SWE-CRAFT: GRAPH-GUIDED TASK SYNTHESIS FOR WEAK SPOTS IN CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Executable software engineering (SWE) tasks provide controlled environments for studying and improving coding agents. Existing perturbation-based synthesis scales task generation but offers limited control over the engineering practice represented in the resulting data. We study target-conditioned SWE task synthesis: a supplied capability target specifies desired model behavior or reasoning and guides task design before construction. In our experiments, model-assisted analysis and human synthesis of base-model trajectories motivate these targets. We introduce SWE-CRAFT, which uses a source-linked code property graph to explore functional modules across functions and files. A task design agent matches repository behavior to a capability target and specifies required and preserved behaviors. The task specification then guides separate construction and validation of executable bug-fixing, feature-addition, and feature-enhancement tasks. Quality-screened teacher trajectories provide supervision for fine-tuning. In a source-blinded LLM audit, SWE-CRAFT scores 93.16 out of 100 for alignment with a shared task profile, compared with 53.36 for SWE-smith and 42.67 for BugPilot. Fine-tuning Ling-3.0-flash on 1,500 trajectories achieves 46.88% and 68.00% Pass@1 on SWE-bench Pro and SWE-bench Multilingual, outperforming SWE-smith by 2.74 and 3.55 percentage points, and BugPilot by 2.05 and 3.00 points, respectively[1]

1 INTRODUCTION

Large language model (LLM)-powered coding agents, such as Claude Code (Anthropic, 2025) and SWE-agent (Yang et al., 2024), are increasingly used to automate repository-level software engineering (SWE) tasks. Benchmarks such as SWE-bench (Jimenez et al., 2024) and DeepSWE (Huang et al., 2026) evaluate this setting by requiring agents to inspect codebases, make repository-level code changes, and verify them through execution. A common way to improve coding agents is to train them on data derived from executable SWE tasks (Pan et al., 2025; Yang et al., 2025). The choice of tasks therefore shapes the engineering problems and solution strategies represented in training data.

To supply such tasks at scale, existing synthesis methods modify runnable repositories through code mutations, function rewrites, or injected bugs (Yang et al., 2025; Pham et al., 2026; Sonwane et al., 2025). For example, SWE-smith (Yang et al., 2025) generates candidate patches, retains those that break passing tests, and derives problem statements from the resulting changes and failures. These methods scale executable task generation, but offer limited control over what generated tasks are meant to exercise: the resulting practice is largely determined by the perturbations that succeed.

This limitation becomes particularly important when a coding agent exhibits recurring weak spots, or patterns of difficulty inferred from its trajectories. For example, an agent may update a configuration option in one component while overlooking another consumer. Rather than generating tasks whose demands are determined by whichever perturbations succeed, we would like to construct tasks that deliberately exercise the abilities associated with such weaknesses. We express these desired abilities as capability targets, such as maintaining consistency across interacting components. In this formulation, a weak spot describes an observed difficulty, while a capability target specifies the behavior or reasoning ability that training tasks should exercise.

[1]Experimental materials are available at: https://anonymous.4open.science/r/SWE-Craft.

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.

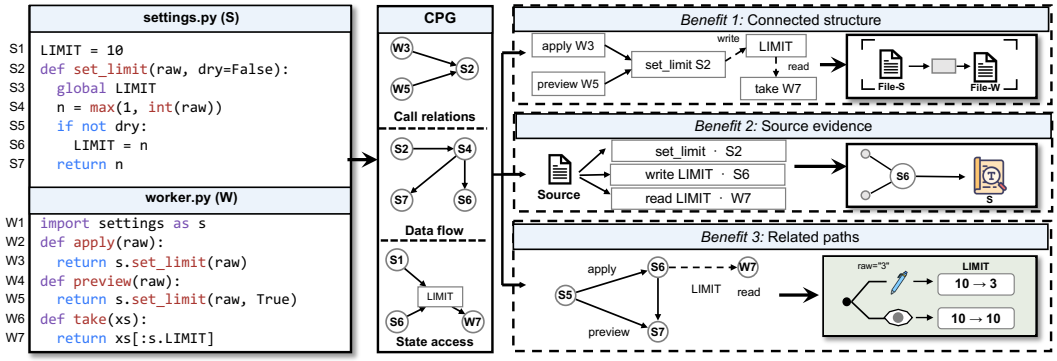


Figure 1: A running example to show how a CPG guides code repository understanding.

This motivates the problem we study: target-conditioned SWE task synthesis, in which a supplied capability target guides the construction of executable repository tasks that exercise the specified ability. A key challenge in target-conditioned task synthesis is locating repository context that can realize a given capability target. Relevant behavior is often distributed across multiple functions and files whose interactions must be understood together. Local exploration may recover individual components while missing their dependencies, leading to incomplete or unsupported task designs. We therefore need a connected, source-grounded view of the relevant code.

We obtain this view with a source-linked code property graph (CPG), which represents program constructs as nodes and structural relations such as calls, control flow, data flow, and state access as typed edges (Yamaguchi et al., 2014). Figure 1 illustrates how organizing code in this way supports repository understanding. By following calls and shared-state relations, the agent can connect `set_limit` with its callers and the cross-file consumer `take`, identifying a functional module whose components jointly implement a behavior. Because these connections link back to source locations, the agent can inspect the code supporting each dependency and check its interpretation against source evidence. Within this module, control flow keeps both `apply` and `preview` paths visible, while their linked code explains why only `apply` updates `LIMIT`. With this understanding, SWE-CRAFT matches functional modules to capability targets and develops task specifications. Task Construction and Validation then turns these specifications into executable tasks for teacher trajectory collection and supervised fine-tuning.

We introduce SWE-CRAFT, a graph-guided framework for target-conditioned SWE task synthesis. Given a capability target, we identify functional modules that can exercise the specified abilities and derive a task specification from source code and tests. This specification defines repository-specific required and preserved behaviors and is fixed before construction. With a shared repository pool, SWE-CRAFT outperforms SWE-smith by 2.74 and 3.55 percentage points and BugPilot by 2.05 and 3.00 points in Pass@1 on SWE-bench Pro and SWE-bench Multilingual.

- We formulate *target-conditioned SWE task synthesis*, using explicit capability targets to guide what the generated tasks should exercise.
- We introduce SWE-CRAFT, which grounds capability targets in connected repository behavior and preserves the resulting task specifications through construction and validation.
- Under shared repository and training settings, we show stronger task-profile alignment and downstream Pass@1 than the compared synthesis baselines.

2 BACKGROUND

2.1 EXECUTABLE REPOSITORY TASKS

We follow the SWE-bench task format, in which a coding agent modifies an existing repository to resolve a problem and is evaluated by tests (Jimenez et al., 2024). Each task consists of three parts.

Executable environment. Each task provides a task repository R: a fixed code snapshot that contains the bug to be fixed or lacks the functionality to be implemented. The accompanying runtime,

dependencies, and configuration make its code and tests executable, providing a reproducible starting state for solving and evaluation. In our synthesis setting, we construct R by modifying a source repository S in which the required behavior is already implemented.

Problem statement. The natural-language problem statement explains the bug or missing functionality in R and specifies the expected behavior. The agent addresses this request by producing a code patch, a set of file changes applied to R. A known solution, called the reference repair, is used to validate the task and is withheld from the agent; the agent's patch may use a different implementation.

Evaluation: F2P and P2P tests. An evaluation test suite V evaluates the patched repository. Fail-to-pass (F2P) tests fail on R and must pass after a correct patch, checking that the requested behavior has been achieved. Pass-to-pass (P2P) tests pass on R and must remain passing after the patch, checking that the existing behavior they cover has been preserved. The task is solved only when all designated F2P and P2P tests pass (OpenAI, 2024). Private evaluation tests are withheld from the agent, which can use the repository's existing tests during its attempt.

2.2 Training from Executable SWE Tasks

Trajectory collection. Executable SWE tasks provide environments in which teacher agents attempt development requests by inspecting code, editing files, and running tests. The resulting trajectories record model responses, tool actions, and execution feedback. Evaluation determines whether an attempt solves the task, while quality screening determines whether its trajectory is suitable for training. Depending on the selection policy, both successful and unsuccessful attempts may be retained, and multiple trajectories may be collected for the same task.

Supervised fine-tuning. Selected trajectories can be used to fine-tune a student model π_θ. Given the problem statement and preceding interaction, the model learns to predict the teacher's next response or tool action. Tool feedback provides context, while the training loss is applied to the teacher's assistant outputs (Pan et al., 2025). Task synthesis determines which problems are available for practice; trajectory collection and quality screening turn those problems into supervision.

3 SWE-Craft: Our Approach

To realize target-conditioned SWE task synthesis, we propose SWE-Craft, a two-stage framework (Figure 2). Graph-Guided Task Design first uses repository structural relations in a CPG to locate candidate code regions and provide their relation contexts. Building on this context, the task design agent queries the CPG, matches functional modules to capability targets, and develops a task specification. Task Construction and Validation then supplements missing tests, constructs the task repository, and validates it against the fixed specification and tests.

3.1 Graph-Guided Task Design

Before the task synthesis, we analyze the base model's task-solving trajectories using model-assisted statistical analysis and human synthesis to summarize recurring difficulties as weak spots. These observations inform a set of capability targets. To begin synthesis, we specify the capability targets and select a source repository. Each capability target describes the behaviors or reasoning abilities we want a model to exhibit. It may specify a broad task type (bug fixing, feature addition, or feature enhancement), a finer task subtype (such as configuration-related bugs or input-validation errors), a behavioral focus (such as error recovery), or a structural reasoning scope (such as reasoning across execution paths). These dimensions can be combined: a target might request configuration-related bug fixes that require tracing a value across multiple execution paths.

We select source repositories from a prepared pool of projects with fixed revisions, runnable environments, and reproducible native tests which already present in each repository (Appendix D). We denote the selected source snapshot by S and use its implemented behavior as evidence of what a feasible task can require. The design problem is therefore to *identify a functional module that supports a capability target and translate that target into concrete requirements*.

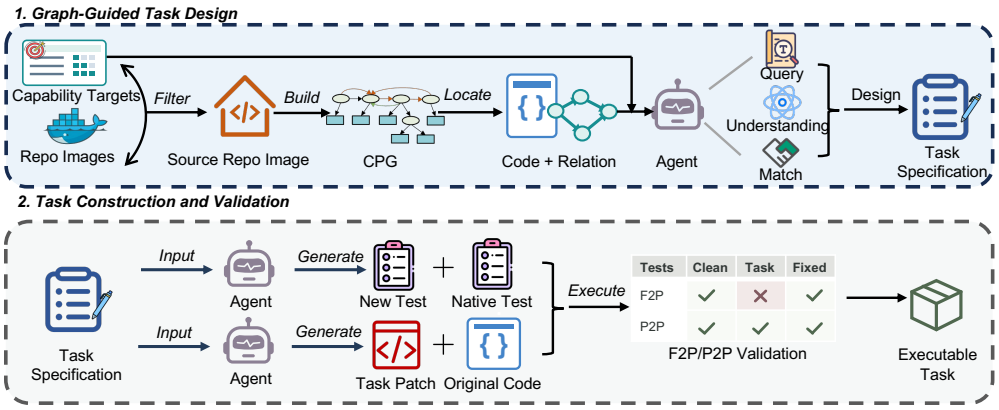


Figure 2: The SWE-CRAFT task synthesis pipeline.

Locating candidate code through repository structure. Because capability targets describe desired abilities rather than code locations, we first need to identify where tasks can concretely exercise those abilities. Text search can find relevant names, but it does not reveal how the retrieved components interact, leaving the agent to reconstruct dependencies from separate code fragments. To make these dependencies explicit, we build a code property graph G from S using the supported CodeQL or Joern analyses (GitHub, n.d.; Joern Project, n.d.) (Appendix D). In G, nodes identify functions, parameters, types, and state, while directed, typed edges encode repository structural relations such as calls, data propagation, and state access. Together with linked source locations, these relations expose which components interact, in which direction, and across which function or file boundaries. We then use this structure to prioritize candidate code regions and attach a compact relation context to each. To preserve useful breadth within the limited context budget, this initial view samples different relation types, directions, and file boundaries, so that a dense local neighborhood does not obscure potentially relevant connections elsewhere.

Matching functional modules to capability targets. Given these code locations and relation contexts, the task design agent consults the capability-target table, which records the requested targets and their coverage (Appendix B.2). To match candidate code with capability targets, the agent progressively expands the initial CPG context by querying G. By tracing callers, consumers, propagation paths, and state-access relations, it follows the connected structure of a functional module across functions and files. This graph-guided exploration keeps the relationships between newly discovered components and the candidate code explicit, providing direct structural evidence for target matching. Guided by these relations, the agent inspects linked source and native tests to interpret the module's behavior and determine more precisely which capability requirements it can support. Without this structure, the agent would have to reconstruct these connections from separate searches and code readings, increasing the risk of overlooking a dependency or assuming a relationship that the code does not support.

Based on this evidence, the agent compares the recovered functional modules with the requirements in the capability-target table. For each plausible match, it identifies the behavior that a task should require, the components responsible for it, and the interactions a solver would need to reason about. The agent then selects a functional module and a corresponding capability target based on the existing task distribution and the degree of relationship matching, carrying forward the associated design requirements. Thus, locating narrows where to explore, while graph-guided matching determines which capability target that code can actually support.

Designing the task from the selected match. Once the module and capability target have been matched, the agent develops their design requirements into a task specification. Building on the established repository behavior, it defines the required behavior, retained behavior, acceptance scenarios, and the intended gap that construction should introduce. To keep these requirements coherent, it links them to the module's components and the repository structural relations through which they interact. These links expose which related paths and consumers must be considered

together, helping the design cover the relevant behavior across functions and files instead of treating the initially located code as the whole task.

Before the specification passes to construction, its referenced code elements and relation identities are checked against G and S, and its behavioral claims are checked against source and test evidence. If this check reveals an omitted consumer or an unsupported interaction, the agent revisits the graph and revises the design before approval. In this way, graph guidance continues beyond locating: the relations used to match a module to a target also make the resulting task requirements checkable. The approved task specification then provides construction with both the requirements and their supporting evidence.

3.2 Task Construction and Validation

The specification defines what the task should require; the second stage must make that requirement executable without weakening it. In particular, the task repository R must expose the intended gap while preserving the designated surrounding behavior, and the evaluation tests V must distinguish those two obligations.

Making the specification testable. To turn the specification into an executable acceptance criterion, we first check whether native tests cover its requirements. Since these tests were written for the repository's existing development needs, relying on them alone could leave an omitted path or a partial implementation undetected, a limitation also observed in SWE-bench evaluations (Yu et al., 2025; Wang et al., 2026b). The test generation agent therefore retains applicable native tests and adds behavioral tests for uncovered acceptance scenarios and preservation checks, forming V. Before construction begins, all selected tests must run stably and pass on S; their assertions follow the specified behavior rather than a particular future patch.

Preserving the requirements during construction. Test coverage is useful only if construction cannot change what the tests are meant to enforce. If the same agent could revise the specification or its evaluator while editing code, it could drop a difficult requirement or weaken a test until an easier task passed validation. We therefore freeze the approved specification and V and separate test generation from the construction agent. The constructor receives the specification and source context, while supplementary evaluation tests and their authoring history remain private; tests and evaluation commands are protected from its edits. This separation keeps construction accountable to the original requirements. Under these constraints, the construction agent turns S into R by introducing the specified fault or removing the specified feature or mode while preserving the retained behavior. Reversing these edits provides the reference repair.

Validating the constructed task. We run the fixed test suite V on S, R, and the reference-repaired repository. All tests must pass on S and after reference repair; in R, at least one F2P test must expose the specified gap while all P2P tests remain passing. To ensure these outcomes reflect the intended task, we also check the code changes against the specification and exclude unrelated build failures or missing test discovery as F2P evidence.

For an accepted task, the problem statement is written from the fixed requirements and verified observations. We export R, the statement, and V, keeping the reference repair and private tests separate from the teacher's inputs. The resulting task is ready for the trajectory collection and quality screening described in Section 2.2.

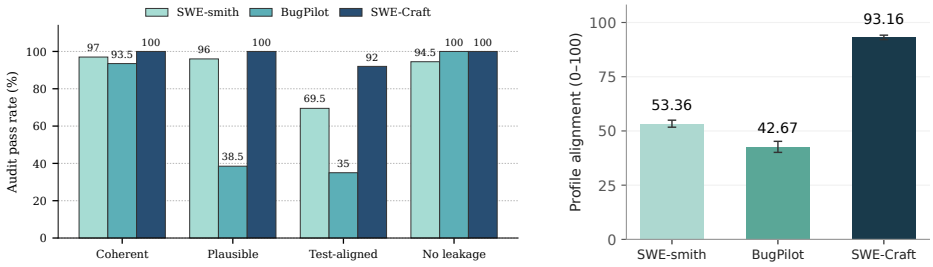
4 Experiments and Applications

4.1 Experimental Setup

Synthesis repositories. SWE-rebench V2 provides executable repository tasks with pre-built Docker images (Badertdinov et al., 2026). Guided by the capability targets, we select 500 images with suitable repository functionality and a domain distribution approximating SWE-bench Pro and SWE-bench ML (Deng et al., 2026; Yang et al., 2025), covering Python, C, C++, Go, Java, JavaScript, TypeScript, PHP, Ruby, and Rust. The selected synthesis repositories exclude all repositories used for diagnosis or evaluation. All synthesis methods share this selected pool.

Table 1: Task statistics and teacher repair results across synthesis methods.

Method	Tasks	Issue tok.	Patch tok.	Files	Net LOC	Changed LOC	Evals.	Solve (%)	Actions
SWE-smith	1,500	168.00	479.13	1.00	-5.14	39.13	1,324	77.13	46.19
BugPilot	1,500	315.83	479.48	1.12	+5.94	13.17	1,061	77.47	69.44
SWE-CRAFT	1,500	247.37	963.84	1.94	+28.63	51.30	1,106	52.47	72.91



(a) Task quality audit pass rates.

(b) Profile alignment scores.

Figure 3: Task quality and profile alignment across synthesis methods.

Baselines. We compare SWE-CRAFT with two task-synthesis methods. **SWE-smith** creates executable bugs through code perturbations (Yang et al., 2025); we mix its Procedural, LM Modify, and LM Rewrite strategies. **BugPilot** uses a coding agent to introduce test-revealed bugs (Sonwane et al., 2025). All BugPilot results use its BugInstruct variant, which explicitly requests bug injection; FeatAdd has a low construction success rate in our setting. The methods share the selected repository pool while retaining their respective synthesis procedures.

Models and training. All three methods use GLM-5.2 for model-driven synthesis with Claude Code v2.1.205 as the agent harness. Teacher trajectories are collected using GLM-5.2 with OpenHands SDK v1.22.1 as teacher. Our SFT student is Ling-3.0-flash (InclusionAI, 2026), released by InclusionAI, a hybrid-attention mixture-of-experts model with 124B total and 5.1B active parameters. We fine-tune it on 1,500 trajectories per training source for three epochs with a learning rate of 5.5×10^{-5}.

4.2 STATISTICS OF GENERATED TASKS

Training tasks should require substantive repository reasoning rather than small local edits. We characterize this difficulty through patch scope and the shared GLM-5.2 teacher's solve rate and action count (Table 1). SWE-CRAFT tasks contain roughly twice as many patch tokens as either baseline and span more files. Their teacher solve rate is 52.47%, compared with over 77% for both baselines, and they require more actions on average. Together, these results indicate that SWE-CRAFT generates tasks with broader edit scope and greater repair demands than the compared synthesis baselines.

4.3 TASK QUALITY AND PROFILE ALIGNMENT

Difficult tasks are useful only if they are well formed and exercise the desired engineering behaviors. We use GLM-5.2 as a source-blinded judge of goal coherence, predecessor plausibility, test alignment, and solution leakage. Separately, *profile alignment* measures fit to a shared task profile covering a coherent goal, plausible predecessor, behavioral scope, causal coupling, and repair depth (Appendix F.3). This common rubric allows comparison across methods. Figure 3 shows stronger coherence, plausibility, and test alignment for SWE-CRAFT, with leakage avoidance matching BugPilot. Its profile-alignment score exceeds SWE-smith and BugPilot by 39.79 and 50.49 points. These gains demonstrate that SWE-CRAFT tasks provide more effective supervision for repository-level problem solving than the compared synthesis baselines.

Table 2: Downstream performance on Ling-3.0-flash-base after SFT on tasks synthesized by different methods.

Training data	SWE-bench Pro (%)	SWE-bench ML (%)
Base model	39.08 ± 0.62	56.67 ± 1.73
SWE-smith	44.14 ± 0.83	64.45 ± 0.69
BugPilot	44.83 ± 0.21	65.00 ± 0.33
SWE-CRAFT (No-Graph)	45.55 ± 0.41	66.11 ± 0.51
SWE-CRAFT	$\mathbf{46.88 \pm 0.55}$	$\mathbf{68.00 \pm 0.58}$

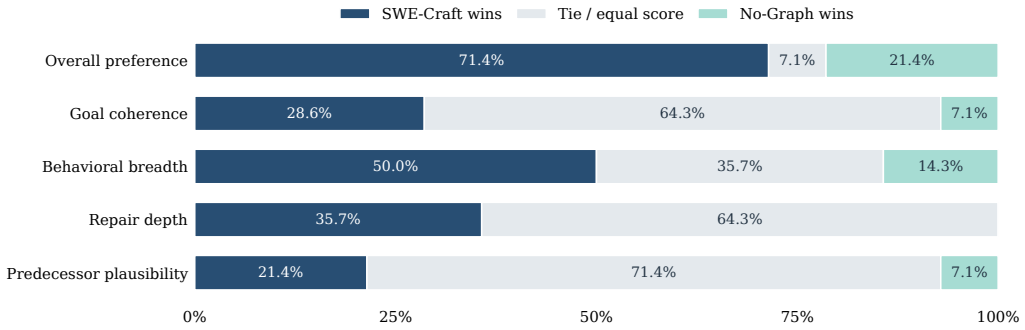


Figure 4: SWE-CRAFT win, tie, and loss rates against No-Graph.

4.4 SUPERVISED FINE-TUNING PERFORMANCE

We assess the downstream utility of the synthesized tasks by fine-tuning Ling-3.0-flash on each source and measuring Pass@1 on SWE-bench Pro and SWE-bench ML, reporting mean $\pm$ SD across three evaluations. Table 2 shows improvements of 7.80 and 11.33 percentage points over the base model and 2.05 and 3.00 points over BugPilot, the stronger synthesis baseline. These gains indicate that capability-targeted task synthesis can produce effective training supervision for improving the model's downstream software-engineering capabilities.

4.5 ABLATION STUDIES

Graph-guided design. No-Graph uses the same repository starting points and task-type interface, but discovers modules through source and test inspection; subsequent construction and validation are unchanged (Appendix E.4). GLM-5.2 judges accepted task pairs for coherence, behavioral breadth, repair depth, and predecessor plausibility. Figure 4 favors SWE-CRAFT overall and shows more wins than losses on each dimension. SFT improves over No-Graph by 1.33 and 1.88 points (Table 2). Graph-guided design improves both the quality of synthesized tasks and downstream training utility.

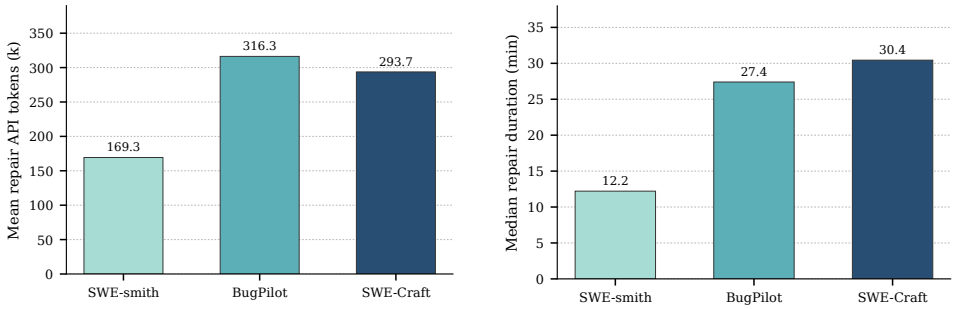
Preserving the task specification. Task construction can drift from the requirements established during design. We audit specification preservation under Direct construction, Mutation with a revisable specification, and Frozen construction with fixed requirements. Table 3 shows that Frozen preserves issue–specification alignment and has no audited cases with missing obligations. Its *declared-goal alignment*, which checks the constructed artifact against the original task goal, improves over Direct but remains slightly below Mutation. Freezing helps retain requirements without improving every audit measure or establishing complete test coverage.

4.6 EFFICIENCY, DATA-COLLECTION COST, AND ROBUSTNESS

Efficiency and collection cost. More demanding tasks may require greater effort to collect useful teacher trajectories. We measure GLM-5.2 teacher token usage and repair duration to assess this trade-off (Figure 5). SWE-CRAFT requires more tokens than SWE-smith but fewer than BugPilot, while its median repair duration is the longest. Its broader repair demands therefore come with additional collection time, rather than uniformly higher token usage. This means SWE-CRAFT

Table 3: Specification preservation audit results across construction variants.

Audit criterion	Direct (%)	Mutation (%)	Frozen (%)
Issue–specification alignment ↑	95.7	97.9	100.0
Declared-goal alignment ↑	91.5	95.7	93.6
Unchanged requirement scope ↑	80.9	95.7	95.7
Cases with missing obligations ↓	6.4	4.3	0.0



(a) Mean teacher token usage.

(b) Median teacher repair duration.

Figure 5: Teacher token usage and repair duration across synthesis methods.

maintains stronger profile alignment across task subsets, with teacher collection taking longer but using fewer tokens than BugPilot.

Robustness across task regimes and audit evidence. To examine whether profile alignment depends on task mix or inconsistent issue–patch pairs, we compare GLM-5.2 scores within bug-fixing tasks and issue–patch-consistent tasks (Figure 6). SWE-CRAFT retains a substantial lead over both baselines in each subset, with the smallest margin exceeding 25 points. The result supports shared-profile alignment across these subsets.

4.7 CASE STUDY

A `django-cors-headers` task illustrates how a capability target becomes a task specification (Figure 7). The illustrative capability target is to coordinate validation and runtime behavior while preserving supported cases. It is operationalized in a feature-enhancement task involving configuration/input validation and reasoning across consumers.

The matching module is the CORS whitelist: configuration checks accept malformed origins, while runtime matching ignores the scheme. The specification requires rejecting incomplete or malformed origins and blocking same-host requests with a different scheme, while preserving `null`, `file://`, and valid matching origins. The reference repair coordinates `checks.py` and `middleware.py` around this contract. The recorded teacher attempt fails, illustrating the repair demands of satisfying the full specification. This case illustrates how SWE-CRAFT grounds a capability target in a concrete task that requires coordinated changes across validation and runtime behavior.

5 RELATED WORK

Software-engineering task construction. Existing work obtains executable SWE tasks either from development histories or through synthetic construction. SWE-ReBench V2, SWE-Factory, and ScaleSWE reconstruct historical tasks (Badertdinov et al., 2026; Guo et al., 2026; Zhao et al., 2026), while SWE-smith, SWE-Synth, and BugPilot create new tasks through mutations, rewrites, or injected bugs (Yang et al., 2025; Pham et al., 2026; Sonwane et al., 2025). SWE-CRAFT makes the capability target explicit before construction, enabling task requirements to be designed around the desired model capabilities and preserved through behavioral checks.

Program structure and behavioral specifications. Repository graphs have been used to expose cross-file dependencies for solving existing tasks, as in RepoGraph and CodexGraph (Ouyang et al.,

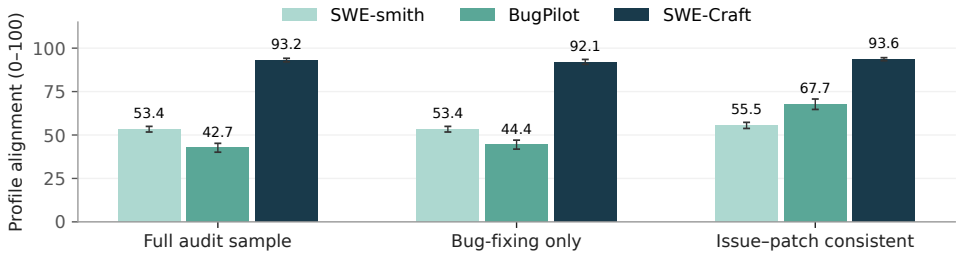


Figure 6: Profile alignment scores by task subset.

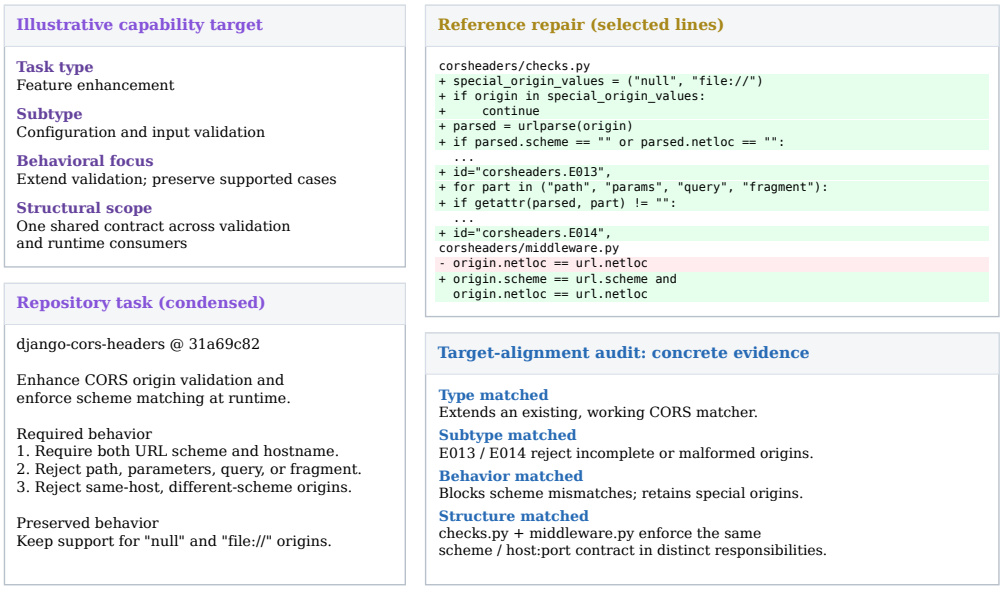


Figure 7: Illustrative capability target and task specification for CORS origin validation.

2025; Liu et al., 2025), while RPG and CodeSpec use structured representations for development planning or requirements (Luo et al., 2026; Wang et al., 2026a). SWE-CRAFT uses repository graphs during task design, following source-linked relations to identify interacting components and ground task requirements in their behavior.

Learning from interactions and adaptive generation. SWE-Gym and SWE-smith collect agent trajectories for training (Pan et al., 2025; Yang et al., 2025), while adaptive methods such as Socratic-SWE and FrogNano use learner behavior to guide data generation (Xiao et al., 2026; Kim et al., 2026). These directions are complementary to SWE-CRAFT, which translates supplied capability targets into executable tasks.

6 CONCLUSION

In this paper, we have presented SWE-CRAFT for target-conditioned SWE task synthesis. It translates capability targets into repository-grounded task specifications and realizes them through separate construction and validation. With a shared repository pool, SWE-CRAFT achieves stronger task-profile alignment and SFT gains over SWE-smith of 2.74 and 3.55 points and over BugPilot of 2.05 and 3.00 points on SWE-bench Pro and Multilingual. In our future work, we plan to study alternative sources of capability targets, closed-loop target refinement as models evolve, and the effects of individual target categories. We believe that target-conditioned synthesis can provide a useful foundation for constructing training data that more directly addresses recurring model weaknesses.

AI USE STATEMENT

The authors used generative AI tools for manuscript editing, code implementation assistance, data preparation, and model-assisted statistical analysis of base-model performance trajectories. Human synthesis of that analysis informed the capability targets. Generative models also serve as functional components of the synthesis, teacher-trajectory collection, and audit procedures described in this paper. The authors reviewed AI-assisted outputs and take responsibility for the final claims, code, and experimental results.

ETHICS STATEMENT

This work studies automated software-engineering task synthesis and coding-agent evaluation using open-source repositories. Task construction operates in isolated environments, and the selected synthesis repositories exclude repositories used for diagnosis or evaluation.

REPRODUCIBILITY STATEMENT

Appendix B describes how weak-spot analysis informs capability targets and distinguishes this preparation step from synthesis. Appendices B.2 and C provide target/specification forms and artifact-grounded illustrative cases. Appendix D records model configurations and turn limits; Appendices E, F, and G document agent prompts, audit scoring, and the specification-preservation comparison. The accompanying assets retain the original prompt wording and data fields, with terminology mappings explained in the text. The anonymous code repository is linked in the abstract.

REFERENCES

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*, 2018.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-Supervised Bug Detection and Repair. In *Advances in Neural Information Processing Systems*, volume 34, pp. 27865–27876. Curran Associates, Inc., 2021.

Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pp. 1–19, 1970.

Anthropic. Claude 3.7 Sonnet and Claude Code. Official product announcement, 2025.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Aleksandr Golubev. SWE-rebench V2: Language-Agnostic SWE Task Collection at Scale. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026.

Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. Generative Code Modeling with Graphs. In *International Conference on Learning Representations*, 2019.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering*, 48(9): 3280–3296, 2022.

Wei Cheng, Yuhan Wu, and Wei Hu. Dataflow-Guided Retrieval Augmentation for Repository-Level Code Completion. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7957–7977. Association for Computational Linguistics, 2024.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks? In *Proceedings of the 43rd International Conference on Machine Learning*, 2026.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46701–46723. Curran Associates, Inc., 2023.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

GitHub. CodeQL Documentation. Official software documentation, n.d. Accessed September 25, 2026.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*, 2021.

Lianghong Guo, Yanlin Wang, Caihua Li, Wei Tao, Pengyu Yang, Jiachi Chen, Haoyu Song, Duyu Tang, and Zibin Zheng. SWE Data Construction, Automatically! *Proceedings of the ACM on Software Engineering*, 3(FSE):525–546, 2026.

Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring Frontier Coding Agents on Original, Long-Horizon Engineering Tasks. arXiv preprint arXiv:2607.07946v1, 2026.

InclusionAI. Ling-3.0-flash. https://huggingface.co/inclusionAI/Ling-3.0-flash, 2026. Hugging Face model.

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.

Joern Project. Joern: Code Property Graph Documentation. Official software documentation, n.d. Accessed September 25, 2026.

Minseon Kim, Zhengyan Shi, Emiliano Penaloza, Christopher Cui, Roger Creus Castanyer, Maryam Hashemzadeh, Isadora White, Jonathan Light, Jeonghye Kim, Matheus Pereira, Darya Moldavskaya, Chinmay Singh, Fabio Vera, Baolin Peng, Xingdi Yuan, Marc-Alexandre Côté, and Alessandro Sordoni. FrogNano: Training a 4B Coding Agent via Online Task Synthesis. arXiv preprint arXiv:2609.07925v4, 2026.

Li-Cheng Lan, Andrew Bai, Minhao Cheng, Cho-Jui Hsieh, and Tianyi Zhou. Exploring Expert Failures Improves LLM Agent Tuning. arXiv preprint arXiv:2504.13145v2, 2025.

Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. In *International Conference on Learning Representations*, pp. 47832–47850, 2024a.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. GraphCoder: Enhancing Repository-Level Code Completion via Coarse-to-fine Retrieval Based on Code Context Graph. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 570–581, 2024b.

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. CodexGraph: Bridging Large Language Models and Code Repositories via Code Graph Databases. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 142–160. Association for Computational Linguistics, 2025.

Jane Luo, Xin Zhang, Steven Liu, Jie Wu, Jianfeng Liu, Yiming Huang, Yangyu Huang, Chengyu Yin, Ying Xin, Yuefeng Zhan, Hao Sun, Qi Chen, Scarlett Li, and Mao Yang. RPG: A Repository Planning Graph for Unified and Scalable Codebase Generation. In *International Conference on Learning Representations*, pp. 91932–91985, 2026.

OpenAI. Introducing SWE-bench Verified. Official benchmark announcement, 2024.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph. In *International Conference on Learning Representations*, pp. 30098–30121, 2025.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training Software Engineering Agents and Verifiers with SWE-Gym. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47717–47737. PMLR, 2025.

Minh V. T. Pham, Huy N. Phan, Nhat Hoang Phan, Cuong Chi Le, Tien N. Nguyen, and Nghi Bui. Synthetic Repo-level Bug Dataset for Training Automated Program Repair Models. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pp. 349–361, 2026.

Huy N. Phan, Hoang N. Phan, Tien N. Nguyen, and Nghi D. Q. Bui. RepoHyper: Search-Expand-Refine on Semantic Graphs for Repository-Level Code Completion. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pp. 14–25, 2025.

Rushi Qiang, Yuchen Zhuang, Anikait Singh, Percy Liang, Chao Zhang, Sherry Yang, and Bo Dai. MLE-Smith: Scaling MLE Tasks with Automated Multi-agent Pipeline. In *International Conference on Learning Representations*, pp. 5963–5990, 2026.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 49–61, 1995.

Monoshi Kumar Roy, Simin Chen, Benjamin Steenhoek, Jinjun Peng, Gail Kaiser, Baishakhi Ray, and Wei Le. CodeSense: a Real-World Benchmark and Dataset for Code Semantic Reasoning. In *International Conference on Learning Representations*, pp. 13735–13768, 2026.

Atharv Sonwane, Isadora White, Hyunji Lee, Matheus Pereira, Lucas Caccia, Minseon Kim, Zhengyan Shi, Chinmay Singh, Alessandro Sordoni, Marc-Alexandre Côté, and Xingdi Yuan. BugPilot: Complex Bug Generation for Efficient Learning of SWE Skills. arXiv preprint arXiv:2510.19898v2, 2025.

Peiding Wang, Li Zhang, Fang Liu, Taichuan Li, and Yinghao Zhu. CodeSpec: Dual Executable Specifications for Agentic Long-Horizon Feature Development. arXiv preprint arXiv:2607.26777v1, 2026a.

Renxi Wang, Xudong Han, Yixuan Zhang, Timothy Baldwin, and Haonan Li. NAT: Enhancing Agent Tuning with Negative Samples. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 7385–7398. Association for Computational Linguistics, 2025.

You Wang, Michael Pradel, and Zhongxin Liu. Are ``Solved Issues"in SWE-bench Really Solved Correctly? An Empirical Study. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pp. 169–181, 2026b.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-Based Software Engineering Agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025.

Chuan Xiao, Zhengbo Jiao, Shaobo Wang, Wei Wang, Bing Zhao, Hu Wei, Linfeng Zhang, and Lin Qu. Socratic-SWE: Self-Evolving Coding Agents via Trace-Derived Agent Skills. arXiv preprint arXiv:2606.07412v1, 2026.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*, pp. 590–604, 2014.

Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pp. 797–812, 2015.

John Yang, Carlos 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.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling Data for Software Engineering Agents. In *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc., 2025.

Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. UTBoost: Rigorous Evaluation of Coding Agents on SWE-Bench. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3762–3774. Association for Computational Linguistics, 2025.

Yucheng Zeng, Shupeng Li, Daxiang Dong, Ruijie Xu, Zimo Chen, Liwei Zheng, Yuxuan Li, Zhe Zhou, Haotian Zhao, Lun Tian, Heng Xiao, Tianshu Zhu, Longkun Hao, and Jianmin Wu. SWE-Hub: A Unified Production System for Scalable, Executable Software Engineering Tasks. arXiv preprint arXiv:2603.00575v1, 2026.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484. Association for Computational Linguistics, 2023.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 783–794, 2019.

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.

Jiale Zhao, Guoxin Chen, Fanzhe Meng, Minghao Li, Jie Chen, Hui Xu, Yongshuai Sun, Wayne Xin Zhao, Ruihua Song, Yuan Zhang, Peng Wang, Cheng Chen, Jirong Wen, and Kai Jia. Immersion in the GitHub Universe: Scaling Coding Agents to Mastery. arXiv preprint arXiv:2602.09892v4, 2026.

Pujun Zheng, Zixin Shang, Shufan Jiang, Wenhui Tian, Dongsheng Zhu, Zerun Ma, Dingbo Yuan, and Qi Zhang. SWE-Bench Pro Verified: A Reliable Benchmark for Software Engineering Agents. arXiv preprint arXiv:2609.08149v2, 2026.

Qixing Zhou, JiaCheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, Dandan Tu, and Zhaoxiang Zhang. FeatureBench: Benchmarking Agentic Coding for Complex Feature Development. In *International Conference on Learning Representations*, pp. 22019–22055, 2026.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Part I

Appendix

Table of Contents

A Related Work **15**
- A.1 Software-Engineering Task Construction . . . 15
- A.2 Code Graphs and Code Property Graphs . . . 15
- A.3 Behavioral Specifications and Evaluation . . . 16
- A.4 Learning from Interactions and Adaptive Generation . . . 16

B Weak Spots, Capability Targets, and Task Specifications **16**
- B.1 From performance analysis to capability targets . . . 16
- B.2 Capability target and task specification forms . . . 17

C Case Studies Across Task Types **18**
- C.1 Bug Fixing: Restoring Pyflakes Diagnostics . . . 18
- C.2 Feature Addition: Per-Project OSV-Scanner Policies . . . 19
- C.3 Feature Enhancement: Consistent CORS Origin Validation . . . 20

D Agent Configuration **21**
- D.1 Models and execution . . . 21
- D.2 Agent turn limits . . . 21

E Agent Prompts **21**
- E.1 Prompt inputs and outputs . . . 21
- E.2 Task Design Agent . . . 22
- E.3 Construction Agent . . . 22
- E.4 No-Graph design . . . 23
- E.5 Test Generation Agent . . . 23
- E.6 Issue projection . . . 24

F Audit Prompts and Scoring **25**
- F.1 Audit inputs . . . 25
- F.2 Artifact quality and native-test alignment . . . 26
- F.3 Shared task profile and alignment scoring . . . 27
- F.4 Full/No-Graph pairwise audit . . . 27

G Specification-Preservation Prompts **28**
- G.1 Independent construction review . . . 29

A Related Work

A.1 Software-Engineering Task Construction

Historical and synthetic task collection provide complementary sources of engineering problems. SWE-ReBench V2, SWE-Factory, and ScaleSWE reconstruct tasks from development histories, using environment recovery and execution checks to make them usable for training or evaluation (Badertdinov et al., 2026; Guo et al., 2026; Zhao et al., 2026). SWE-smith synthesizes test-breaking changes, while SWE-Synth uses code rewriting to construct bug-fix data (Yang et al., 2025; Pham et al., 2026). BugPilot studies instructed bug generation and regressions introduced while agents implement features as distinct synthesis variants (Sonwane et al., 2025). SWE-Hub combines task-production components, FeatureBench extracts functionality through test-rooted dependency analysis, and MLE-Smith constructs tasks for machine-learning engineering (Zeng et al., 2026; Zhou et al., 2026; Qiang et al., 2026). SWE-Craft conditions task design on a requested capability target and grounds that target in connected repository behavior before constructing the task state.

A.2 Code Graphs and Code Property Graphs

Program-analysis foundations. Code graphs represent program elements and their relationships at different granularities. Control-flow analysis characterizes possible execution paths (Allen, 1970); program dependence graphs make control and data dependencies explicit (Ferrante et al., 1987). System dependence graphs support interprocedural slicing, while graph-reachability formulations enable precise interprocedural dataflow analysis (Horwitz et al., 1990; Reps et al., 1995). These representations provide complementary structural evidence: a syntactic relationship, a possible control-flow step, and a data dependence describe different aspects of program behavior. Static relationships also depend on the analysis assumptions and do not by themselves establish that a path is exercised in a particular execution.

Code property graphs and vulnerability analysis. Yamaguchi et al. (2014) introduced the code property graph (CPG), combining abstract syntax trees, control-flow graphs, and program dependence graphs in a common representation for vulnerability discovery. Subsequent work inferred graph traversals for taint-style vulnerabilities (Yamaguchi et al., 2015). Joern provides CPG construction and traversal infrastructure (Joern Project, n.d.). CodeQL exposes queryable program databases and libraries for control-flow and dataflow analysis (GitHub, n.d.). In SWE-Craft, supported backend analyses provide source-linked relations that are normalized into the repository representation used for task design.

Learning from program structure. Graph-based program models combine syntactic and semantic edges to learn variable-use and naming relationships (Allamanis et al., 2018), and graph-based generative models use evolving program graphs during code generation (Brockschmidt et al., 2019). ASTNN learns representations from abstract-syntax-tree fragments, while GraphCodeBERT incorporates dataflow into code pretraining (Zhang et al., 2019; Guo et al., 2021). For vulnerability detection, Devign and the graph-based approach studied by Chakraborty et al. (2022) use structural program representations (Zhou et al., 2019). BugLab jointly trains bug detection and repair with a learned selector of synthetic bugs, connecting structured program modeling to the choice of training examples (Allamanis et al., 2021). These approaches mainly use structure as model input or as part of a prediction or generation process. SWE-Craft uses explicit, source-linked relations to select repository contexts and ground the requirements of executable training tasks.

Repository-level retrieval and agent navigation. RepoCoder iterates retrieval and code generation, providing a retrieval-based approach to repository-level completion (Zhang et al., 2023). RepoBench and CrossCodeEval evaluate the use of cross-file context in code completion (Liu et al., 2024a; Ding et al., 2023). Graph-based retrieval makes dependencies more explicit: DraCo uses dataflow-guided repository contexts; GraphCoder retrieves through code context graphs; and RepoHyper searches, expands, and refines semantic-graph contexts (Cheng et al., 2024; Liu et al., 2024b; Phan et al., 2025). For issue resolution, RepoGraph supplies repository-level code relationships, and CodexGraph lets agents query code graph databases (Ouyang et al., 2025; Liu et al., 2025). AutoCodeRover combines code-structure-aware search with program improvement, while Agentless organizes localization,

repair, and validation into a constrained workflow (Zhang et al., 2024; Xia et al., 2025). These studies inform repository exploration and task solving. SWE-CRAFT applies structural evidence earlier, when deciding which connected behavior can support a requested training task, and validates the resulting requirements against source code and tests.

A.3 BEHAVIORAL SPECIFICATIONS AND EVALUATION

RPG uses an explicit repository planning graph for codebase generation, and CodeSpec introduces executable architectural and behavioral specifications for feature development (Luo et al., 2026; Wang et al., 2026a). CodeSense evaluates code-semantic reasoning, complementing structural access with an assessment of behavioral understanding (Roy et al., 2026). In SWE-CRAFT, graph relations guide source inspection, while the task specification records required and preserved observations before construction. Its test-generation stage addresses incomplete evaluation, a problem examined by UTBoost and PatchDiff (Yu et al., 2025; Wang et al., 2026b). DeepSWE and SWE-Bench Pro Verified further emphasize reliable evaluation of repository-level changes (Huang et al., 2026; Zheng et al., 2026). These works motivate behavioral checks that distinguish incomplete repairs without unnecessarily restricting valid implementations.

A.4 LEARNING FROM INTERACTIONS AND ADAPTIVE GENERATION

SWE-Gym and SWE-smith collect agent interactions as training data (Pan et al., 2025; Yang et al., 2025). Exploring Expert Failures and Negative-Aware Training (NAT) investigate how unsuccessful trajectories can provide useful learning signals (Lan et al., 2025; Wang et al., 2025). These studies motivate distinguishing task validity from the outcome of one teacher rollout. Adaptive approaches additionally use learner behavior to guide generation: Socratic-SWE derives skills from traces, and FrogNano synthesizes tasks near the learner's frontier (Xiao et al., 2026; Kim et al., 2026). SWE-CRAFT supplies the repository-grounded synthesis step conditioned on capability targets.

B WEAK SPOTS, CAPABILITY TARGETS, AND TASK SPECIFICATIONS

B.1 FROM PERFORMANCE ANALYSIS TO CAPABILITY TARGETS

Our experimental preparation procedure analyzes base Ling-3.0-flash trajectories on SWE-bench Pro and SWE-bench Multilingual. Model-assisted statistical analysis organizes observed behavior; human synthesis groups recurring difficulties and formulates capability targets. A *weak spot* is an inferred pattern of difficulty in the model's observed performance. A *capability target* states the behaviors or reasoning abilities the model should exhibit. A *task specification* translates those capability requirements into concrete software-behavior obligations in a repository. Thus, the target concerns model capability, while synthesis and fine-tuning are procedures that can use it.

Illustrative observation	Inferred weak spot	Illustrative capability target
An edit handles one path but overlooks another consumer.	Difficulty maintaining a shared contract across paths.	Maintain a shared contract across consumers or execution paths.
A fix handles the ordinary case but misses a recovery path.	Difficulty reasoning about error or recovery behavior.	Reason about error conditions and implement complete recovery behavior.
An extension changes behavior that should remain supported.	Difficulty extending functionality while preserving existing behavior.	Extend functionality while preserving supported behavior.
A configuration check and its runtime consumer disagree.	Difficulty connecting validation with downstream use.	Keep configuration validation consistent with downstream use.

Table 4: Illustrative transitions from observations to weak spots and capability-target requirements.

Repository selection and comparison scope. Targets guide the selection of suitable synthesis repositories. All selected repositories exclude those used for diagnosis or evaluation, and the

compared synthesis baselines share this selected pool. Thus, the comparison makes the target-informed repository pool available to all methods.

Diagnostic records and evaluation scope. Repository separation limits direct reuse of diagnostic or evaluation codebases in synthesis.

B.2 Capability target and task specification forms

Capability target: desired model abilities before repository matching. Listing 1 gives a compact form for operationalizing a capability target through the four dimensions in Section 3.1. Task type and subtype identify the engineering setting; behavioral focus and structural reasoning scope describe the abilities exercised. A target can use a subset of these dimensions, with its identifier linking it to coverage in the capability-target table. Repository paths and acceptance checks are determined after module matching.

Listing 1: Compact capability-target form. Angle brackets denote fields to fill.

```
{
  "target_id": "<target identifier>",
  "task_type": "<bug fixing | feature addition | feature enhancement>",
  "task_subtype": "<finer engineering problem>",
  "behavioral_focus": "<behavior to practice>",
  "structural_reasoning_scope": "<interactions to reason about>"
}
```

Task specification: a capability target grounded in repository behavior. Listing 2 organizes the resulting requirements and evidence. Required behavior states what the solver must restore or implement; preserved behavior states what must continue to work. Acceptance scenarios bind both to observable outcomes. The intended gap directs construction from S to R, while source bindings and structural evidence connect each requirement to the matched module. The approved specification guides test selection or supplementation and construction.

Listing 2: Compact task-specification form, with requirement identifiers linking behavior, scenarios, and source responsibilities.

```
{
  "target_id": "<selected target>",
  "repository": {"name": "<repository>", "revision": "<fixed revision>"},
  "functional_module": "<connected behavior and responsible components>",
  "public_goal": "<user-visible capability>",
  "required_behavior": [{"id": "R1", "observation": "<required outcome>"}],
  "retained_behavior": [{"id": "K1", "observation": "<preserved outcome>"}],
  "acceptance_scenarios": [
    {"trigger": "<input and precondition>", "expected": "<observation>",
     "covers": ["R1"]},
    {"trigger": "<preservation input>", "expected": "<unchanged outcome>",
     "covers": ["K1"]}
  ],
  "intended_gap": "<fault or missing capability to introduce in R>",
  "source_bindings": [
    {"role": "<responsibility>", "location": "<path::symbol>",
     "supports": ["R1"]}
  ],
  "structural_evidence": [
    {"relation_id": "<graph reference>", "type": "<relation type>",
     "from": "<source element>", "to": "<source element>",
     "supports": ["R1"]}
  ],
  "behavioral_evidence": ["<supporting source or native-test locations>"]
}
```

C Case Studies Across Task Types

The following illustrative capability targets use the four dimensions above. The repository-specific task specifications are condensed from saved task artifacts and reference repairs. Acceptance scenarios state expected observations. Each case follows the progression: illustrative capability target, matched module, task specification, and acceptance scenarios. The accompanying `target_specification/evidence_manifest.json` records this evidence boundary.

C.1 Bug Fixing: Restoring Pyflakes Diagnostics

Listing 3: Illustrative bug-fixing capability target.

```
{
  "target_id": "T-diagnostics",
  "task_type": "bug fixing",
  "task_subtype": "error handling and diagnostic reporting",
  "behavioral_focus": "restore distinct error reports and preserve failure status",
  "structural_reasoning_scope": "coordinate error paths with reporting across modules"
}
```

Matched module. In `PyCQA/pyflakes` (revision `43541ee`), `pyflakes/api.py` classifies parsing and file-access failures, and `pyflakes/reporter.py` renders diagnostics. The relevant interaction carries error category and source-location information from `check` to the reporter; `checkPath` handles file-access errors. This module supports practice in coordinating distinct failure paths with their shared reporting interface.

Specification field	Repository-grounded content
Public goal	Restore clear, location-qualified diagnostics while retaining the checker's failure signal.
Required R1	For a syntax error with a known column, report file, line, column, message, source line, and a caret at the offending position.
Required R2	Report undecodable source with the stable message `problem decoding source`.
Required R3	Report a file-access failure using the OS reason, without an errno prefix or a second copy of the path inside that reason.
Preserved K1–K2	Preserve ordinary checking of syntactically valid input (K1) and the nonzero warning count and exit status for failures (K2).
Intended gap	Route parsing failures through a generic exception channel and stringify file-access errors, losing the category-specific diagnostic behavior.

Table 5: Pyflakes task-specification summary.

Covers	Input or precondition	Expected observation
R1, K2	Python input with a syntax error and known column	Location, source line, and caret on stderr; failure status.
R2, K2	Declared encoding cannot decode the file	Stable decoding message; warning count 1 and nonzero exit.
R3, K2	Missing or inaccessible path	Path followed by the clean OS reason; failure status.
K1	Syntactically valid Python input	Ordinary checking and reporting continue.

Table 6: Acceptance scenarios for the bug-fixing case.

The reference repair reconnects error classification to `Reporter.syntaxError` and extracts the file-access reason in `checkPath`. Restoring the reporting format alone would leave the decoding and file-access paths incomplete.

C.2 Feature Addition: Per-Project OSV-Scanner Policies

Listing 4: Illustrative feature-addition capability target.

```
{
  "target_id": "T-project-policy",
  "task_type": "feature addition",
  "task_subtype": "configuration discovery and policy application",
  "behavioral_focus": "add per-source policies with expiration semantics",
  "structural_reasoning_scope": "carry source context through resolution and filtering"
}
```

Matched module. In `google/osv-scanner` (revision `39b4f2d`), `pkg/config/config.go` resolves configurations and evaluates ignore entries, while `pkg/osvscanner/osvscanner.go` filters scan results. The relevant path passes each result's source to `ConfigManager.Get`, then applies `Config.ShouldIgnore` inside `filterResponse`. It connects source identity, configuration choice, and the reported vulnerability set.

Specification field	Repository-grounded content
Public goal	Add per-project ignore policies to a scanner using a global default or an explicit configuration override.
Required R1	An ignore entry whose expiration has passed no longer suppresses the vulnerability.
Required R2	Discover `osv-scanner.toml` beside each manifest; load and cache it, with a `Loaded filter from` notice.
Required R3	Apply each source's resolved configuration to its own results in a multi-source scan.
Preserved K1–K3	An explicit `--config` override takes precedence (K1); entries without an expiration remain active (K2); absent local files use the default configuration (K3).
Intended gap	Retain global/default filtering but remove local discovery, source-specific policy application, and expiration-aware ignore decisions.

Table 7: OSV-scanner task-specification summary.

Covers	Input or precondition	Expected observation
R1	A listed vulnerability with a past expiration	The vulnerability appears in the scan output.
R2	A manifest with a neighboring ignore file, no override	Local policy is applied; repeated access reuses the configuration without another load notice.
R3	Two sources contain the same vulnerability; only one local policy ignores it	Suppression occurs only for that source.
K1, K2	An explicit override lists the vulnerability without an expiration	The override applies and suppresses it, regardless of local policy.
K3	No local configuration and no override	The default configuration applies.

Table 8: Acceptance scenarios for the feature-addition case.

The reference repair makes configuration lookup depend on each source path and evaluates expiration when filtering. Local discovery alone cannot deliver the requested capability if every result still uses one global policy.

C.3 Feature Enhancement: Consistent CORS Origin Validation

Listing 5: Illustrative feature-enhancement capability target, expanding the main-text case.

```
{
  "target_id": "T-origin-validation",
  "task_type": "feature enhancement",
  "task_subtype": "configuration and input validation",
  "behavioral_focus": "extend validation while preserving supported behavior",
  "structural_reasoning_scope": "coordinate validation and runtime consumers"
}
```

Matched module. In `adamchainz/django-cors-headers` (revision `31a69c82`), `corsheaders/checks.py` validates `CORS_ORIGIN_WHITELIST`, and `corsheaders/middleware.py` uses the same setting to authorize request origins. These two consumers share an origin contract: configuration validation and runtime matching must agree on which origin components matter.

Specification field	Repository-grounded content
Public goal	Enhance the existing whitelist with structured-origin validation and scheme-aware matching while preserving supported special values.
Required R1	Emit `corsheaders.E013` for an ordinary origin missing its scheme or network location.
Required R2	Emit `corsheaders.E014` for an origin containing a path, parameters, query, or fragment.
Required R3	Require both scheme and network location to match at runtime; a cross-scheme request receives no `Access-Control-Allow-Origin` header.
Preserved K1–K2	Preserve the special values `null` and `file://` without E013/E014 errors (K1), and authorize valid matching origins (K2).
Intended gap	Keep the existing whitelist interface but leave entry validation at the generic sequence check and compare only network locations at runtime.

Table 9: CORS task-specification summary.

Covers	Input or precondition	Expected observation
R1	Check an entry `example.com` or `https://`	E013 identifies the incomplete origin.
R2	Check an entry `https://example.com/foobar`	E014 rejects the path component.
R3	Allow `https://example.org`; request from `http://example.org`	GET and OPTIONS emit no allow-origin header.
K1	Check entries `null` and `file://`	Neither E013 nor E014 is emitted.
K2	Allow and request from `https://example.org`	The matching origin remains authorized.

Table 10: Acceptance scenarios for the feature-enhancement case.

The reference repair adds per-entry URL checks and makes the runtime matcher compare scheme together with network location. Updating only validation leaves cross-scheme authorization unchanged; updating only runtime matching leaves malformed settings undiagnosed.

D AGENT CONFIGURATION

D.1 MODELS AND EXECUTION

Task synthesis uses GLM-5.2 with Claude Code v2.1.205. Full and No-Graph use the same synthesis model. Each task starts from a fixed repository revision with a runnable environment and native tests. Teacher trajectories use GLM-5.2 with OpenHands SDK 1.22.1. The teacher receives the public issue and task repository through the OpenHands agent interface.

The SFT student is Ling-3.0-flash, trained on 1,500 trajectories per source for three epochs with a learning rate of 5.5×10^{-5} and assistant-only loss. Table 2 reports means and sample standard deviations over three repeated evaluations.

D.2 AGENT TURN LIMITS

Table 11 gives the configured turn limits for each interactive stage. Completion and correction stages continue the existing task when the preceding stage needs additional work.

Stage	Maximum agent turns
Task design	100
Task construction	48
Construction completion	12
Test generation	28
Test completion	20
Test correction	10
Teacher repair	200

Table 11: Agent turn limits used in the experiments.

Issue projection uses a structured completion followed by deterministic rendering. Execution validation runs the designated tests on the clean, constructed, and reference-repaired states. These operations use the approved specification and evaluator.

E AGENT PROMPTS

E.1 PROMPT INPUTS AND OUTPUTS

Table 12 summarizes the prompts used at each stage. The listings show the central design and construction instructions and the test-generation and issue-projection templates. Braced fields are filled with task-specific context. Complete templates and prompt-assembly functions are included in the accompanying assets.

Agent	Prompt inputs	Required output
Task design	Capability target, code-seed context, source-linked relations, source and native-test evidence	Behavioral specification with required and preserved behavior, responsibilities, and acceptance scenarios
Construction	Approved specification, bound production-source responsibilities, repository context	Production-source edits realizing the specified task state
Test generation	Public behavioral obligations, observation paths, test destination and repository conventions	Executable tests for uncovered obligations
Issue projection	Verified public functionality and behavior fields	User-facing functionality brief and issue
Teacher repair	Public issue, task repository, and tool feedback through OpenHands	Repair trajectory and proposed patch

Table 12: Prompt inputs and outputs for task synthesis and teacher collection.

E.2 TASK DESIGN AGENT

The design prompt combines the requested task type and structural reasoning scope with a code seed and queryable source-linked context. The agent inspects source and native tests and records concrete obligations in a task specification, called a *Blueprint* by the implementation. The complete message is assembled by `build_agent_guided_blueprint_prompt`. Its `L2` and `L3` structural profiles request, respectively, coordinated public branches across production-code boundaries and nonlocal functionality or lifecycle/protocol reasoning; `RELATIVE_HIGH` asks for the strongest naturally supported scope among candidates. These are prompt-side structural requirements.

Listing 6: Task Design Agent: core instructions and output discipline (excerpts).

```
You are the repair-first Task Design Agent of SWE-Craft.

The task type and structural reasoning scope are fixed externally. CPG has selected an
important code seed and supplied a qualified, queryable structural context.
CPG facts identify structural importance, implementation roles and program
relationships. Retrieved repair shapes are abstract design references, never
fixed recipes. Read clean production source and official tests and query the
local CPG ledger whenever the initial context is insufficient. The displayed
neighborhood is deliberately compact; the persisted ledger is the complete
fact authority. CPG does not infer a capability surface for you: use source and
tests to interpret its endpoints, relations and cross-file paths semantically.

Do not edit any file during this phase. You may use read-only Read/Grep/Glob
and Bash commands such as rg, sed, find, and the supplied CPG query tool. Do
not run repository tests or arbitrary programs; the pipeline owns the exact
official A-B test execution.

{evidence_discipline}

Fixed task type: {task_type}
Fixed structural reasoning scope: {structural_reasoning_scope}
{task_direction}
{scope_requirements}

[... intervening task-specific instructions ...]

Return exactly one compact semantic JSON object with this schema and no
Markdown. Do not invent protocol identities. For CPG references, copy the
original `endpoint_id` / `edge_id` values exactly from the supplied CPG context
or ledger. If an endpoint ID is unavailable, use a structured locator containing
both `file_path` and `qualified_name` (or `symbol`), optionally with the source
line copied from CPG. Never use a made-up short alias. The pipeline supplies
only schema/run/context/seed IDs, stable local semantic IDs, and a qualified
seed-to-anchor path when one can be proven from the ledger; it does not invent
semantic cross-section bindings.
```

E.3 CONSTRUCTION AGENT

The constructor receives the approved task specification and its source bindings. The prompt fixes the selected behavior while allowing the agent to choose the production-code implementation. The full message is assembled by `agent_guided_implementation_prompt`; the shared system template is `construction_agent_v4.txt`. The original construction prompt uses *base model* for a counterfactual repository state. This implementation term is distinct from the Ling-3.0-flash student, which is the meaning of *base model* in the paper's analysis and evaluation.

Listing 7: Construction Agent: approved specification and edit constraints (excerpts).

```
Continue in this same repository session and construct the
approved counterfactual repository base in production source.

{blueprint_context}

Work backwards from the clean capability and its future repair obligations.
Create the plausible older/incomplete implementation described by
`counterfactual_base`; do not merely damage the clean code until tests fail.

[... intervening task-specific instructions ...]


The seed is a structural origin, not a mandatory edit line. Use the CPG roles
and relations to choose the semantically correct production surface. The
```

```
blueprint constrains capability, base model, responsibilities, endpoint/file
ownership and behaviors. Exact lines and code shape remain free, but the Agent
may not compress the task back into one file, one local mutation or a set of
witness-specific edits.

You may use Read/Grep/Glob/Bash/Edit/Write, read existing tests, and query the
local CPG ledger. Do not edit/add tests, fixtures, snapshots, documentation, or
pipeline metadata. Do not run ad-hoc or alternative test commands; the
pipeline owns the exact official B run. Keep the project buildable and avoid
unconditional failure, stubs, syntax/import/link errors, debug markers, or
independent mutations. If the approved task is not implementable, make no
production edit and report IMPLEMENTATION_BLOCKED.
```

E.4 NO-GRAPH DESIGN

No-Graph uses the same task-type interface and repository starting point, but discovers candidate modules through source and test inspection. The prompt builders accept task-type and structural-profile fields; the No-Graph prompt removes CPG-specific bindings and navigation instructions. The corresponding construction prompt replaces graph bindings with repository evidence. This comparison does not require identical selected modules, concrete behaviors, or task specifications.

Listing 8: No-Graph design: repository exploration instructions.

```
You are the repair-first Task Design Agent of SWE-Craft.

The task type and structural reasoning scope are fixed externally. No code seed, function,
file, capability surface, implementation role or program relation has been
selected for you. Start at the exact clean repository root and use production
source, official tests and documentation to discover the strongest bounded
semantic candidate yourself. The current checkout is the clean reference; the
edit direction is clean to a plausible older/incomplete base, and the reverse
is the reference repair.

Do not edit any file during this phase. You may use the same read-only
Read/Grep/Glob/LSP and Bash repository-navigation tools as the Full arm. Do not
access CPG/CodeQL/Joern databases, relation ledgers, previous synthesis output,
Git history, network resources, or paths outside this repository except the
pipeline-provided read-only stable official-test catalog. Do not run
repository tests or arbitrary programs; the pipeline owns the exact official
A-B test execution.
```

E.5 TEST GENERATION AGENT

The test prompt supplies behavior witnesses, public triggers, observation paths, and the target test-file location. Generated tests follow native test conventions and are checked on the clean repository before construction.

Listing 9: Test Generation Agent prompt template.

```
You are generating evaluator tests for a SWE-Craft synthetic benchmark.

The blueprint agent designed a counterfactual bug for a clean codebase. Some
behaviors need a test because no existing official test observes them. Cover ALL
the behaviors below.{placement_directive}

Test requirements:
- The pipeline has already resolved the exact repository-relative destination
  `{test_file_path}`. Do not spend calls listing the repository root or test
  directories. Read only the nearest existing test/config/source slices needed
  for syntax and setup, using at most four targeted evidence calls before the
  first Write/Edit. Create the evaluator promptly; later calls are for one
  focused compile/syntax correction, not broader survey.
- Keep the evaluator narrowly proportional to its witness count: for this
  {len(needed)}-behavior request, target at most {min(220, 80 + 60 * len(needed))}
  non-blank source lines. Reuse the nearest official test's existing helpers,
  fixtures, fakes, and setup pattern. Do not construct a general-purpose fake
  server, archive builder, protocol implementation, or duplicate repository
  test infrastructure when one focused stub/fixture can observe the public
  behavior. If the behavior genuinely cannot be tested within this bounded
  shape, report that limitation instead of emitting a large speculative test.
{witness_identity_requirement}
  If a behavior specifies
```

```
  multiple inputs or boundary examples, execute all of them as assertions inside
  that behavior's single test method/subtest. You may also include the
  human-readable behavior id, but the opaque marker is the authoritative
  protocol identity. Shared setup may be reused, but each witness must make its
  own public-behavior assertion; do not duplicate an assertion merely to satisfy
  the count.
- The test must PASS on the current clean code (the worktree is clean).
- Follow the repository's existing test conventions (check test files in the repo).
- Use the same test framework the repo already uses (e.g. go test, pytest, jest,
  Minitest, JUnit). Infer from existing test files.{framework_directive}
- Test function names must be distinctive and follow the repo's naming convention.
{generated_name_requirement}
- Tests must assert observable behavior, not internal implementation details.
  Do not use reflection, private-field access, object-memory inspection, or
  language-specific escape hatches (for example Ruby `instance_variable_get`)
  merely to prove that a requested behavior is implemented by a particular
  cache, field, helper, or call path. Observe the public result instead. Private
  access is permitted only when it is unavoidable to reach the behavior in a
  repository's existing internal-test convention; even then, assert the
  behavior's externally meaningful result rather than the implementation shape.
- Every generated test must also terminate promptly on the described degraded
  implementation, not only on clean code. If the degraded behavior can block,
  wait, retry, poll, or deadlock, arrange an independent bounded fallback
  stimulus that releases the operation and then assert the wrong observation,
  or use a framework timeout that terminates the complete worker/process.
  Never rely on an assertion timeout while leaving a spawned thread alive.
- Exercise the supplied Trigger/Usage and observation path, including a
  production responsibility that the described degraded base behavior changes.
  If the degradation is in a caller, orchestrator, lifecycle, propagation or
  renderer path, a direct unit call to an unchanged lower-level helper is not a
  witness for that behavior even when the helper output resembles the final
  assertion. Do not replace an end-to-end public trigger with such a shortcut.
  For a sequential or short-circuiting parser, matcher, validator, dispatcher,
  or lifecycle, choose inputs that make every earlier gate succeed so execution
  must reach the responsibility named by this behavior. A test that obtains the
  expected result from an earlier mismatch/error/return is not a witness.
- Do NOT import or reference any bug/patch/mutation concepts | test pure clean behavior.

Behaviors to test:
{chr(10).join(behavior_specs)}

Write the test file now using the Write tool. Then perform the smallest relevant
syntax/compile/link check described above (for example go test compilation,
python -m py_compile, node --check, ruby -c, one native target build, or Cargo
--no-run). Do not finish while that focused check has a known syntax, undefined
name, import, or compilation error. Do not run the full test suite | the
pipeline will do that.
```

E.6 ISSUE PROJECTION

The issue-projection prompt converts verified public behavior into a structured user-facing brief. Its output is rendered into the public issue using the task type and current/desired behavior fields. Here the original prompt's *capability* fields refer to software functionality, not to a diagnosed capability of the student model.

Listing 10: Issue-projection system and instance prompts.

```
normalizer_system: |-
  You convert a private task designer's verified behavior requirements into a
  safe, user-perspective capability brief for a separate GitHub issue writer.

  The input contains only a coarse capability and requirement fields that were
  declared public during task design.  Even these fields may still contain
  bug-designer language, exact synthetic examples, internal identifiers, error
  codes, or causal explanations.  They are not ready to publish.

  Preserve what a user must be able to do and the externally visible distinction
  between the current and desired behavior.  Abstract away how the behavior is
  implemented or repaired.  Keep independent requirements distinct, but merge
  arbitrary examples that express the same boundary into descriptions of user-visible
     functionality.

  Rules (STRICT):
  - Return exactly one JSON object matching the requested schema, with every
    numeric index exactly once.  Do not add behaviors.
  - Use only a user's action/context, current visible limitation, and desired
    public outcome.  Never state a root cause, implementation mechanism, data
```

```
    flow, control flow, internal state transition, or repair instruction.
  - Remove source paths, line numbers, internal modules/classes/helpers/fields,
    test identities, requirement IDs, benchmark/synthesis terminology, and
    base/clean/reference-version narration.
  - Do not copy exact error codes, arbitrary fixture names or values, synthetic
    messages, assertion wording, or generated identifiers.  Describe their
    semantic role instead (for example, "a missing configured value").
  - The token "[private detail]" means that a designer-specific name was removed.
    Omit the token and express only the surrounding user-visible behavior; never
    print the token or guess what it replaced.
  - A public command or API name may remain only when the observable scenario
    explicitly uses it and the name is necessary to explain the user's action.
    Never promote an identifier found only in current/desired prose to an API.
  - Do not expose that verification exists or that any check failed.
  - Preserve important semantic categories such as missing versus wrong-typed
    input, ordering versus lifecycle behavior, or one supported mode versus an
    unsupported mode when those distinctions define separate requirements.
  - Treat behavior_goal and the coarse capability as the authority for semantic
    direction when current/desired prose uses an antonym inconsistently (for
    example favored versus unfavored).  Use the scenario to describe context and
    current/desired fields to distinguish present from requested results.  Do
    not preserve an obvious terminology inversion.
  - Do not claim history, documentation, compatibility, or prior support unless
    the input explicitly establishes it.

  Task-family framing (STRICT):
  - bug_fix: an exercised public behavior produces the wrong observable result.
  - feature_add: users need a capability that the current product does not offer;
    never call it a regression or imply it used to work.
  - feature_enhance: an existing capability is limited or incomplete for the
    supplied cases and should be extended; never frame it as a defect.

  The output is a publication-ready semantic brief.  A deterministic renderer
  will turn it into the final issue without asking another model to elaborate,
  so every field must be natural, self-contained user-facing prose.  Return JSON
  only.

normalizer_instance: |-
  Convert the following verified blueprint projection into a safe user-view
  brief.  The requirement index is an opaque transport key.

  <TASK_TYPE>{{ task_type }}</TASK_TYPE>
  <RAW_BLUEPRINT_PROJECTION>
  {{ raw_projection_json }}
  </RAW_BLUEPRINT_PROJECTION>

  Required JSON schema:
  {
    "capability": "one user-facing capability phrase",
    "behaviors": [
      {
        "index": 1,
        "user_context": "what the user is doing, without arbitrary fixture detail",
        "current_limitation": "only the externally visible present behavior",
        "desired_outcome": "only the public result/capability users need"
      }
    ]
  }

  Return exactly one JSON object and no Markdown.
```

F AUDIT PROMPTS AND SCORING

F.1 AUDIT INPUTS

All task-quality audits use GLM-5.2 with source labels hidden. Table 13 lists the evaluated evidence and the corresponding prompt. Source labels are restored after judging to aggregate results.

Audit	Judge input and output	Prompt
Artifact quality	Issue, diff, F2P/P2P test identities and available test patch; four Yes/No/Unknown verdicts	Listing 11
Native-test alignment	Public issue, complete native F2P list and test-source excerpts; statement–test verdict	Listing 12
Profile alignment	Shared task profile, issue and associated diff; five scores from 0 to 4	Listing 13
Full/No-Graph	Two accepted tasks from the same repository revision; five scores per task and A/B/tie preference	Listing 14
Specification preservation	Shared original specification and three construction artifacts; obligation-level and scope judgments	Listing 17

Table 13: Audit evidence and prompt references.

F.2 Artifact quality and native-test alignment

The artifact-quality audit evaluates sampled tasks from each source for goal coherence, predecessor plausibility, statement–test alignment, and absence of severe answer leakage. The native-test prompt supplies the statement–test assessment for baseline tasks with existing repository tests.

Listing 11: Four-criterion artifact-quality judge.

```
You are a source-blinded auditor of a synthetic software repair task. Judge the realized TASK,
     not a solver's success or the dataset source. Use only the supplied public issue,
     associated diff, named tests and test patch when present. The diff direction may be
     repair or construction; do not assume either. Test bodies may be unavailable. Return yes/
     no/unknown for each criterion and a short concrete reason. Unknown is required when
     evidence is insufficient.
Criteria:
goal_coherence: The PUBLIC ISSUE alone states a consistent observable goal and conditions for
     an independent solver.
predecessor_plausibility: The issue and diff support a credible incomplete predecessor for
     that goal; obvious unrelated edits or a mismatched issue justify no. If direction is
     unclear, use unknown.
statement_test_alignment: The actual test patch, when present, and named F2P/P2P tests
     correspond to the public goal and retained behavior. Names alone are weak evidence; use
     unknown unless alignment or conflict is clear.
no_severe_leakage: The PUBLIC ISSUE alone does not reveal the exact solution or hidden
     assertions; a normal behavioral specification is not leakage.
Do not infer execution stability from test names. Do not infer acceptance of alternative
     repairs from a reference diff. Return exactly JSON:
{"criteria":{"goal_coherence":{"verdict":"yes|no|unknown","reason":"..."},"
     predecessor_plausibility":{"verdict":"yes|no|unknown","reason":"..."},"
     statement_test_alignment":{"verdict":"yes|no|unknown","reason":"..."},"no_severe_leakage
     ":{"verdict":"yes|no|unknown","reason":"..."}},"confidence":"high|medium|low"}
```

Listing 12: Native F2P statement–test alignment judge.

```
You are a source-blinded reviewer of a synthetic software repair task. Compare its PUBLIC
     ISSUE with the COMPLETE named FAIL_TO_PASS (F2P) test list and available source excerpts
     of the ORIGINAL NATIVE tests. Baseline tasks need not introduce a new test patch; an
     empty test_patch is not missing evaluation evidence. PASS_TO_PASS (P2P) names are
     retained-behavior context only. The associated repair/construction diff may help
     interpret behavior but is not a substitute for tests.

Return JSON exactly: {"statement_test_alignment":"yes|no|unknown","reason":"concrete issue-to-
     F2P behavior match or mismatch, naming relevant tests/functions", "evidence_basis":"
     native_test_source|test_names_only|insufficient", "uncovered_or_conflicting_requirements
     ":[], "confidence":"high|medium|low"}.

Use yes when available F2P tests exercise the public issue's central required behavior and no
     material conflict is visible. Use no for a concrete contradiction or a central required
     behavior demonstrably absent from the COMPLETE F2P list and available source. Use unknown
     only if the supplied names/source cannot decide; do not use unknown solely because no
     NEW test patch exists. Source excerpts may cover only a subset of the complete F2P list:
     do not infer that unseen tests lack coverage. Do not infer execution stability or solver
     success from test names.
```

F.3 SHARED TASK PROFILE AND ALIGNMENT SCORING

The profile-alignment audit applies the same shared task profile to all corpora. The judge receives up to 6,500 characters of issue text and 5,500 characters of associated diff. It is a common evaluation rubric, distinct from the individual capability target supplied before generation and from the repository-specific task specification.

Listing 13: Shared-profile audit prompt and rubric (original implementation wording).

```
You are conducting a source-blinded, evidence-based audit of software-repair tasks. The task
     source, method label, benchmark grade, and claimed difficulty are hidden. Compare the
     supplied public issue and associated code diff to the SAME COMMON TARGET for every item.
     Do not infer a higher score from patch length, number of files, issue verbosity, task
     type, or suspected corpus. Do not score tests or teacher repair success. The diff may
     show a reference repair or a predecessor construction; infer its relation to the issue
     from the artifact, and use unknown if uncertain.

TARGET:
Common production target derived from SWE-CPG Agent V4 Blueprint prompts and L2/L3 task
     profiles:
1. One coherent, bounded, observable software capability, in one of three forms: bug_fix
     repairs incorrect existing behavior; feature_add restores one capability absent from a
     credible earlier base; feature_enhance broadens a correct narrower capability. All three
     forms are valid; do not prefer one type by itself.
2. A plausible working but incomplete predecessor, without unrelated breakage, stubs, syntax/
     build failures, or arbitrary regression.
3. Several related, independently observable behavior obligations, including an edge/error/
     default/compatibility or retained behavior condition. Count distinct semantic obligations,
      not repeated examples of one assertion.
4. Causal coupling across implementation responsibilities such as producer/consumer, public
     API/core, state transition/observation, dispatch/type, or cross-component propagation. A
     single file can qualify if it contains distinct causally necessary roles. Several touched
      files are not sufficient by themselves.
5. Repair requires understanding the shared semantic contract and coordinating meaningful
     decisions. A one-line literal/operator/guard reversal or copying a sibling is low
     alignment. Do not reward size, artificial complexity, boilerplate, or problem verbosity.
This is a common profile for all three corpora, not an instance-specific private Blueprint.
     Judge task artifacts, not synthesis method, benchmark outcome, or solver success.

Score each dimension as an integer 0..4. Use the evidence anchors below. Be conservative if
     only a plausible interpretation exists; mark evidence_sufficiency limited/insufficient
     and confidence low when needed.
- goal_kind: 4 clear coherent observable goal of one allowed kind, 3 mostly clear, 2 partly
     defined, 1 vague/conflicted, 0 contradictory or unrelated.
- predecessor: 4 credible incomplete but working base is supported by issue+diff, 3 plausible
     with limited support, 2 ambiguous, 1 implausible, 0 contradictory/unrelated.
- behavior_scope: 4 at least three distinct related obligations including a boundary or
     preservation condition; 3 at least two distinct obligations with such a condition; 2 two
     related behaviors but no clear boundary/preservation, or only one behavior plus an edge
     condition; 1 one isolated outcome; 0 no coherent behavior.
- causal_coupling: 4 clear multi-role/cross-layer causal dependency needed for one goal; 3 two
      connected responsibilities plausibly necessary; 2 local but nontrivial linked logic; 1
     isolated single decision; 0 unrelated/padding. Do not count file number itself.
- repair_depth: 4 multiple substantive coordinated semantic decisions; 3 substantial semantic
     repair with more than one dependent step; 2 moderate local policy/data-flow reasoning; 1
     obvious literal/operator/guard/revert; 0 trivial or incoherent.

Return exactly one JSON object with keys: task_kind (bug_fix|feature_add|feature_enhance|
     unknown), evidence_sufficiency (adequate|limited|insufficient), issue_diff_consistency (
     yes|no|unknown), scores (object with exactly the five named integer dimensions), evidence
      (object with one short concrete reason per dimension referring to issue/diff behavior),
     confidence (high|medium|low). No overall score; it is computed by the auditor. All
     reasons must be supported by visible artifact. Do not guess access to private Blueprint
     or source code beyond the diff.
```

For goal fit g, plausible predecessor p, behavior scope b, causal coupling c, and repair depth r, each scored from zero to four, the aggregate is

$$S = \tfrac{15}{4}g + \tfrac{15}{4}p + \tfrac{20}{4}b + \tfrac{25}{4}c + \tfrac{25}{4}r, \qquad S \in [0, 100]. \tag{1}$$

Confidence intervals are 95% bootstrap intervals for the mean.

F.4 FULL/NO-GRAPH PAIRWISE AUDIT

The judge compares accepted task pairs from matched repository revisions. Each arm may select its own concrete behavior. Tasks are presented as A and B, and the judge returns five dimension

scores plus an overall preference. The reported preference is conditional on accepted pairs; it is not a fixed-task-specification intervention.

Listing 14: Full/No-Graph pairwise task-quality prompt.

```
You are reviewing two synthetic software repair tasks from the same clean repository and
     commit. Their construction methods and task identities are hidden. They were
     independently chosen and may target different functionality. Assess each task on its own
     merits; do not penalize different targets or reward a task for matching the other. Ignore
      solver success and the number of synthesis attempts.

For each task assess: (1) coherent, observable public goal; (2) substantive functional breadth,
      including distinct behavior branches; (3) causal implementation depth rather than
     superficial line count; (4) plausible defective predecessor and bounded repair; (5)
     whether supplied test bodies or named F2P/P2P tests support the stated goal and preserved
      behavior. The construction diff runs from clean code to a defective predecessor, so
     reversing it gives a reference repair. The optional test patch adds tests to the clean
     checkout. Do not infer a test passed from its name or body. If original test bodies are
     absent, acknowledge that limit.

Return only JSON. For A and B, give integers 0-4 for goal, breadth, depth, predecessor, tests,
      and one short evidence sentence per dimension. 0 means clear defect, 1 weak, 2 adequate,
      3 strong, 4 exceptional. Use null if evidence is insufficient. Also provide preference A,
      B, or tie based on overall task-design quality and a concise reason. Do not infer the
     construction method.
{"A":{"goal":{"score":0,"reason":""},"breadth":{"score":0,"reason":""},"depth":{"score":0,"
     reason":""},"predecessor":{"score":0,"reason":""},"tests":{"score":0,"reason":""}},"B":{"
     goal":{"score":0,"reason":""},"breadth":{"score":0,"reason":""},"depth":{"score":0,"
     reason":""},"predecessor":{"score":0,"reason":""},"tests":{"score":0,"reason":""}},"
     preference":"A|B|tie","preference_reason":""}
```

G SPECIFICATION-PRESERVATION PROMPTS

Direct, Mutation, and Frozen use shared repository contexts and a fixed public issue. A common planning prompt defines the original requirements. Their construction prompts vary whether that specification is supplied and whether it can be revised. The prompt implementation names the Mutation arm `MUTABLE`.

Listing 15: Shared specification-planning prompt for the construction ablation.

```
You are the read-only planning stage of a controlled software-task construction
experiment. The repository is currently in the correct clean state A. From the
public repair request and supplied clean production files, specify the exact
engineering problem that a constructor should create by degrading A to a
plausible incomplete predecessor B. Do not edit code, invent unsupported
behavior, prescribe one exact textual diff, or expose test answers.

Return exactly one JSON object:
{
  "goal": "bounded public repair goal",
  "obligations": [{"id":"o1", "statement":"observable behavior to restore", "evidence_paths
     ":["repo/relative/file"]}],
  "retained_behaviors": [{"id":"k1", "statement":"behavior that must keep working", "
     evidence_paths":["repo/relative/file"]}],
  "responsibilities": [{"id":"r1", "role":"semantic owner", "paths":["repo/relative/file"]}],
  "degradation_strategy": "plausible clean-to-incomplete transformation",
  "acceptance_scenarios": [{"id":"s1", "trigger":"public trigger", "expected":"clean behavior",
      "degraded":"observable predecessor behavior"}],
  "confidence": "high|medium|low",
  "abstain_reason": "empty when feasible, otherwise concrete reason"
}

Every path must be one of the supplied source paths. Obligations and scenarios
must be distinct and supported by source evidence. Keep the problem open to
multiple repairs. If evidence is insufficient, abstain instead of inventing.
```

Listing 16: Direct, Mutation, and Frozen construction prompts.

```
DIRECT
Construct an incomplete predecessor B directly from clean A. There is no
separate Blueprint: select the concrete degradation jointly while editing.
Preserve the public repair goal and unrelated behavior. Return a unified git
diff over supplied production paths only; never edit or add tests. Avoid
syntax-only sabotage, unconditional crashes, TODO stubs, or exact-answer hints.
```

```
MUTABLE
Construct B from clean A using the supplied initial Blueprint. You MAY revise
the Blueprint before construction when implementation convenience or evidence
suggests a different goal, obligation, retained behavior, responsibility, or
scenario. Record every semantic revision explicitly. Then return a unified git
diff over supplied production paths only; never edit or add tests.

FROZEN
Construct B from clean A while treating the supplied initial Blueprint as an
immutable semantic contract. Do not delete, add, rewrite, narrow, or reinterpret
its goal, obligations, retained behaviors, responsibilities, or scenarios.
Either realize it with a unified git diff over supplied production paths only,
or abstain. Never edit or add tests.
```

The side-by-side judge compares realized artifacts with the original task specification using source excerpts and named F2P/P2P tests. Its output separates requirement preservation, changes in public task scope, and test-coverage evidence. The field `declared_target_alignment` is reported as *declared-goal alignment* in Table 3: it checks a construction's declared task goal against the original specification, rather than the repository-independent capability target or shared task profile.

Listing 17: Specification-preservation audit prompt.

```
You are a blinded auditor comparing three synthesized repair tasks made from the SAME clean
     repository and fixed public repair request. The generation arms are hidden as A/B/C. One
     arm was made without access to the original Blueprint; compare it to that Blueprint POST
     HOC anyway. The construction patches intentionally degrade clean behavior to make buggy
     predecessor states. Do not mistake a removed behavior in a predecessor for a removed
     obligation in the solver-facing repair task.

Judge each arm's functional requirements against the ORIGINAL Blueprint: goal, every
     obligation, retained behavior, responsibilities, and acceptance scenarios. Use the fixed
     public issue, constructor target/revisions, actual clean-to-buggy patch, clean source
     excerpts, and complete named F2P/P2P lists. The public issue is fixed for all arms.
     Distinguish (a) constructor internally targeting less, (b) buggy predecessor choosing a
     different failure mode, (c) solver-facing task actually losing an obligation, and (d)
     tests apparently not covering an obligation. Test names alone may be inconclusive; say
     unclear rather than inventing test behavior. Do not judge by construction success rate or
      patch size alone.

Return exactly one JSON object with keys:
{
  "obligation_matrix": [{"id":"original Blueprint obligation id", "A":"preserved|narrowed|
     expanded|unclear", "B":"preserved|narrowed|expanded|unclear", "C":"preserved|narrowed|
     expanded|unclear", "reason":"concrete functional evidence"}],
  "arms": {
    "A":{"public_issue_alignment":"yes|no|unclear", "declared_target_alignment":"yes|no|
     unclear", "solver_facing_scope":"same|narrower|broader|unclear", "missing_obligation_ids
     ":[], "added_requirements":[], "coverage_risk":"yes|no|unclear", "functional_difference":"
     specific behavior this artifact asks a solver to repair, compared with the original
     Blueprint", "evidence":"short concrete evidence"},
    "B":{"public_issue_alignment":"yes|no|unclear", "declared_target_alignment":"yes|no|
     unclear", "solver_facing_scope":"same|narrower|broader|unclear", "missing_obligation_ids
     ":[], "added_requirements":[], "coverage_risk":"yes|no|unclear", "functional_difference
     ":"...", "evidence":"..."},
    "C":{"public_issue_alignment":"yes|no|unclear", "declared_target_alignment":"yes|no|
     unclear", "solver_facing_scope":"same|narrower|broader|unclear", "missing_obligation_ids
     ":[], "added_requirements":[], "coverage_risk":"yes|no|unclear", "functional_difference
     ":"...", "evidence":"..."}
  },
  "pairwise_functional_comparison":"what is materially the same or different across A/B/C,
     with obligation IDs and examples",
  "most_blueprint_faithful":"A|B|C|tie|unclear",
  "confidence":"high|medium|low"
}
Include each original obligation ID exactly once in obligation_matrix. Treat internal target
     revisions as evidence, not automatic proof of a simpler public task. If the supplied
     evidence cannot establish actual test coverage, coverage_risk must be unclear.
```

G.1 Independent construction review

The independent artifact-review prompt evaluates one construction at a time. It is used for construction review and the fixed-context relation diagnostic, while the side-by-side prompt above supplies the reported specification-preservation comparison.

Listing 18: Independent construction-review prompt.

```
You are an independent artifact reviewer. Compare one construction output with
the public problem, clean-source evidence, and, when present, the immutable
initial Blueprint. Do not reward verbosity or patch size. Direct has no initial
Blueprint, so its drift is null. Mutable may revise; Frozen must not.

Return exactly one JSON object:
{
  "artifact_complete": true,
  "semantic_coherence": true,
  "plausible_predecessor": true,
  "requirement_preservation_score": 0,
  "obligations_retained": 0,
  "obligations_added": 0,
  "obligations_removed": 0,
  "obligations_rewritten": 0,
  "drift": true,
  "retained_behavior_risk": "low|medium|high",
  "answer_leakage_risk": "low|medium|high",
  "expected_executable": true,
  "quality_score": 0,
  "failure_category": "none|abstained|invalid_patch|unsupported_semantics|
     incomplete_obligations|semantic_drift|collateral_damage|answer_leakage|
     insufficient_evidence",
  "evidence": ["short concrete reason"],
  "confidence": "high|medium|low"
}

Scores are integers 0..100. drift must be null for Direct. For Mutable/Frozen,
drift means the final target changed the selected problem rather than merely
using equivalent wording, organization, or an alternative correct edit.
```