduce interference between criteria. The common prompt wrapper is:

> *You are a quality auditor for LoLBench EP–PR mapping files. Read the provided EP, PR data, and mapping. Apply only rule Sk. Report clear, actionable errors, identifying the affected section, file, or scope. Return a JSON list of issue strings; return `[]` when no issue is found (except for the verdict format specified by S8). Do not report borderline stylistic judgments.*

The rule-specific instructions appended to this wrapper are:

S1. **Section coverage.** Extract all Markdown and reStructuredText headings, and report headings missing from the SCS, SCS entries unsupported by the EP, and ordering errors.
S2. **Section classification.** Read each section and verify both its implementability and its category: implementation/evaluation for concrete behavior or test requirements, and knowledge/contextual/process for background, rationale, and other contextual descriptions.
S3. **Test mapping.** Verify that changed test files are assigned to the finest available test subsection and that corresponding test sections are implementable evaluation sections, rather than assigning tests to an unrelated implementation section.
S4. **PR-file category.** Classify each patch as source, test, test-data, data, build, documentation, generated, or vendor. Inspect patch content in ambiguous cases instead of relying only on paths or suffixes.
S5. **Mapping semantics.** For every mapped section, verify that its direct files/scopes implement the stated behavior, that the most specific matching section owns them, and that associated rows have genuine support relationships. Report clear cross-layer or cross-feature mismatches.
S6. **Citation and summary fidelity.** Verify that each quotation is exact text from the section named by its path, that any truncation is a faithful prefix, and that the one-to-three-sentence summary is accurate and contains no fabricated claim.
S7. **Direct versus associated changes.** Put only changes explicitly required by the section in the direct table. Treat tests, generated files, dependencies, vendored code, build/CI files, and documentation as associated unless the section explicitly requires them. Also verify that each direct file's modification summary accurately explains its contribution.
S8. **External-link sufficiency.** Enumerate external links and decide whether each is incidental background or contains essential specification text. Essential accessible content must be inlined. If essential content is inaccessible, return `invalid:<reason>` and reject the candidate; otherwise return `complete` or `inlined` as appropriate.

The semantic audits are run through Claude Code (Anthropic, 2026a) as independent subagents in batches of five. For S1–S7, an empty issue list is a pass. Rule S8 is applied to EPs containing

Table 6: Dimensions and score anchors used to evaluate EP-to-PR mappings. Each dimension is scored from 1 to 10, and the final section score is their weighted sum.

Dimension	Weight	Score anchors
Requirement coverage	40%	**9–10:** all concrete requirements are mapped; **7–8:** one minor sub-requirement or edge case is missing or indirect; **5–6:** an important requirement is only partly covered; **3–4:** several central requirements are missing or mapped only to support; **1–2:** little valid coverage.
Mapping precision	25%	**9–10:** no meaningful unrelated behavior; **7–8:** only incidental support or shared helpers; **5–6:** substantial unrelated behavior; **3–4:** many loosely related scopes that belong elsewhere or should be associated; **1–2:** mostly unrelated scopes.
Scope correctness	15%	**9–10:** correct section, direct/associated role, category, and unique scope ownership; **7–8:** a few minor helper or category errors; **5–6:** several incorrect assignments, but still usable; **3–4:** assignment is mostly based on file proximity; **1–2:** ownership is largely incorrect.
Granularity	10%	**9–10:** a fine-grained semantic boundary; **7–8:** a few broad rows, but coherent ownership; **5–6:** a large mixed set of scopes; **3–4:** the section is mostly a catch-all; **1–2:** no useful granularity.
Requirement specificity	10%	**9–10:** concrete APIs, syntax, semantics, configuration, or deliverables; **7–8:** only routine details are implicit; **5–6:** substantial inference is required; **3–4:** very short or vague; **1–2:** too vague to justify the mapping without external information.

external links, especially those flagged by Appendix B.1, Rule 26. A curator reviews the reported evidence, regenerates a faulty mapping rather than locally patching it, and reruns both validation stages. Only candidates with no unresolved static or semantic findings proceed to quality ranking.

B.3 Quality ranking

The validation rules are hard constraints, but ranking measures the quality of a valid mapping. The LLM judge considers only semantic alignment between the EP and PR, not whether the upstream implementation itself is a good engineering solution. It grades every implementable section on the five dimensions in Table 6, and non-implementable sections are excluded in this process.

Judge prompt. Bracketed fields below are replaced with the full candidate artifacts.

> *You are grading the semantic quality of an EP-to-PR mapping, not the quality of the PR implementation. Read [EP], [PR patches and changed class/function scopes], and [mapping]. For every implementable EP section: (1) extract its concrete requirement claims; (2) determine whether its direct files/scopes cover all claims without importing unrelated behavior; (3) verify file, class, function, and direct-versus-associated ownership; (4) assess whether the mapping is sufficiently fine-grained; and (5) assess whether the EP text is specific enough to justify the mapping. Score requirement coverage, mapping precision, scope/function correctness, granularity, and requirement specificity from 1 to 10 using the supplied rubric. Assign a semantic importance weight independently of the number of mapped files. Return one row per implementable section containing the five scores, importance weight, weighted section score, and a concise evidence-based reason. Then return the weighted instance score and its main reason. Do not apply an additional instance-level adjustment.*

Let d_{ij} be the score of section i on dimension j. Its quality score is

$$q_i = 0.40d_{i,\text{cov}} + 0.25d_{i,\text{prec}} + 0.15d_{i,\text{scope}} + 0.10d_{i,\text{gran}} + 0.10d_{i,\text{spec}}.$$

The judge separately assigns each section an importance weight w_i using Table 7.

The number of mapped files never affects w_i. Broad or weakly justified mappings are instead penalized through precision, granularity, and requirement specificity.

Table 7: Semantic importance weights for EP sections. Modifiers are applied independently after choosing the base weight. Implementable section weights are clipped to $[0.1, 1.0]$; non-implementable sections always receive zero.

Type	Weight	Section semantics or condition
Base	1.0	Core public API, syntax, protocol/schema, runtime semantics, or principal feature behavior.
Base	0.8	Important internal mechanism required by the main behavior.
Base	0.6	Compatibility, migration, configuration, rollout, error handling, or deprecation behavior.
Base	0.4	Explicit test, validation, benchmark, or performance requirement.
Base	0.2	Explicit documentation, build, generated-output, fixture, or support requirement.
Base	0.0	Non-implementable knowledge, context, or process section.
Modifier	$+0.1$	Strong normative language such as *must*, *shall*, or *required*, or an explicitly named goal.
Modifier	$+0.1$	Many other implementable sections conceptually depend on this section.
Modifier	-0.1	Primarily illustrative or example-based.
Modifier	-0.1	Vague with weak requirement force, but still implementable.
Modifier	-0.2	Optional, deferred, exploratory, or nice-to-have.

The candidate score is the importance-weighted mean

$$Q = \frac{\sum_i w_i \, q_i}{\sum_i w_i}.$$

Section and candidate scores are rounded to one decimal place. We retain a candidate only if it has passed both validation gates, receives $Q \geq 8.0$, and passes final expert review.

C Details of Task Enhancement

C.1 Incorrect solution mutant generation

We construct incorrect solution mutants to identify requirement-level deficiencies in the original F2P suites. Each mutant is a complete candidate solution that intentionally violates a section in the EP. An accepted mutant must preserve the behavior exercised by all P2P tests and have a documented functional difference from the reference solution.

EP mutation. We do not directly mutate solutions. Instead, we mutate one implementable section in the EP and first generate EP mutants. We apply three methods for EP mutations:

1. **Section Revert.** Remove or reverse behavior attributed to a section, yielding a partial implementation that omits a required functionality while retaining the support needed by the remaining implementation.
2. **Requirement Mismatch.** Introduce a plausible semantic error relative to an explicit requirement clause. Examples include an incorrect default or validation boundary, omitted parameter propagation, an incorrect enumeration or option mapping, altered error behavior, or a disabled feature flag.
3. **Semantic Mutation.** Alter the required structure or execution flow while preserving a plausible implementation. Examples include bypassing a new dispatch path, calling the old implementation, dropping a required method call, returning a default, empty, or routing execution to an incorrect handler.

Solution Mutation. Given the EP mutants and the reference solutions, we then ask LLMs to modify the reference solutions based on the EP mutants. For the generated solution candidates, we verify their semantic non-equivalence with the original reference solutions. Every candidate must identify the exact mapped section and requirement it violates, describe the reference and mutated behaviors, and specify a concrete scenario under which they diverge. The candidate also includes

Table 8: Distribution of retained solution mutants, excluding retired or specification-equivalent candidates.

Mutation method	Count	Share (%)
Section Revert	158	9.36
Requirement Mismatch	1,205	71.39
Semantic Mutation	325	19.25
Total	**1,688**	**100.00**

Table 9: F2P test augmentation. "Overall" combines original and added tests. Coverage is the mean per-task coverage of F2P tests. Mutation scores use the 1,688 retained mutants in Table 8.

Metric	Original	Overall
F2P tests	1,480	2,234
Added: mutant-killing	—	622
Added: coverage	—	132
Mutants killed	1,031	1,620
Mutation score (%)	61.08	95.97
Mean F2P coverage (%)	60.07	78.50

a non-equivalence rationale explaining how this difference violates the requirement. These records must allow a reviewer to assess the functional difference without relying on an F2P failure. We then validate the candidates on P2P tests and filter out candidates that cannot pass. Finally, we combine LLM judges and expert reviewers to review the remaining candidates and select those with distinct functionality as solution mutants. Table 8 summarizes the solution mutants generated by three mutation methods.

C.2 F2P TEST AUGMENTATION

Mutation-guided test generation. We run all original F2P tests on generated solution mutants. When the original F2P tests cannot kill certain solution mutants, we use Codex with GPT-5.6 Sol and Claude Code with Opus 4.8 (Anthropic, 2026b) to add tests that distinguish these mutants from the reference solution. Each candidate targets one or more surviving mutants and asserts an EP-specified outcome through system-level or end-to-end behavior. The EP defines the expected outcome, while the mutant's documented distinguishing scenario guides the input choice.

Coverage-guided test generation. We identify tasks whose original F2P test suite has line coverage lower than 50%. For these tasks, we direct Codex with GPT-5.6 Sol and Claude Code with Opus 4.8 to add new F2P tests targeting uncovered reference-patch behavior.

Test audit. We audit every generated F2P test before inclusion. We inspect the selected test body and the source symbols it references. Tests that invoke or otherwise depend on internal symbols are rejected. We also reject tests that directly use newly introduced source symbols whose names are absent from the EP. Rejected candidates must be rewritten to exercise permitted public behavior. We deduplicate candidates, discard unstable tests, and retain the added tests separately from the original suite. Table 9 summarizes the statistics of added F2P tests. On all 1,688 solution mutants, the original tests kill 1,031 (61.08%), while the combined original and added tests kill 1,620 (95.97%), leaving only 68 surviving mutants. The combined test suite also improves the line coverage from 60.07% to 78.50%.

Mutation-guided generation prompts. The following prompt templates summarize the mutation-guided F2P generation instructions. Angle brackets denote task-specific inputs, including the EP, evaluation environment, original tests, reference solution, and surviving-mutant records.

System prompt.

You generate additional fail-to-pass (F2P) tests for a software task described by an enhancement proposal (EP). Your goal is to expose requirement violations in solution mutants that survive the original F2P suite. Use the EP as the behavioral oracle and the mutants' distinguishing scenarios to identify missing checks.

Follow these constraints:

1. *Exercise system-level or end-to-end behavior through permitted public entry points. Assert the outcome required by the EP, rather than merely asserting that the result differs from a mutant.*
2. *Do not invoke, import, construct, or assert directly on internal source symbols. Do not directly use a newly introduced source symbol unless its exact name appears in the EP. Generated tests are audited, and candidates that violate these rules are rejected.*
3. *Use the project's existing test framework and execution conventions. Add tests separately from the original suite; do not modify the reference solution or original tests.*
4. *Identify the EP clause, input, expected outcome, public entry point, and targeted mutant IDs for each candidate. Prefer a parameterized test when several mutants can be distinguished through the same requirement and entry point.*
5. *An accepted candidate must fail or error before implementation, pass with the reference solution, and fail on every targeted mutant. Remove duplicates and reject unstable candidates.*
6. *If no permitted public behavior distinguishes a mutant, report that limitation for review. Do not bypass the restriction by calling an internal symbol.*

Task prompt.

Generate new F2P test cases for the following task.

EP: <enhancement proposal and section identifiers>

Repository and evaluation environment: <base revision, source tree, test framework, and execution commands>

Reference solution: <reference implementation patch>

Original F2P suite: <test sources, test selectors, and mutant execution results>

Surviving mutants: <mutant IDs, patches, violated requirement clauses, distinguishing scenarios, and expected differences from the reference solution>

Test-surface constraints: <permitted public entry points, internal-symbol restrictions, and newly introduced source symbols>

For each surviving mutant, translate its distinguishing scenario into a test that asserts the EP-required outcome through permitted public behavior. Return the added test patch and executable test selectors. For each candidate, report its EP clause, targeted mutant IDs, input, expected outcome, and public entry point. Explain why its imports, calls, constructions, and assertions satisfy the symbol restrictions.

Run each candidate before implementation, with the reference solution, and with its targeted mutants; report the observed results. Audit the selected test body for internal-symbol dependencies and newly introduced symbols absent from the EP. Rewrite or discard any candidate that fails this audit or the execution checks. Report any surviving mutant for which no admissible test could be produced.

Coverage-guided generation prompts. The following templates summarize the coverage-guided F2P generation instructions. They generate extra F2P test cases for tasks whose combined existing F2P and P2P suites have line coverage below 50%.

System prompt.

You generate additional fail-to-pass (F2P) tests for a software task described by an enhancement proposal (EP). Your goal is to exercise uncovered executable lines in the reference implementation through system-level or end-to-end behavior, while asserting outcomes required by the EP.

Follow these constraints:

1. *Use the coverage report to identify executable reference-patch lines not reached by the union of the current F2P and P2P suites. Prioritize large connected uncovered spans and relate each target to an implementable EP section using the EP–implementation mapping.*

2. *Identify a permitted public entry point and an input that reaches each target span. Assert the EP-required observable outcome, rather than an internal implementation detail or a coverage count alone.*
3. *Do not invoke, import, construct, or assert directly on internal source symbols. Do not directly use a newly introduced source symbol unless its exact name appears in the EP. Audit each candidate for these restrictions.*
4. *Use the project's existing test framework and execution conventions. Add tests separately from the original suite; do not modify the reference implementation or original tests. Deduplicate candidates and reject unstable tests.*
5. *Each retained candidate must fail or error before implementation because the required behavior is absent, pass with the reference implementation, and exercise previously uncovered target lines. A test that passes before implementation is not an F2P test, even if it increases coverage.*
6. *Rerun coverage after adding accepted tests and report newly covered reference-patch lines and the updated F2P–P2P union coverage using the same executable-line denominator. Record the EP clause, public entry point, input, expected outcome, test selector, and validation results for each candidate.*
7. *If an uncovered span cannot be reached through permitted public behavior, document the span and the reason for review. Do not bypass the restrictions with an internal-symbol test or weaken F2P validity to meet a coverage target.*

Task prompt.

Generate new F2P tests for the uncovered requirement behavior in the following task.

EP and implementation mapping: <enhancement proposal, section identifiers, and mapped implementation files and scopes>

Repository and evaluation environment: <base revision, source tree, test framework, test commands, and coverage commands>

Reference solution: <reference implementation patch>

Current test suites: <original and previously accepted augmented F2P tests, P2P tests, and executable selectors>

Coverage report: <executable reference-patch lines, per-file F2P and P2P covered-line sets, their union, and baseline union coverage>

Test-surface constraints: <permitted public entry points, internal-symbol restrictions, and newly introduced source symbols>

Identify uncovered executable spans and their EP requirements. For each reachable target, design a public-behavior test with an EP-derived expected outcome. Return the added test patch and executable selectors, together with each candidate's target span, EP clause, entry point, input, expected outcome, and symbol-restriction audit.

Run each candidate before implementation and with the reference solution, then rerun the augmented F2P and P2P suites with the reference solution and collect coverage. Report the observed results, newly covered target lines, and before/after union coverage. Retain only valid, stable F2P tests with verified coverage gains; report remaining gaps and explain any that cannot be exercised through permitted public behavior.

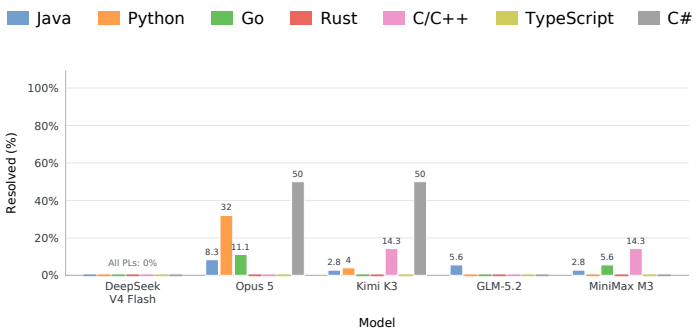
D Effectiveness on Different Programming Languages

Figures 5–8 break down the four effectiveness metrics by programming language across the five scaffolds. Each figure uses the same language colors and reports the models evaluated with each scaffold. Tasks use the project languages listed in Table 5. C and C++ tasks are grouped as C/C++.

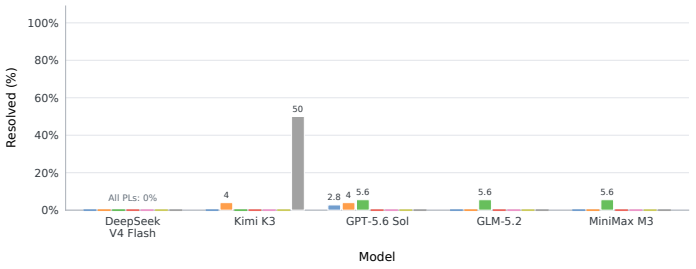
Modular development tasks remain hard for each programming language. Fig. 5 shows that the median resolved rate across the 28 agents is zero in six of the seven language groups. Even the strongest agent resolves only 32.0% of Python tasks, while the best rates for Java, Go, Rust, and C/C++ range from 11.1% to 14.3%, and no agent resolves a TypeScript task. Section-level progress is also limited: completed rates, averaged over the 28 agents within each language group, range from 4.0% to 13.8% across the groups other than C# in Fig. 6. Although some agents resolve all C# tasks, this subset contains only two tasks, and 20 of the 28 agents resolve neither. Therefore, the difficulty of LoLBench extends across programming languages.

Current agents perform better on Python, C/C++, and Rust. This advantage is most evident in Fig. 7. Averaging the language-specific F2P pass rates equally over all 28 agents yields 31.7% for Python, 48.5% for C/C++, and 46.3% for Rust, compared with 6.0% for Java, 13.5% for Go, and 21.7% for TypeScript. This comparison shows that current agents are more capable of handling modular development tasks in Python, C/C++, and Rust, rather than Java, Go, and TypeScript.

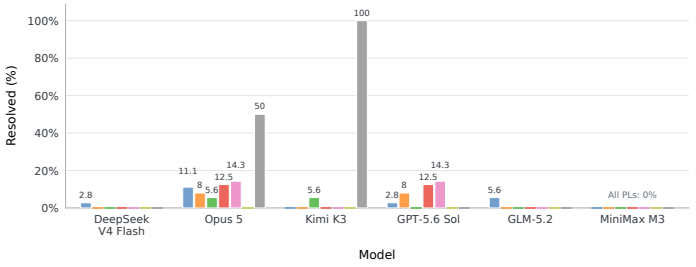
Scaffold rankings vary across different languages. In Figures 5 and 7, we find that with Opus 5, Claude Code achieves a Python resolved rate of 32.0% and an F2P pass rate of 78.5%, but OpenCode achieves only 8.0% and 51.7%, respectively. The ordering reverses for C/C++: OpenCode resolves 14.3% of tasks and passes 65.4% of F2P tests, while Claude Code resolves none and passes 59.4% of F2P tests. Thus, for Opus 5, OpenCode achieves higher resolved and F2P pass rates on C/C++ tasks, while Claude Code achieves higher rates on Python tasks. The same model can therefore benefit from different scaffolds across these tasks with different languages.



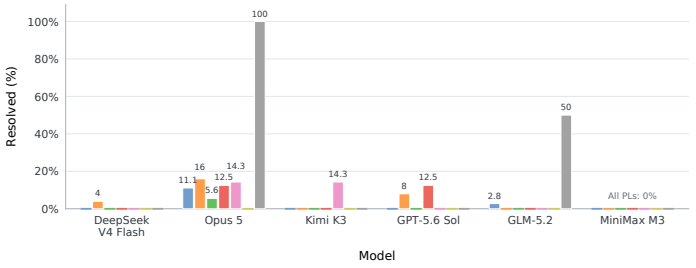
(a) Claude Code



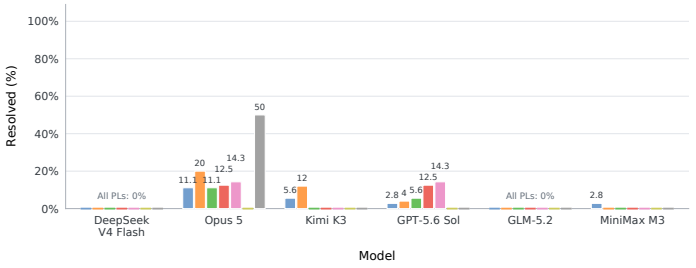
(b) Codex



(c) OpenCode

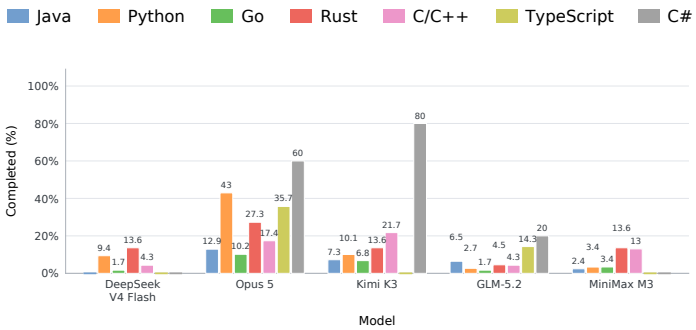


(d) Pi

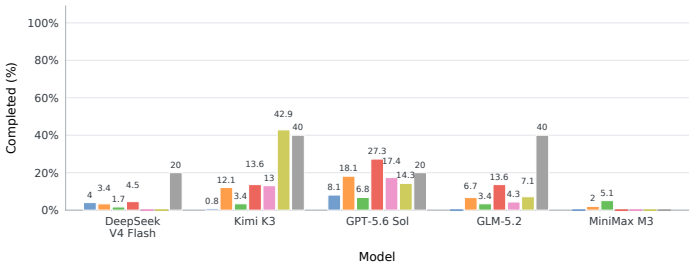


(e) mini-SWE-agent

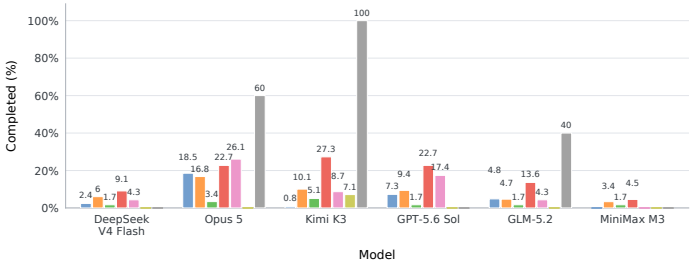
Figure 5: Resolved% by programming language.



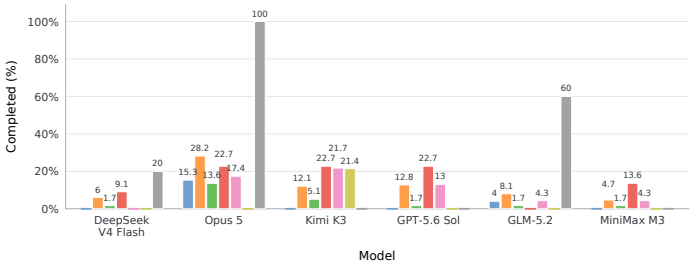
(a) Claude Code



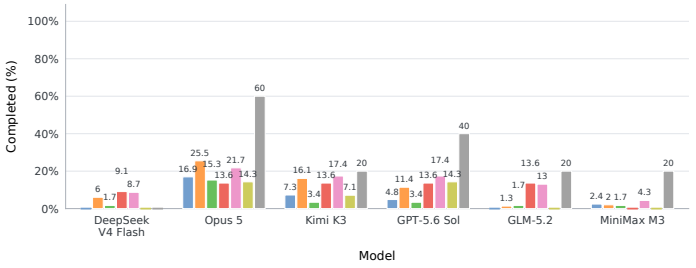
(b) Codex



(c) OpenCode

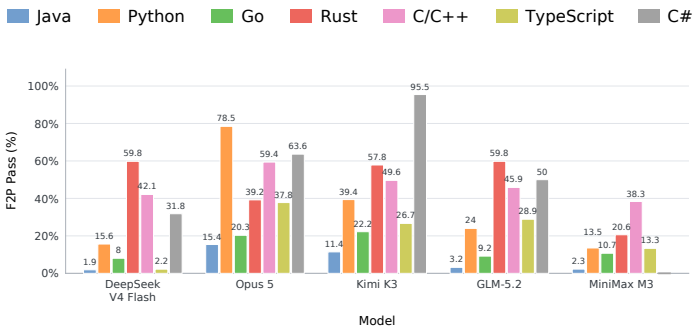


(d) Pi

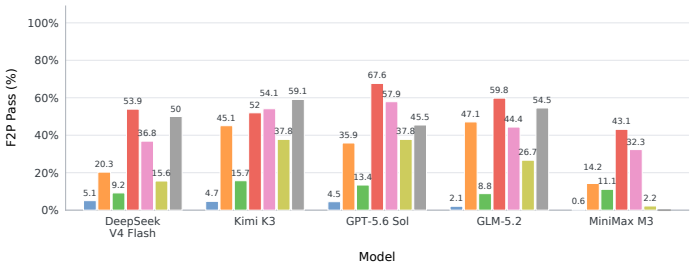


(e) mini-SWE-agent

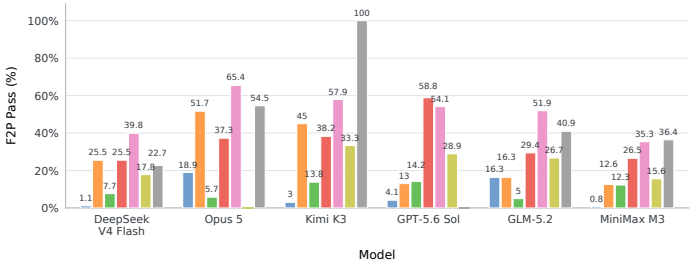
Figure 6: Completed% by programming language.



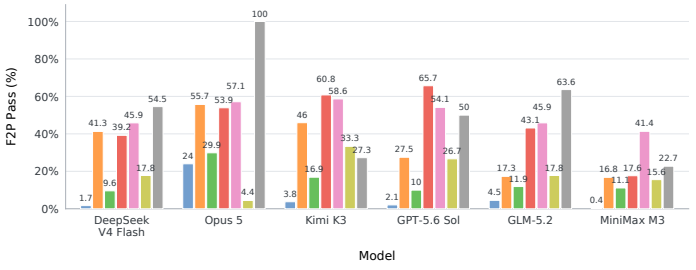
(a) Claude Code



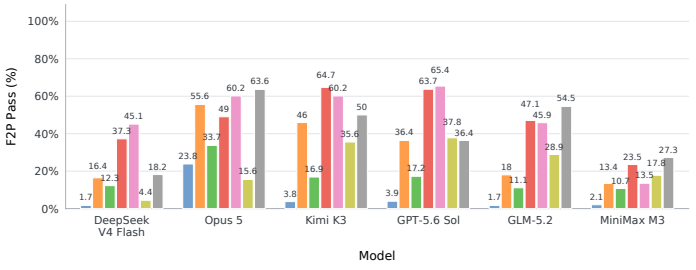
(b) Codex



(c) OpenCode

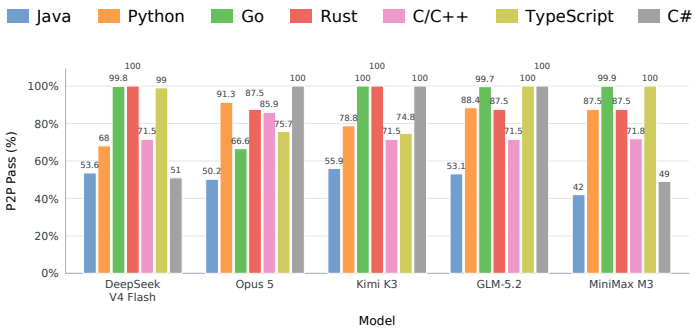


(d) Pi

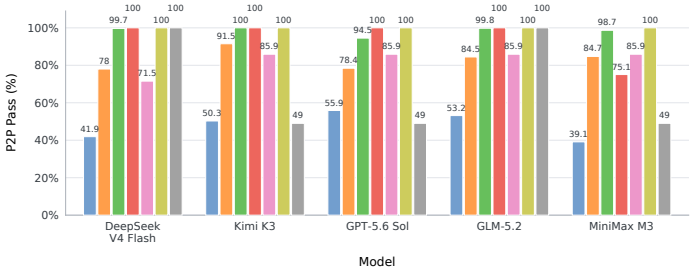


(e) mini-SWE-agent

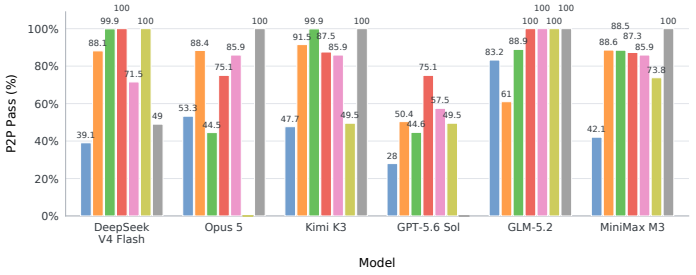
Figure 7: F2P Pass% by programming language.



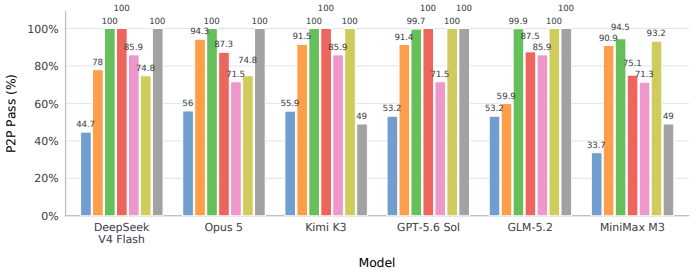
(a) Claude Code



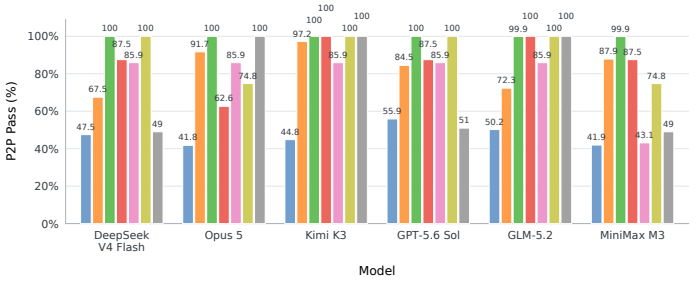
(b) Codex



(c) OpenCode

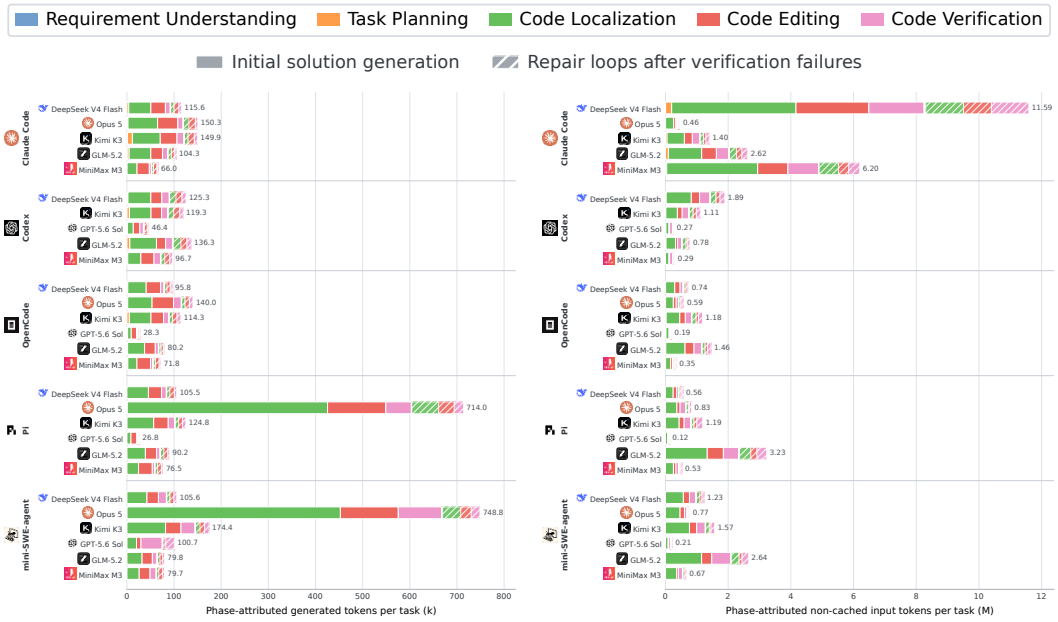


(d) Pi



(e) mini-SWE-agent

Figure 8: P2P Pass% by programming language.



(a) Generated tokens (thousands).

(b) Non-cached input tokens (millions).

Figure 9: Phase-attributed token usage per attempt across 28 agents. Non-cached input excludes cache reads but includes cache writes. Bar labels sum the five colored phases and omit usage without phase attribution. Colors denote phases, solid segments denote initial solution generation, and hatched segments denote repair loops after verification failures.

E Phase Distribution on Token Usage

Current agents consume about 11× as many non-cached input tokens as generated tokens. We report the non-cached input token usage instead of overall input token usage to avoid duplicate counts of input contexts such as the system prompts. Averaging over all attempts with recorded usage, each attempt costs 1.60 million non-cached input tokens and 145.8 thousand generated tokens, yielding a ratio of 11.0×. Computing the ratio of mean non-cached input tokens to mean generated tokens separately for each agent yields a median of 7.9× across the 28 agents. Given the complex nature of tasks in LoLBench, current agents typically spend more tokens understanding and exploring the EP and codebase than editing the code.

Code localization dominates non-cached input and generated token usage. Code localization has the largest macro-averaged shares, accounting for 52.4% of non-cached input tokens and 45.4% of generated tokens in Fig. 9. It is the largest phase in non-cached input token usage for 26 of the 28 agents and in generated token usage for 19. Code editing accounts for smaller macro-averaged ratios of 19.8% and 35.1%, respectively. Within repair loops, code localization remains the largest phase for non-cached input tokens at 39.5%. The overall localization shares reinforce the major efficiency bottleneck identified in Fig. 3.

Code verification is the second-largest phase of non-cached input token usage. Code verification has a macro-averaged share of 26.4% of non-cached input tokens, exceeding code editing at 19.8% in Fig. 9b. It ranks second for 18 of the 28 agents and first for two. Its macro-averaged share of non-cached input usage rises from 23.5% during initial solution generation to 36.6% during repair loops, indicating the frequent need for code verification during revision. This demonstrates the difficulty of modular development tasks, in which current agents have to spend a large token budget on verifying their solutions against the complex dependencies in large software systems.

Table 10: Failure phases, failure modes, and detection methods for the 24 modes observed across the 28 evaluated agents on LoLBench.

Failure Phase	Failure Mode	Description	Detection Method
Requirement Understanding	misread_requirement	Interprets the stated requirement incorrectly.	LLM-based
	missed_constraint	Omits an explicit constraint from the reasoning or solution.	LLM-based
	incomplete_requirement_coverage	Addresses only part of the requested behavior.	LLM-based
	scope_misjudged	Chooses a boundary for the change that is too narrow or too broad.	LLM-based
Task Planning	flawed_plan	Forms a plan that cannot fully satisfy the requirement.	LLM-based
	abandoned_plan	Stops following the stated plan before completing it.	LLM-based
Code Localization	missed_relevant_file	Fails to inspect or modify a file needed for the change.	Pre-defined rules
	wrong_file_localization	Focuses edits on files unrelated to the required change.	Pre-defined rules
	cross_module_context_missing	Misses required context or dependencies across modules.	Pre-defined rules
Code Editing	incorrect_patch	Changes relevant code, but implements the required behavior incorrectly.	Pre-defined rules
	relevant_change_omitted	Leaves out a required change in relevant code.	Pre-defined rules
	no_patch_produced	Completes the attempt without producing a source patch.	Pre-defined rules
	gave_up_early	Stops the attempt before producing an adequate fix.	Pre-defined rules
Code Verification	no_validation_attempted	Runs no recognized validation of the patch.	Pre-defined rules
	no_test_after_final_edit	Makes a final edit after the last test and does not retest.	Pre-defined rules
	authored_test_never_run	Writes or modifies a test but never executes it.	Pre-defined rules
	test_result_misinterpreted	Interprets the test output or status incorrectly.	Pre-defined rules
Self-Repair	repair_not_reverified	Makes a repair after a failure but does not verify it.	Pre-defined rules
	repeated_ineffective_attempt	Repeats repair attempts that do not resolve the failure.	Pre-defined rules
	retry_without_change	Retries a failed action without an intervening change.	Pre-defined rules
	reverted_own_change	Undoes an earlier edit while attempting to repair the solution.	Pre-defined rules
Tool Use	tool_failure_not_retried	Leaves a failed tool action unretried.	Pre-defined rules
	repeated_patch_apply_failure	Repeatedly submits a patch that the tool cannot apply.	Pre-defined rules
	hallucinated_path_unrecovered	Uses a nonexistent path and never recovers from the error.	Pre-defined rules

F Failure Mode Analysis

F.1 Method

Failure taxonomy. The analysis decomposes failures into seven phases and their associated failure modes. Table 10 presents the mappings of failure phases and failure modes. To ensure a stable and mostly objective analysis, we implement predefined rules to detect 18 failure modes. For the 6 remaining failure modes that require semantic understanding, we use an LLM judge.

Objective evidence extraction. We normalize scaffold-specific logs into a common sequence of events such as messages, tool calls, and results. We then analyze the sequences using pre-defined deterministic rules and identify potential evidence. This evidence does not directly establish a failure mode; instead, it is only a suspect candidate pending further analysis. For example, a file in the reference solution that is neither read nor edited provides evidence of a localization miss, whereas a file that is read but left unmodified supports a relevant-change omission in Code Editing. Editing only part of the reference file set supports a cross-module localization failure, and editing all reference files while still failing verification supports an incorrect-patch failure. We collect all evidence identified by pre-defined rules for further analysis.

Semantic judgments. Requirement understanding and task planning failures require semantic understanding, so we assess them with an outcome-blinded LLM judge using a closed set of labels. The judge receives the task and visible process evidence, but neither the reference solution nor the outcome, and must cite supporting trajectory steps. We admit only medium- or high-confidence judgments with valid evidence citations, and we exclude low-confidence and unparseable judgments. We use GPT-5.6 Luna (OpenAI, 2026c) with the pro reasoning mode (OpenRouter id `openai/gpt-5.6-luna-pro`) and temperature 0 to perform the semantic judgment.

Judge prompts. The fixed system prompt below includes the complete label set and response format.

> *You are a rigorous evaluator of an AI software-engineering agent's PROCESS on a bug-fix / feature task. You assess exactly two capabilities:*
>
> - `requirement_understanding`*: did the agent correctly grasp WHAT the issue asks for?*
> - `task_planning`*: did the agent choose a viable APPROACH / sequence to get there?*
>
> *For each capability, choose exactly ONE label from this closed set (or* `none`*):*
> `requirement_understanding`*:*
>
> - `misread_requirement`*: Misunderstood what the issue is actually asking for.*
> - `missed_constraint`*: Overlooked an explicit constraint, edge case, or requirement stated in the issue.*
> - `incomplete_requirement_coverage`*: Understood and addressed only part of a multi-part requirement.*
> - `scope_misjudged`*: Tackled a meaningfully broader or narrower problem than the issue asks.*
> - `none`*: the agent's understanding looks adequate from the evidence.*
>
> `task_planning`*:*
>
> - `flawed_plan`*: Adopted an approach that cannot satisfy the stated requirement.*
> - `abandoned_plan`*: Stated a reasonable plan, then diverged from it without justification.*
> - `none`*: the agent's planning looks adequate from the evidence.*
>
> *CRITICAL RULES:*
>
> 1. *You are NOT told whether the task ultimately passed or failed, and you are NOT shown the reference solution. Judge ONLY the agent's process from the evidence below. Do not try to guess the outcome.*
> 2. *Only flag an error you can support with SPECIFIC step numbers from the evidence. If you cannot point to evidence, answer* `none`*.*

3. *Code* correctness *is judged by a separate system. Do NOT label a misunderstanding just because you suspect the final code is wrong. A different-but-valid approach is NOT an error.*
4. *Some agents' internal reasoning is hidden/encrypted (noted below). When reasoning is hidden, do NOT infer that the agent "failed to consider" something from its absence — judge from visible actions and messages, and lower your confidence.*
5. *Be conservative.* `none` *is the right answer whenever the agent's comprehension/plan looks adequate. Reserve* `low` *confidence for genuinely ambiguous cases.*
6. *The requested scope is the whole TASK REQUIREMENT unless that text explicitly narrows it. If the agent says the requested work is too large/infeasible and therefore plans or submits only a subset, stubs, documentation, or a "minimal" approximation, that is direct evidence of a narrower scope (*`scope_misjudged`*) and usually a plan that cannot satisfy the task (*`flawed_plan`*). Do not excuse deliberate partial implementation merely because the agent describes its resource limits.*
7. *Conversely, ordinary sequencing language such as "first", "focus on X", or "start with X" is not scope reduction when the agent continues toward the remaining requirements.*
8. *The user message may contain a HIGHLIGHTED SCOPE/PLAN RISK section. Classify every highlighted statement as either temporary sequencing or a submission-scope reduction. A coherent, well-tested partial implementation is still a* `flawed_plan` *because it cannot satisfy the full task. Returning* `none` *despite an explicit partial/subset/stub decision is allowed only when later cited evidence shows that the agent rescinded that decision and completed the remaining requested scope. State that disposition in the rationale; "reasonable given time/constraints" is not a valid excuse.*

Respond with a SINGLE JSON object and nothing else, in exactly this shape:

```
{
  "requirement_understanding": {
    "error_type": "<label or none>",
    "confidence": "high|medium|low",
    "evidence_steps": [<int>, ...],
    "rationale": "<=240 chars"
  },
  "task_planning": {
    "error_type": "<label or none>",
    "confidence": "high|medium|low",
    "evidence_steps": [<int>, ...],
    "rationale": "<=240 chars"
  }
}
```

The accompanying user-message template is shown below. Angle-bracketed fields are replaced with the task specification, visible trajectory evidence, and coverage metadata; the step rows are repeated for the selected evidence. The benchmark-contract block, hidden-reasoning note, and highlighted scope/plan statements are included only when applicable.

TASK REQUIREMENT (the work requested):

<task_requirement>

TASK COVERAGE: <task_coverage_mode> — shown <shown_characters> of <total_characters> characters.

BENCHMARK CONTRACT (not the feature specification):

<benchmark_instructions>

AGENT PROCESS EVIDENCE (numbered steps; cite these step numbers):

[<step_id>] <message_or_action_type> <tool_if_any>: <visible_process_text>

NOTE: this agent's internal reasoning is HIDDEN/encrypted. Judge only from the visible messages and actions above, and lower confidence accordingly.

HIGHLIGHTED SCOPE/PLAN RISK STATEMENTS (also present above):

[<risk_step_id>] <message_or_action_type>: <scope_or_plan_statement>

For each highlighted statement, decide whether it is temporary sequencing or a deliberate reduction of the submitted scope. If you return `none`*, cite later evidence that the agent rescinded and completed each reduction.*

PROCESS COVERAGE: <process_coverage_mode> — shown <shown_rows> of <total_eligible_rows> eligible rows. Rows containing explicit scope-reduction language were prioritized. Do not infer a missing plan solely from omitted rows.
Return the JSON verdict now.

Confidence assignment. Each candidate first receives a categorical confidence label: high, medium, or low. For candidates identified by deterministic rules, the triggering predicate specifies the label from the observed evidence. For semantic judgments, we use the confidence label supplied by the single judge response, together with its error label and supporting trajectory steps. The judge is instructed to lower confidence when reasoning is hidden and to use low confidence for ambiguous cases. An error judgment without citations to steps visible in the judge input is downgraded to low confidence before aggregation. Only high- and medium-confidence candidates enter the attribution calculation, with confidence scores of 1.0 and 0.6, respectively. A failed task can exhibit multiple failure modes. For task t, let E_t denote its admitted candidates and c_{ti} the confidence score of candidate i. We normalize these scores within each task to obtain each candidate's fractional attribution w_{ti}:

$$w_{ti} = \frac{c_{ti}}{\sum_{j \in E_t} c_{tj}}, \qquad \sum_{i \in E_t} w_{ti} = 1. \tag{1}$$

Normalization allocates exactly one unit of attribution per failed task, irrespective of the number of detected modes.

Aggregation. Let F_a be the evaluable failed tasks for agent a, and let p_i and m_i denote the phase and failure mode of candidate i. The attribution of failure mode m in phase p, and the total attribution to that phase, are

$$M_{a,p,m} = \sum_{t \in F_a} \sum_{i \in E_t:\, p_i = p,\, m_i = m} w_{ti}, \qquad A_{a,p} = \sum_m M_{a,p,m}. \tag{2}$$

The per-phase failure mode tables report $M_{a,p,m}$, while the phase summary reports $A_{a,p}$. Thus, failure mode attribution values sum to their phase total, and $\sum_p A_{a,p} = |F_a|$. We use largest-remainder rounding to one decimal place, first preserving each agent's total across phases and then each displayed phase total across failure modes. These values are fractional allocations of failed tasks, not counts of tasks with a particular error flag. The resulting decomposition provides a reproducible diagnosis of observable failure patterns.

F.2 Results

Requirement Understanding. Table 11 shows that scope misjudgment receives the largest or joint-largest attribution for 27 agents at the requirement understanding phase. For Claude Code with DeepSeek V4 Flash, it accounts for 8.9 of the phase's 11.2 attribution units. Missed constraints and incomplete requirement coverage are secondary contributors, while direct requirement misreading receives at most 1.0 units per agent. This pattern suggests that the major difficulty is determining the full extent of the requested change, including its constraints and affected functionality.

Task Planning. Table 12 shows that nearly all planning attribution comes from flawed plans. For example, this mode accounts for all 10.5 planning attribution units for Claude Code with DeepSeek V4 Flash and all 10.1 units for Pi with GLM-5.2. Abandoned plans receive at most 0.8 units per agent. These judgments suggest that agents often formulate strategies that cannot fully satisfy the requirements. Effective planning therefore requires checking whether the proposed steps collectively cover the requested behavior and its implementation dependencies.

Code Localization. Missing cross-module context is the largest localization failure mode for all 28 agents in Table 13. It accounts for 41.4 of 41.7 attribution units for Codex with Kimi K3, whereas missed relevant files and wrong-file localization receive at most 2.1 and 0.5 units, respectively, across agents. The dominant pattern is incomplete coverage of the files and modules involved in the reference solutions. This suggests that agents can reach part of the relevant code but struggle to identify the dependencies needed for a complete implementation.

Code Editing. Table 14 identifies incorrect patches as the largest editing contributor for 22 agents and omitted relevant changes as the largest for the remaining six. Both can contribute substantially for a single agent: Pi with GPT-5.6 Sol receives 19.4 units for incorrect patches and 20.5 for omitted

changes, together accounting for all 39.9 editing units. Missing patches and early abandonment contribute comparatively little overall. These results highlight two implementation difficulties: correctly modifying relevant code and carrying identified requirements through to all necessary edits.

Code Verification. The absence of recognized validation is the leading verification failure mode for all 28 agents in Table 15. For mini-SWE-agent with DeepSeek V4 Flash, it accounts for 17.5 of 24.9 attribution units at the verification phase, followed by 6.3 units for missing tests after the final edit. Authored tests that are never executed and misinterpreted test results receive smaller aggregate attribution. The distribution highlights gaps in executing validation and ensuring that its results apply to the final submitted solution.

Self Repair. Table 16 shows that unverified repairs and repeated ineffective attempts dominate repair attribution. Their relative importance varies across agents: repeated ineffective attempts are the largest contributor for all six agents using Pi, while unverified repairs account for 14.6 of 18.2 repair units for mini-SWE-agent with GPT-5.6 Sol. Self-reversions and unchanged retries contribute less overall. These patterns suggest that effective repair requires both revising the implementation strategy in response to failure evidence and verifying whether subsequent changes actually resolve the observed problem.

Tool Use. Tool-use attribution ranges from 0.0 to 1.5 units per agent in Table 17. Unrecovered invalid paths receive the largest aggregate attribution, although other mechanical errors dominate particular agents. Unretried tool failures account for 1.3 of 1.4 units for Pi with Opus 5, while repeated patch-application failures account for 0.9 of 1.5 units for Codex with MiniMax M3. These results show that failures to recover from tool errors contribute relatively little to the overall failure attribution.

G LIMITATIONS

Our study has the following limitations.

LOLBENCH evaluates only functional correctness. LOLBENCH evaluates implementations through F2P tests and P2P regression checks, but does not separately assess readability, maintainability, or consistency with project design conventions. Solutions with similar verifier scores may differ substantially in code duplication, abstraction quality, and ease of future modification. Successful task resolution therefore establishes that an implementation satisfies the evaluated behavioral requirements, while its broader engineering quality remains unassessed. Complementary expert code review would help determine whether such implementations are suitable for long-term maintenance.

Tasks in LOLBENCH are limited to some software systems. High-quality EPs for large software systems are primarily available in well-maintained open-source communities with established proposal and review processes. Relying on these communities limits the diversity of our data sources and the number of eligible proposals. Our tasks therefore cover a limited set of software systems, and the findings may not generalize to projects with different development practices or less formal requirements documentation.

The evaluated models and scaffolds are limited. Computational cost restricts our evaluation to 28 agents built from six models and five scaffolds. This selection covers only part of the possible agents and execution settings. Alternative models and scaffolds may produce different results. We also use one attempt per task and agent, except when provider or infrastructure failures require a retry, limiting our characterization of run-to-run variability. The findings therefore describe the evaluated agents under the specified execution conditions and may not generalize to other agents or resource budgets.

The instructions may omit important context. EPs may rely on assumptions or clarifications recorded in issue threads, design discussions, or review comments. Our tasks provide the curated EP and repository but cannot exhaustively reproduce this surrounding discussion history. Omitted details could introduce ambiguity about expected behavior, compatibility constraints, or design choices. Consequently, some task difficulty may arise from incomplete specification context in addition to the challenge of grounding a proposal in a large codebase.

Table 11: Failure mode distribution for Requirement Understanding failures across the 28 agents. The phase column reproduces the corresponding failure attribution in Table 3 and equals the sum of failure mode values in every row. Darker backgrounds indicate a larger share of total failure attribution for each agent's solution failures.

Scaffold	Model	Phase	Failure Modes			
		Requirement Understanding	Scope Misjudged	Missed Constraint	Incomplete Coverage	Misread Requirement
CLAUDE CODE	DeepSeek V4 Flash	11.2	8.9	1.1	1.0	0.2
	Opus 5	6.9	3.7	1.8	1.4	0.0
	Kimi K3	5.1	2.8	1.2	0.8	0.3
	GLM-5.2	8.2	6.4	0.7	1.1	0.0
	MiniMax M3	9.4	6.6	0.5	1.6	0.7
CODEX	DeepSeek V4 Flash	4.5	2.9	0.7	0.7	0.2
	Kimi K3	2.1	1.0	0.3	0.8	0.0
	GPT-5.6 Sol	1.2	0.2	0.2	0.8	0.0
	GLM-5.2	3.3	2.1	0.0	1.2	0.0
	MiniMax M3	6.3	5.6	0.7	0.0	0.0
OPENCODE	DeepSeek V4 Flash	8.2	6.2	0.8	1.1	0.1
	Opus 5	5.7	3.0	1.7	1.0	0.0
	Kimi K3	5.6	3.2	0.9	1.5	0.0
	GPT-5.6 Sol	0.8	0.6	0.0	0.0	0.2
	GLM-5.2	7.8	6.3	0.5	1.0	0.0
	MiniMax M3	8.3	5.6	1.3	1.1	0.3
PI	DeepSeek V4 Flash	10.2	8.7	0.3	1.0	0.2
	Opus 5	7.1	5.0	1.5	0.6	0.0
	Kimi K3	4.2	2.9	0.3	1.0	0.0
	GPT-5.6 Sol	1.4	0.6	0.5	0.3	0.0
	GLM-5.2	10.4	7.9	1.9	0.6	0.0
	MiniMax M3	10.4	6.1	2.3	1.6	0.4
MINI-SWE-AGENT	DeepSeek V4 Flash	3.0	1.3	1.0	0.4	0.3
	Opus 5	0.6	0.2	0.2	0.2	0.0
	Kimi K3	1.1	1.1	0.0	0.0	0.0
	GPT-5.6 Sol	1.5	1.2	0.1	0.2	0.0
	GLM-5.2	4.1	2.3	1.6	0.2	0.0
	MiniMax M3	7.6	5.6	0.4	0.6	1.0

Failure analysis relies partly on LLM judges. LLM judges infer semantic failure causes from recorded trajectories, and their judgments may depend on the model, prompt, and interpretation of incomplete evidence. Although we combine predefined rules with outcome-blinded judgments, these controls do not completely eliminate potential bias or ambiguity. Different judges may assign the same failure to different phases or failure modes, particularly when several errors co-occur. The reported attribution distributions should therefore be interpreted as diagnostic estimates. Replication across judges and systematic comparison with human annotations would further establish their robustness.

H A TASK EXAMPLE

We reproduce the original instructions for the `cpython_1` task instance, PEP 617 (*New PEG parser for CPython*), including the enhancement proposal as supplied to the agent. We remove a few sections in the EP, such as the test plan and validation sections, since they may disclose the hidden tests.

```
> Implement the requirement described below in the project's source
    tree.
> Put implementation changes in `solution.patch`. If you add tests, put
> them in `test.patch`; tests are optional and must not be included in
> `solution.patch`.
```

Table 12: Failure mode distribution for Task Planning failures across the 28 agents. The phase column reproduces the corresponding failure attribution in Table 3 and equals the sum of failure mode values in every row. Darker backgrounds indicate a larger share of total failure attribution for each agent's solution failures.

Scaffold	Model	Phase	Failure Modes	
		Task Planning	Flawed Plan	Abandoned Plan
Claude Code	DeepSeek V4 Flash	10.5	10.5	0.0
	Opus 5	6.2	6.2	0.0
	Kimi K3	4.4	4.4	0.0
	GLM-5.2	7.7	7.7	0.0
	MiniMax M3	9.2	9.2	0.0
Codex	DeepSeek V4 Flash	3.9	3.9	0.0
	Kimi K3	1.7	1.7	0.0
	GPT-5.6 Sol	1.0	1.0	0.0
	GLM-5.2	2.9	2.9	0.0
	MiniMax M3	6.4	6.4	0.0
OpenCode	DeepSeek V4 Flash	8.5	8.5	0.0
	Opus 5	5.1	5.1	0.0
	Kimi K3	6.1	6.1	0.0
	GPT-5.6 Sol	0.6	0.6	0.0
	GLM-5.2	7.5	7.5	0.0
	MiniMax M3	7.8	7.8	0.0
Pi	DeepSeek V4 Flash	10.4	10.4	0.0
	Opus 5	6.5	6.5	0.0
	Kimi K3	4.3	4.3	0.0
	GPT-5.6 Sol	0.7	0.7	0.0
	GLM-5.2	10.1	10.1	0.0
	MiniMax M3	9.8	9.0	0.8
mini-SWE-agent	DeepSeek V4 Flash	1.7	1.5	0.2
	Opus 5	0.6	0.6	0.0
	Kimi K3	1.1	1.1	0.0
	GPT-5.6 Sol	1.5	1.5	0.0
	GLM-5.2	3.2	3.2	0.0
	MiniMax M3	7.1	6.8	0.3

```
>
> This environment has no outbound internet access -- `curl`/`wget`, `
    git fetch`/`clone`, package installs, and web fetch/search will all
    fail. Implement the requirements using only the code already in the
     workspace and your own knowledge; do not attempt to fetch or
    search external resources.

---

# PEP 617: New PEG parser for CPython

## Overview
This PEP proposes replacing the current LL(1)-based parser of CPython
with a new PEG-based parser. This new parser would allow the
    elimination of multiple
"hacks" that exist in the current grammar to circumvent the LL(1)-
    limitation.
It would substantially reduce the maintenance costs in some areas
    related to the
compiling pipeline such as the grammar, the parser and the AST
    generation. The new PEG
parser will also lift the LL(1) restriction on the current Python
    grammar.
```

Table 13: Failure mode distribution for Code Localization failures across the 28 agents. The phase column reproduces the corresponding failure attribution in Table 3 and equals the sum of failure mode values in every row. Darker backgrounds indicate a larger share of total failure attribution for each agent's solution failures.

Scaffold	Model	Phase	Failure Modes		
		Code Localization	Cross-Module Context Missing	Missed Relevant File	Wrong File Localization
Claude Code	DeepSeek V4 Flash	27.8	26.8	0.7	0.3
	Opus 5	25.1	24.3	0.8	0.0
	Kimi K3	26.4	26.0	0.4	0.0
	GLM-5.2	25.9	23.9	1.6	0.4
	MiniMax M3	24.2	23.4	0.5	0.3
Codex	DeepSeek V4 Flash	39.9	39.3	0.6	0.0
	Kimi K3	41.7	41.4	0.3	0.0
	GPT-5.6 Sol	35.5	35.2	0.3	0.0
	GLM-5.2	42.1	39.6	2.1	0.4
	MiniMax M3	36.1	34.6	1.3	0.2
OpenCode	DeepSeek V4 Flash	25.6	25.0	0.6	0.0
	Opus 5	27.1	25.8	1.3	0.0
	Kimi K3	26.5	26.3	0.2	0.0
	GPT-5.6 Sol	13.2	13.2	0.0	0.0
	GLM-5.2	26.5	24.7	1.5	0.3
	MiniMax M3	25.6	24.2	1.1	0.3
Pi	DeepSeek V4 Flash	32.6	30.7	1.4	0.5
	Opus 5	30.7	30.5	0.2	0.0
	Kimi K3	38.4	37.7	0.7	0.0
	GPT-5.6 Sol	29.5	29.2	0.3	0.0
	GLM-5.2	27.3	26.0	0.9	0.4
	MiniMax M3	24.5	23.7	0.7	0.1
mini-SWE-agent	DeepSeek V4 Flash	38.6	38.0	0.6	0.0
	Opus 5	30.6	30.2	0.4	0.0
	Kimi K3	37.5	37.0	0.5	0.0
	GPT-5.6 Sol	35.9	35.4	0.5	0.0
	GLM-5.2	39.0	37.6	1.2	0.2
	MiniMax M3	31.4	30.6	0.6	0.2

```
## Background on LL(1) parsers
The current Python grammar is an LL(1)-based grammar. A grammar can be
    said to be
LL(1) if it can be parsed by an LL(1) parser, which in turn is defined
    as a
top-down parser that parses the input from left to right, performing
    leftmost
derivation of the sentence, with just one token of lookahead.
The traditional approach to constructing or generating an LL(1) parser
    is to
produce a *parse table* which encodes the possible transitions between
    all possible
states of the parser. These tables are normally constructed from the *
    first sets*
and the *follow sets* of the grammar:

* Given a rule, the *first set* is the collection of all terminals that
     can occur
  first in a full derivation of that rule. Intuitively, this helps the
    parser decide
  among the alternatives in a rule. For
  instance, given the rule:
```

Table 14: Failure mode distribution for Code Editing failures across the 28 agents. The phase column reproduces the corresponding failure attribution in Table 3 and equals the sum of failure mode values in every row. Darker backgrounds indicate a larger share of total failure attribution for each agent's solution failures.

Scaffold	Model	Phase	Failure Modes			
		Code Editing	Incorrect Patch	Relevant Change Omitted	No Patch Produced	Gave Up Early
Claude Code	DeepSeek V4 Flash	25.6	14.6	11.0	0.0	0.0
	Opus 5	18.1	11.0	7.1	0.0	0.0
	Kimi K3	25.9	13.9	11.2	0.8	0.0
	GLM-5.2	31.7	14.5	16.7	0.5	0.0
	MiniMax M3	29.3	13.1	16.2	0.0	0.0
Codex	DeepSeek V4 Flash	24.6	21.5	3.1	0.0	0.0
	Kimi K3	23.8	20.9	2.9	0.0	0.0
	GPT-5.6 Sol	20.5	19.8	0.7	0.0	0.0
	GLM-5.2	25.2	21.1	4.1	0.0	0.0
	MiniMax M3	23.5	20.7	2.8	0.0	0.0
OpenCode	DeepSeek V4 Flash	30.7	15.4	15.3	0.0	0.0
	Opus 5	16.9	9.8	6.6	0.5	0.0
	Kimi K3	25.5	13.0	9.6	1.9	1.0
	GPT-5.6 Sol	18.4	9.9	8.2	0.3	0.0
	GLM-5.2	32.2	14.6	16.6	1.0	0.0
	MiniMax M3	33.0	12.8	20.0	0.2	0.0
Pi	DeepSeek V4 Flash	24.1	16.0	8.1	0.0	0.0
	Opus 5	17.6	12.4	4.2	1.0	0.0
	Kimi K3	28.7	19.2	9.5	0.0	0.0
	GPT-5.6 Sol	39.9	19.4	20.5	0.0	0.0
	GLM-5.2	29.9	15.4	13.9	0.6	0.0
	MiniMax M3	34.2	14.4	17.9	0.9	1.0
mini-SWE-agent	DeepSeek V4 Flash	25.7	21.2	4.5	0.0	0.0
	Opus 5	14.2	11.2	3.0	0.0	0.0
	Kimi K3	18.5	16.6	1.9	0.0	0.0
	GPT-5.6 Sol	24.0	20.2	3.8	0.0	0.0
	GLM-5.2	23.4	20.0	3.4	0.0	0.0
	MiniMax M3	23.6	17.5	5.8	0.3	0.0

````
```
rule: A | B
```
  if only `A` can start with the terminal *a* and only `B` can start
    with the
  terminal *b* and the parser sees the token *b* when parsing this rule
    , it knows
  that it needs to follow the non-terminal `B`.

* An extension to this simple idea is needed when a rule may expand to
    the empty string.
  Given a rule, the *follow set* is the collection of terminals that
    can appear
  immediately to the right of that rule in a partial derivation.
    Intuitively, this
  solves the problem of the empty alternative. For instance,
  given this rule:

```
rule: A 'b'
```
````

Table 15: Failure mode distribution for Code Verification failures across the 28 agents. The phase column reproduces the corresponding failure attribution in Table 3 and equals the sum of failure mode values in every row. Darker backgrounds indicate a larger share of total failure attribution for each agent's solution failures.

Scaffold	Model	Phase	Failure Modes			
		Code Verification	No Validation Attempted	No Test After Final Edit	Authored Test Never Run	Test Result Misinterpreted
Claude Code	DeepSeek V4 Flash	13.8	10.0	2.3	0.7	0.8
	Opus 5	13.8	10.1	2.3	0.9	0.5
	Kimi K3	9.8	4.9	3.3	0.7	0.9
	GLM-5.2	12.9	10.9	1.1	0.3	0.6
	MiniMax M3	12.2	9.9	0.6	1.3	0.4
Codex	DeepSeek V4 Flash	18.3	13.5	4.3	0.3	0.2
	Kimi K3	16.8	10.1	5.0	1.7	0.0
	GPT-5.6 Sol	19.8	16.0	2.1	1.2	0.5
	GLM-5.2	16.3	13.9	2.2	0.0	0.2
	MiniMax M3	15.8	12.0	2.7	0.7	0.4
OpenCode	DeepSeek V4 Flash	12.8	10.3	1.3	0.4	0.8
	Opus 5	9.5	6.2	2.6	0.7	0.0
	Kimi K3	13.7	9.9	2.2	1.0	0.6
	GPT-5.6 Sol	6.5	4.1	1.9	0.1	0.4
	GLM-5.2	12.6	10.7	1.4	0.5	0.0
	MiniMax M3	11.2	9.1	1.5	0.6	0.0
Pi	DeepSeek V4 Flash	10.5	9.1	0.9	0.1	0.4
	Opus 5	8.5	7.5	0.7	0.0	0.3
	Kimi K3	10.7	7.6	1.7	1.0	0.4
	GPT-5.6 Sol	11.7	10.4	1.0	0.0	0.3
	GLM-5.2	10.7	9.7	0.4	0.1	0.5
	MiniMax M3	8.5	7.6	0.0	0.5	0.4
mini-swe-agent	DeepSeek V4 Flash	24.9	17.5	6.3	0.8	0.3
	Opus 5	16.6	11.5	3.4	1.5	0.2
	Kimi K3	23.0	15.5	4.7	1.8	1.0
	GPT-5.6 Sol	13.7	8.5	4.0	0.4	0.8
	GLM-5.2	20.3	16.3	3.3	0.7	0.0
	MiniMax M3	17.0	13.3	2.2	1.3	0.2

```
  if the parser has the token *b* and the non-terminal `A` can only
    start
  with the token *a*, then the parser can tell that this is an invalid
    program.
  But if `A` could expand to the empty string (called an ε-production),
  then the parser would recognise a valid empty `A`,
  since the next token *b* is in the *follow set*  of `A`.

The current Python grammar does not contain ε-productions, so the *
    follow sets* are not
needed when creating the parse tables. Currently, in CPython, a parser
    generator
program reads the grammar and produces a parsing table representing a
    set of
deterministic finite automata (DFA) that can be included in a C program
    , the
parser. The parser is a pushdown automaton that uses this data to
    produce a Concrete
Syntax Tree (CST) sometimes known directly as a "parse tree". In this
    process, the
*first sets* are used indirectly when generating the DFAs.
```

Table 16: Failure mode distribution for Self-Repair failures across the 28 agents. The phase column reproduces the corresponding failure attribution in Table 3 and equals the sum of failure mode values in every row. Darker backgrounds indicate a larger share of total failure attribution for each agent's solution failures.

Scaffold	Model	Phase	Failure Modes			
		Self-Repair	Repair Not Reverified	Repeated Ineffective Attempt	Reverted Own Change	Retry Without Change
CLAUDE CODE	DeepSeek V4 Flash	11.0	5.7	3.9	1.4	0.0
	Opus 5	9.9	6.7	2.5	0.5	0.2
	Kimi K3	18.8	9.1	7.3	2.4	0.0
	GLM-5.2	11.6	5.2	4.2	2.2	0.0
	MiniMax M3	11.5	1.8	5.3	4.3	0.1
CODEX	DeepSeek V4 Flash	8.5	5.9	2.5	0.0	0.1
	Kimi K3	11.4	5.3	5.7	0.4	0.0
	GPT-5.6 Sol	18.0	9.1	7.0	1.2	0.7
	GLM-5.2	9.2	5.9	2.8	0.0	0.5
	MiniMax M3	9.4	2.8	6.3	0.0	0.3
OPENCODE	DeepSeek V4 Flash	11.8	4.4	4.7	2.7	0.0
	Opus 5	14.7	8.5	6.0	0.2	0.0
	Kimi K3	18.1	7.3	7.8	3.0	0.0
	GPT-5.6 Sol	6.5	2.6	3.2	0.7	0.0
	GLM-5.2	11.4	3.5	5.5	2.4	0.0
	MiniMax M3	13.1	3.7	4.7	4.6	0.1
PI	DeepSeek V4 Flash	7.6	1.1	6.0	0.0	0.5
	Opus 5	8.2	0.5	7.4	0.0	0.3
	Kimi K3	11.1	1.3	9.0	0.0	0.8
	GPT-5.6 Sol	11.7	5.0	6.7	0.0	0.0
	GLM-5.2	8.2	1.3	6.9	0.0	0.0
	MiniMax M3	10.4	1.7	7.5	0.0	1.2
MINI-SWE-AGENT	DeepSeek V4 Flash	5.1	2.9	2.0	0.0	0.2
	Opus 5	10.8	9.0	1.8	0.0	0.0
	Kimi K3	11.7	7.6	4.1	0.0	0.0
	GPT-5.6 Sol	18.2	14.6	2.8	0.0	0.8
	GLM-5.2	8.8	5.9	2.7	0.0	0.2
	MiniMax M3	11.2	5.1	5.8	0.0	0.3

````
LL(1) parsers and grammars are usually efficient and simple to
    implement
and generate. However, it is not possible, under the LL(1) restriction,
to express certain common constructs in a way natural to the language
designer and the reader. This includes some in the Python language.

As LL(1) parsers can only look one token ahead to distinguish
possibilities, some rules in the grammar may be ambiguous. For instance
     the rule:

```
rule: A | B
```
is ambiguous if the *first sets* of both `A` and `B` have some elements
     in
common. When the parser sees a token in the input
program that both *A* and *B* can start with, it is impossible for it
    to deduce
which option to expand, as no further token of the program can be
    examined to
disambiguate.
The rule may be transformed to equivalent LL(1) rules, but then it may
be harder for a human reader to grasp its meaning.
````

Table 17: Failure mode distribution for Tool Use failures across the 28 agents. The phase column reproduces the corresponding failure attribution in Table 3 and equals the sum of failure mode values in every row. Darker backgrounds indicate a larger share of total failure attribution for each agent's solution failures.

Scaffold	Model	Phase	Failure Modes		
		Tool Use	Hallucinated Path Unrecovered	Tool Failure Not Retried	Repeated Patch Apply Failure
CLAUDE CODE	DeepSeek V4 Flash	0.1	0.1	0.0	0.0
	Opus 5	0.0	0.0	0.0	0.0
	Kimi K3	0.6	0.1	0.3	0.2
	GLM-5.2	0.0	0.0	0.0	0.0
	MiniMax M3	0.2	0.1	0.1	0.0
CODEX	DeepSeek V4 Flash	0.3	0.3	0.0	0.0
	Kimi K3	0.5	0.5	0.0	0.0
	GPT-5.6 Sol	0.0	0.0	0.0	0.0
	GLM-5.2	0.0	0.0	0.0	0.0
	MiniMax M3	1.5	0.6	0.0	0.9
OPENCODE	DeepSeek V4 Flash	0.4	0.4	0.0	0.0
	Opus 5	0.0	0.0	0.0	0.0
	Kimi K3	0.5	0.4	0.0	0.1
	GPT-5.6 Sol	0.0	0.0	0.0	0.0
	GLM-5.2	0.0	0.0	0.0	0.0
	MiniMax M3	0.0	0.0	0.0	0.0
PI	DeepSeek V4 Flash	0.6	0.2	0.4	0.0
	Opus 5	1.4	0.1	1.3	0.0
	Kimi K3	0.6	0.0	0.6	0.0
	GPT-5.6 Sol	0.1	0.0	0.1	0.0
	GLM-5.2	0.4	0.2	0.2	0.0
	MiniMax M3	1.2	0.6	0.6	0.0
MINI-SWE-AGENT	DeepSeek V4 Flash	0.0	0.0	0.0	0.0
	Opus 5	0.6	0.4	0.0	0.2
	Kimi K3	1.1	1.1	0.0	0.0
	GPT-5.6 Sol	0.2	0.0	0.0	0.2
	GLM-5.2	0.2	0.2	0.0	0.0
	MiniMax M3	1.1	0.4	0.0	0.7

````
Examples later in this document show that the current LL(1)-based
grammar suffers a lot from this scenario.

Another broad class of rules precluded by LL(1) is left-recursive rules
    .
A rule is left-recursive if it can derive to a
sentential form with itself as the leftmost symbol. For instance this
    rule:

```
rule: rule 'a'
```
is left-recursive because the rule can be expanded to an expression
    that starts
with itself. As will be described later, left-recursion is the natural
    way to
express certain desired language properties directly in the grammar.

## Background on PEG parsers
A PEG (Parsing Expression Grammar) grammar differs from a context-free
    grammar
(like the current one) in the fact that the way it is written more
    closely
````

```
reflects how the parser will operate when parsing it. The fundamental
    technical
difference is that the choice operator is ordered. This means that when
     writing:

```
rule: A | B | C
```
a context-free-grammar parser (like an LL(1) parser) will generate
    constructions
that given an input string will *deduce* which alternative (`A`, `B` or
     `C`)
must be expanded, while a PEG parser will check if the first
    alternative succeeds
and only if it fails, will it continue with the second or the third one
     in the
order in which they are written. This makes the choice operator not
    commutative.

Unlike LL(1) parsers, PEG-based parsers cannot be ambiguous: if a
    string parses,
it has exactly one valid parse tree. This means that a PEG-based parser
     cannot
suffer from the ambiguity problems described in the previous section.

PEG parsers are usually constructed as a recursive descent parser in
    which every
rule in the grammar corresponds to a function in the program
    implementing the
parser and the parsing expression (the "expansion" or "definition" of
    the rule)
represents the "code" in said function. Each parsing function
    conceptually takes
an input string as its argument, and yields one of the following
    results:

* A "success" result. This result indicates that the expression can be
    parsed by
  that rule and the function may optionally move forward or consume one
     or more
  characters of the input string supplied to it.
* A "failure" result, in which case no input is consumed.

Notice that "failure" results do not imply that the program is
    incorrect or a
parsing failure because as the choice operator is ordered, a "failure"
    result
merely indicates "try the following option". A direct implementation of
     a PEG
parser as a recursive descent parser will present exponential time
    performance in
the worst case as compared with LL(1) parsers, because PEG parsers have
     infinite lookahead
(this means that they can consider an arbitrary number of tokens before
     deciding
for a rule). Usually, PEG parsers avoid this exponential time
    complexity with a
technique called "packrat parsing" [1]_ which not only loads the entire
program in memory before parsing it but also allows the parser to
    backtrack
arbitrarily. This is made efficient by memoizing the rules already
    matched for
each position. The cost of the memoization cache is that the parser
    will naturally
```

````
use more memory than a simple LL(1) parser, which normally are table-
    based. We
will explain later in this document why we consider this cost
    acceptable.

## Rationale
In this section, we describe a list of problems that are present in the
     current parser
machinery in CPython that motivates the need for a new parser.

### Some rules are not actually LL(1)
Although the Python grammar is technically an LL(1) grammar (because it
     is parsed by
an LL(1) parser) several rules are not LL(1) and several workarounds
    are
implemented in the grammar and in other parts of CPython to deal with
    this. For
example, consider the rule for assignment expressions:

```
namedexpr_test: [NAME ':='] test
```
This simple rule is not compatible with the Python grammar as *NAME* is
     among the
elements of the *first set* of the rule *test*. To work around this
    limitation the
actual rule that appears in the current grammar is:

```
namedexpr_test: test [':=' test]
```
Which is a much broader rule than the previous one allowing constructs
    like ``[x
for x in y] := [1,2,3]``. The way the rule is limited to its desired
    form is by
disallowing these unwanted constructions when transforming the parse
    tree to the
abstract syntax tree. This is not only inelegant but a considerable
    maintenance
burden as it forces the AST creation routines and the compiler into a
    situation in
which they need to know how to separate valid programs from invalid
    programs,
which should be a responsibility solely of the parser. This also leads
    to the
actual grammar file not reflecting correctly what the *actual* grammar
    is (that
is, the collection of all valid Python programs).

Similar workarounds appear in multiple other rules of the current
    grammar.
Sometimes this problem is unsolvable. For instance, [bpo-12782:
    Multiple context expressions do not support parentheses for
    continuation across lines](https://github.com/python/cpython/issues
    /56991) shows how making an LL(1) rule that supports
writing:

```
with (
    open("a_really_long_foo") as foo,
    open("a_really_long_baz") as baz,
    open("a_really_long_bar") as bar
):
  ...
```
````

````
is not possible since the first sets of the grammar items that can
appear as context managers include the open parenthesis, making the
    rule
ambiguous. This rule is not only consistent with other parts of the
    language (like
the rule for multiple imports), but is also very useful to auto-
    formatting tools,
as parenthesized groups are normally used to group elements to be
formatted together (in the same way the tools operate on the contents
    of lists,
sets...).

### Complicated AST parsing
Another problem of the current parser is that there is a huge coupling
    between the
AST generation routines and the particular shape of the produced parse
    trees. This
makes the code for generating the AST especially complicated as many
    actions and
choices are implicit. For instance, the AST generation code knows what
alternatives of a certain rule are produced based on the number of
    child nodes
present in a given parse node. This makes the code difficult to follow
    as this
property is not directly related to the grammar file and is influenced
    by
implementation details. As a result of this, a considerable amount of
    the AST
generation code needs to deal with inspecting and reasoning about the
    particular
shape of the parse trees that it receives.

### Lack of left recursion
As described previously, a limitation of LL(1) grammars is that they
    cannot allow
left-recursion. This makes writing some rules very unnatural and far
    from how
programmers normally think about the program. For instance this
    construct (a simpler
variation of several rules present in the current grammar):

```
expr: expr '+' term | term
```
cannot be parsed by an LL(1) parser. The traditional remedy is to
    rewrite the
grammar to circumvent the problem:

```
expr: term ('+' term)*
```
The problem that appears with this form is that the parse tree is
    forced to have a
very unnatural shape. This is because with this rule, for the input
    program ``a +
b + c` the parse tree will be flattened (`['a', '+', 'b', '+', 'c']``)
    and must
be post-processed to construct a left-recursive parse tree (``[['a',
    '+', 'b'],
'+', 'c']``). Being forced to write the second rule not only leads to
    the parse
tree not correctly reflecting the desired associativity, but also
    imposes further
pressure on later compilation stages to detect and post-process these
    cases.
````

```
### Intermediate parse tree
The last problem present in the current parser is the intermediate
    creation of a
parse tree or Concrete Syntax Tree that is later transformed to an
    Abstract Syntax
Tree. Although the construction of a CST is very common in parser and
    compiler
pipelines, in CPython this intermediate CST is not used by anything
    else (it is
only indirectly exposed by the *parser* module and a surprisingly small
     part of
the code in the CST production is reused in the module). Which is worse
    : the whole
tree is kept in memory, keeping many branches that consist of chains of
     nodes with
a single child. This has been shown to consume a considerable amount of
     memory (for
instance in [bpo-26415: Excessive peak memory consumption by the Python
     parser](https://github.com/python/cpython/issues/70603)).

Having to produce an intermediate result between the grammar and the
    AST is not only
undesirable but also makes the AST generation step much more
    complicated, raising
considerably the maintenance burden.

## The new proposed PEG parser
The new proposed PEG parser contains the following pieces:

* A parser generator that can read a grammar file and produce a PEG
    parser
  written in Python or C that can parse said grammar.

* A PEG meta-grammar that automatically generates a Python parser that
    is used
  for the parser generator itself (this means that there are no
    manually-written
  parsers).

* A generated parser (using the parser generator) that can directly
    produce C and
  Python AST objects.

On the implementation side, the Python parser generator exposes these
    steps as `parse_string`, which runs a parser class over grammar
    source text; `generate_parser`, which turns a parsed `Grammar` into
    a parser class; and `make_parser`, which composes the two to build
    a parser directly from grammar source.
Expose the parser-generator package under `Tools/peg_generator/pegen/`;
     in
particular, provide `pegen.grammar_parser.GeneratedParser` (commonly
    used as
`GrammarParser`), `pegen.testutil.parse_string`,
`pegen.testutil.generate_parser`, `pegen.testutil.make_parser`,
`pegen.testutil.generate_parser_c_extension`,
`pegen.testutil.generate_c_parser_source`, `pegen.grammar.Grammar`,
`pegen.grammar.GrammarError`, `pegen.grammar.GrammarVisitor`, and
`pegen.first_sets.FirstSetCalculator`.

### Left recursion
PEG parsers normally do not support left recursion but we have
    implemented a
technique similar to the one described in Medeiros et al. [2]_ but
    using the
```

````
memoization cache instead of static variables. This approach is closer
    to the one
described in Warth et al. [3]_. This allows us to write not only simple
     left-recursive
rules but also more complicated rules that involve indirect left-
    recursion like:

```
rule1: rule2 | 'a'
rule2: rule3 | 'b'
rule3: rule1 | 'c'
```
and "hidden left-recursion" like:

```
rule: 'optional'? rule '@' some_other_rule
```
### Syntax
The grammar consists of a sequence of rules of the form:

```
rule_name: expression
```
Optionally, a type can be included right after the rule name, which
specifies the return type of the C or Python function corresponding to
the rule:

```
rule_name[return_type]: expression
```
If the return type is omitted, then a `void *` is returned in C and an
`Any` in Python.

#### Grammar Expressions
`# comment`
''''''''''''

Python-style comments.

`e1 e2`
'''''''''

Match e1, then match e2.

```PEG
rule_name: first_rule second_rule
```
`e1 | e2`
''''''''''

Match e1 or e2.

The first alternative can also appear on the line after the rule name
for formatting purposes. In that case, a \| must be used before the
first alternative, like so:

```PEG
rule_name[return_type]:
    | first_alt
    | second_alt
```
`( e )`
'''''''''

Match e.
````

````
```PEG
rule_name: (e)
```
A slightly more complex and useful example includes using the grouping
operator together with the repeat operators:

```PEG
rule_name: (e1 e2)*
```
`[ e ] or e?`
''''''''''''''

Optionally match e.

```PEG
rule_name: [e]
```
A more useful example includes defining that a trailing comma is
optional:

```PEG
rule_name: e (',' e)* [',']
```
`e*`
''''''

Match zero or more occurrences of e.

```PEG
rule_name: (e1 e2)*
```
`e+`
''''''

Match one or more occurrences of e.

```PEG
rule_name: (e1 e2)+
```
`s.e+`
''''''''

Match one or more occurrences of e, separated by s. The generated parse
tree does not include the separator. This is otherwise identical to
`(e (s e)*)`.

```PEG
rule_name: ','.e+
```
`&e`
''''''

Succeed if e can be parsed, without consuming any input.

`!e`
''''''

Fail if e can be parsed, without consuming any input.

An example taken from the proposed Python grammar specifies that a
    primary
consists of an atom, which is not followed by a `.` or a `(` or a
`[`:
````

```
```PEG
primary: atom !'.' !'(' !'['
```
`~`
''''''

Commit to the current alternative, even if it fails to parse.

```PEG
rule_name: '(' ~ some_rule ')' | some_alt
```
In this example, if a left parenthesis is parsed, then the other
alternative won't be considered, even if some_rule or `)' fail to be
parsed.

Variables in the Grammar
A subexpression can be named by preceding it with an identifier and an
`=` sign. The name can then be used in the action (see below), like
    this:

```
rule_name[return_type]: '(' a=some_other_rule ')' { a }
```
Grammar actions
To avoid the intermediate steps that obscure the relationship between
    the
grammar and the AST generation the proposed PEG parser allows directly
generating AST nodes for a rule via grammar actions. Grammar actions
    are
language-specific expressions that are evaluated when a grammar rule is
successfully parsed. These expressions can be written in Python or C
depending on the desired output of the parser generator. This means
    that if
one would want to generate a parser in Python and another in C, two
    grammar
files should be written, each one with a different set of actions,
    keeping
everything else apart from said actions identical in both files. As an
example of a grammar with Python actions, the piece of the parser
    generator
that parses grammar files is bootstrapped from a meta-grammar file with
Python actions that generate the grammar tree as a result of the
    parsing.

In the specific case of the new proposed PEG grammar for Python, having
actions allows directly describing how the AST is composed in the
    grammar
itself, making it more clear and maintainable. This AST generation
    process is
supported by the use of some helper functions that factor out common
    AST
object manipulations and some other required operations that are not
    directly
related to the grammar.

To indicate these actions each alternative can be followed by the
    action code
inside curly-braces, which specifies the return value of the
    alternative:

```
rule_name[return_type]:
    | first_alt1 first_alt2 { first_alt1 }
    | second_alt1 second_alt2 { second_alt1 }
```
```

```
If the action is omitted and C code is being generated, then there are
    two
different possibilities:

1. If there's a single name in the alternative, this gets returned.
2. If not, a dummy name object gets returned (this case should be
    avoided).

If the action is omitted and Python code is being generated, then a
    list
with all the parsed expressions gets returned (this is meant for
    debugging).

The full meta-grammar for the grammars supported by the PEG generator
    is:

```PEG
start[Grammar]: grammar ENDMARKER { grammar }

grammar[Grammar]:
    | metas rules { Grammar(rules, metas) }
    | rules { Grammar(rules, []) }

metas[MetaList]:
    | meta metas { [meta] + metas }
    | meta { [meta] }

meta[MetaTuple]:
    | "@" NAME NEWLINE { (name.string, None) }
    | "@" a=NAME b=NAME NEWLINE { (a.string, b.string) }
    | "@" NAME STRING NEWLINE { (name.string, literal_eval(string.
    string)) }

rules[RuleList]:
    | rule rules { [rule] + rules }
    | rule { [rule] }

rule[Rule]:
    | rulename ":" alts NEWLINE INDENT more_alts DEDENT {
          Rule(rulename[0], rulename[1], Rhs(alts.alts + more_alts.alts
    )) }
    | rulename ":" NEWLINE INDENT more_alts DEDENT { Rule(rulename[0],
    rulename[1], more_alts) }
    | rulename ":" alts NEWLINE { Rule(rulename[0], rulename[1], alts)
    }

rulename[RuleName]:
    | NAME '[' type=NAME '*' ']' {(name.string, type.string+"*")}
    | NAME '[' type=NAME ']' {(name.string, type.string)}
    | NAME {(name.string, None)}

alts[Rhs]:
    | alt "|" alts { Rhs([alt] + alts.alts)}
    | alt { Rhs([alt]) }

more_alts[Rhs]:
    | "|" alts NEWLINE more_alts { Rhs(alts.alts + more_alts.alts) }
    | "|" alts NEWLINE { Rhs(alts.alts) }

alt[Alt]:
    | items '$' action { Alt(items + [NamedItem(None, NameLeaf('
    ENDMARKER'))], action=action) }
    | items '$' { Alt(items + [NamedItem(None, NameLeaf('ENDMARKER'))],
     action=None) }
    | items action { Alt(items, action=action) }
```
```

```
    | items { Alt(items, action=None) }

items[NamedItemList]:
    | named_item items { [named_item] + items }
    | named_item { [named_item] }

named_item[NamedItem]:
    | NAME '=' ~ item {NamedItem(name.string, item)}
    | item {NamedItem(None, item)}
    | it=lookahead {NamedItem(None, it)}

lookahead[LookaheadOrCut]:
    | '&' ~ atom {PositiveLookahead(atom)}
    | '!' ~ atom {NegativeLookahead(atom)}
    | '~' {Cut()}

item[Item]:
    | '[' ~ alts ']' {Opt(alts)}
    |  atom '?' {Opt(atom)}
    |  atom '*' {Repeat0(atom)}
    |  atom '+' {Repeat1(atom)}
    |  sep=atom '.' node=atom '+' {Gather(sep, node)}
    |  atom {atom}

atom[Plain]:
    | '(' ~ alts ')' {Group(alts)}
    | NAME {NameLeaf(name.string) }
    | STRING {StringLeaf(string.string)}

## Mini-grammar for the actions

action[str]: "{" ~ target_atoms "}" { target_atoms }

target_atoms[str]:
    | target_atom target_atoms { target_atom + " " + target_atoms }
    | target_atom { target_atom }

target_atom[str]:
    | "{" ~ target_atoms "}" { "{" + target_atoms + "}" }
    | NAME { name.string }
    | NUMBER { number.string }
    | STRING { string.string }
    | "?" { "?" }
    | ":" { ":" }
```

As an illustrative example this simple grammar file allows directly generating a full parser that can parse simple arithmetic expressions and that returns a valid C-based Python AST:

```PEG
start[mod_ty]: a=expr_stmt* $ { Module(a, NULL, p->arena) }
expr_stmt[stmt_ty]: a=expr NEWLINE { _Py_Expr(a, EXTRA) }
expr[expr_ty]:
    | l=expr '+' r=term { _Py_BinOp(l, Add, r, EXTRA) }
    | l=expr '-' r=term { _Py_BinOp(l, Sub, r, EXTRA) }
    | t=term { t }

term[expr_ty]:
    | l=term '*' r=factor { _Py_BinOp(l, Mult, r, EXTRA) }
    | l=term '/' r=factor { _Py_BinOp(l, Div, r, EXTRA) }
    | f=factor { f }

factor[expr_ty]:
    | '(' e=expr ')' { e }
```

```
    | a=atom { a }

atom[expr_ty]:
    | n=NAME { n }
    | n=NUMBER { n }
    | s=STRING { s }
```
Here `EXTRA` is a macro that expands to ``start_lineno,
    start_col_offset,
end_lineno, end_col_offset, p->arena``, those being variables
    automatically
injected by the parser; `p` points to an object that holds on to all
    state
for the parser.

A similar grammar written to target Python AST objects:

```PEG
start: expr NEWLINE? ENDMARKER { ast.Expression(expr) }
expr:
    | expr '+' term { ast.BinOp(expr, ast.Add(), term) }
    | expr '-' term { ast.BinOp(expr, ast.Sub(), term) }
    | term { term }

term:
    | l=term '*' r=factor { ast.BinOp(l, ast.Mult(), r) }
    | term '/' factor { ast.BinOp(term, ast.Div(), factor) }
    | factor { factor }

factor:
    | '(' expr ')' { expr }
    | atom { atom }

atom:
    | NAME { ast.Name(id=name.string, ctx=ast.Load()) }
    | NUMBER { ast.Constant(value=ast.literal_eval(number.string)) }
```
Migration plan
This section describes the migration plan when porting to the new PEG-
    based parser
if this PEP is accepted. The migration will be executed in a series of
    steps that allow
initially to fallback to the previous parser if needed:

1. Starting with Python 3.9 alpha 6, include the new PEG-based parser
    machinery in CPython
    with a command-line flag and environment variable that allows
    switching between
    the new and the old parsers together with explicit APIs that allow
    invoking the
    new and the old parsers independently. At this step, all Python
    APIs like `ast.parse`
    and `compile` will use the parser set by the flags or the
    environment variable and
    the default parser will be the new PEG-based parser. Which parser
    is active is
    observable at runtime through `sys.flags.use_peg`, a new member of
    the `sys.flags`
    struct sequence that is nonzero while the PEG parser is in effect.

2. Between Python 3.9 and Python 3.10, the old parser and related code
     (like the
    "parser" module) will be kept until a new Python release happens (
    Python 3.10). In

```
    the meanwhile and until the old parser is removed, **no new Python
    Grammar
    addition will be added that requires the PEG parser**. This means
    that the grammar
    will be kept LL(1) until the old parser is removed.

3.  In Python 3.10, remove the old parser, the command-line flag, the
    environment
    variable and the "parser" module and related code.

## Performance and validation
We have done extensive timing and validation of the new parser, and
this gives us confidence that the new parser is of high enough quality
to replace the current parser.

### Performance
We have tuned the performance of the new parser to come within 10% of
the current parser both in speed and memory consumption. While the
PEG/packrat parsing algorithm inherently consumes more memory than the
current LL(1) parser, we have an advantage because we don't construct
an intermediate CST.

Below are some benchmarks. These are focused on compiling source code
to bytecode, because this is the most realistic situation. Returning
an AST to Python code is not as representative, because the process to
convert the *internal* AST (only accessible to C code) to an
*external* AST (an instance of `ast.AST`) takes more time than the
parser itself.

All measurements reported here are done on a recent MacBook Pro,
taking the median of three runs. No particular care was taken to stop
other applications running on the same machine.

The first timings are for our canonical test file, which has 100,000
lines endlessly repeating the following three lines:

```python
1 + 2 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + ((((((11 * 12 * 13 * 14 * 15 + 16
    * 17 + 18 * 19 * 20))))))
2*3 + 4*5*6
12 + (2 * 3 * 4 * 5 + 6 + 7 * 8)
```
- Just parsing and throwing away the internal AST takes 1.16 seconds
  with a max RSS of 681 MiB.

- Parsing and converting to `ast.AST` takes 6.34 seconds, max RSS
  1029 MiB.

- Parsing and compiling to bytecode takes 1.28 seconds, max RSS 681
  MiB.

- With the current parser, parsing and compiling takes 1.44 seconds,
  max RSS 836 MiB.

For this particular test file, the new parser is faster and uses less
memory than the current parser (compare the last two bullets).

We also did timings with a more realistic payload, the entire Python
3.8 stdlib. This payload consists of 1,641 files, 749,570 lines,
27,622,497 bytes. (Though 11 files can't be compiled by any Python 3
parser due to encoding issues, sometimes intentional.)

- Compiling and throwing away the internal AST took 2.141 seconds.
  That's 350,040 lines/sec, or 12,899,367 bytes/sec. The max RSS was
  74 MiB (the largest file in the stdlib is much smaller than our
```

```
  canonical test file).

- Compiling to bytecode took 3.290 seconds. That's 227,861 lines/sec,
  or 8,396,942 bytes/sec. Max RSS 77 MiB.

- Compiling to bytecode using the current parser took 3.367 seconds.
  That's 222,620 lines/sec, or 8,203,780 bytes/sec. Max RSS 70 MiB.

Comparing the last two bullets we find that the new parser is slightly
faster but uses slightly (about 10%) more memory. We believe this is
acceptable. (Also, there are probably some more tweaks we can make to
reduce memory usage.)

## Rejected Alternatives
We did not seriously consider alternative ways to implement the new
parser, but here's a brief discussion of LALR(1).

Thirty years ago the first author decided to go his own way with
Python's parser rather than using LALR(1), which was the industry
standard at the time (e.g. Bison and Yacc).  The reasons were
primarily emotional (gut feelings, intuition), based on past experience
using Yacc in other projects, where grammar development took more
effort than anticipated (in part due to shift-reduce conflicts).  A
specific criticism of Bison and Yacc that still holds is that their
meta-grammar (the notation used to feed the grammar into the parser
generator) does not support EBNF conveniences like
`[optional_clause]` or `(repeated_clause)*`.  Using a custom
parser generator, a syntax tree matching the structure of the grammar
could be generated automatically, and with EBNF that tree could match
the "human-friendly" structure of the grammar.

Other variants of LR were not considered, nor was LL (e.g. ANTLR).
PEG was selected because it was easy to understand given a basic
understanding of recursive-descent parsing.
```