SELF-IMPROVED INTENT ALIGNMENT VIA POST HOC GROUNDING FOR REPOSITORY GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Benefiting from increasingly capable large language models (LLMs), coding agents are able to tackle increasingly complex programming tasks. However, existing agents still face significant limitations in end-to-end repository generation. In particular, fine-grained implementation *intents* described in large-scale task specifications often lack sufficient salience, especially over long-horizon generation trajectories, leading to discrepancies between the generated repositories and the intended functionality. In this paper, we propose SIRA, a runtime augmentation framework that mitigates specification drift by re-grounding coding agents in task intents at file-creation checkpoints. SIRA combines *self-improved key intent extraction* with *retrieval-driven post hoc grounding*. It derives transferable extraction guidance from discrepancies observed between task specifications and their generated repositories in prior repository-generation tasks and uses this guidance to construct a compact index of behaviorally critical intents for the current task. As each new file is created, SIRA retrieves the file-relevant intents and injects them into the agent's context, enabling local inconsistencies to be identified and corrected before they propagate. We evaluate SIRA[1] on two representative repository-generation benchmarks, NL2Repo and RepoGenesis. Experimental results show that SIRA consistently achieves the best performance on the primary metrics, surpassing the strongest baseline by an average of 5.28 points in Score on NL2Repo and 9.63 points in Pass@1 on RepoGenesis.

1 INTRODUCTION

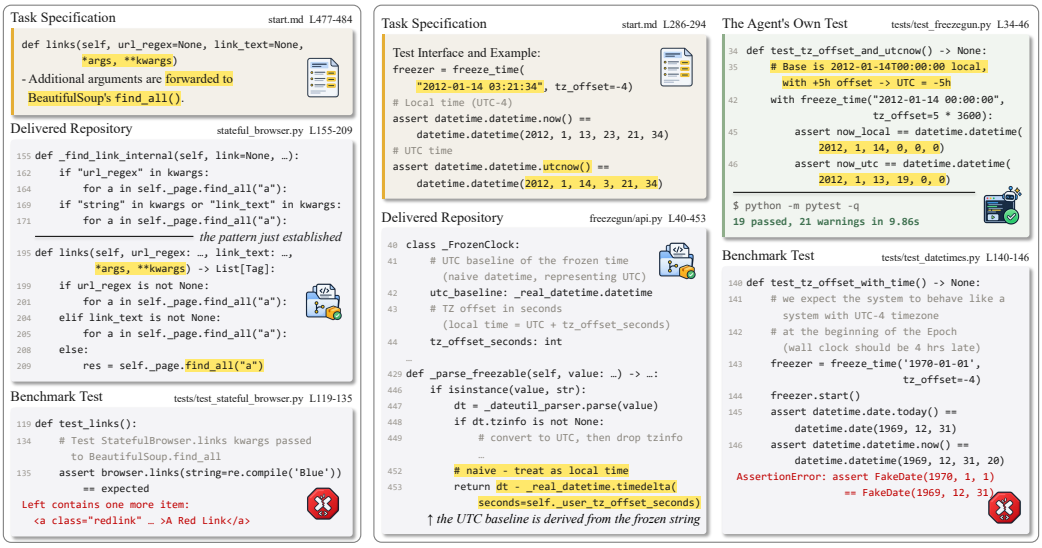
Recent advances in large language models (LLMs) have substantially improved their capabilities for software development, particularly in reasoning and planning over complex programming tasks. By coupling these models with external tools and persistent interaction loops, modern coding agents are able to autonomously carry out multi-step development processes (Yang et al., 2024; Xia et al., 2025). Such agents have demonstrated promising performance across a broad spectrum of software engineering tasks, including repository-level issue resolution (Pan et al., 2024; Li et al., 2025a), program repair (Ye et al., 2025; Bouzenia et al., 2025), test generation (Pan et al., 2025b; Chang et al., 2026), and environment configuration (Hu et al., 2025a; Bouzenia & Pradel, 2025).

Despite this broader progress, end-to-end repository generation remains a particularly challenging task, where an agent is given a natural-language specification and must construct a complete software repository from scratch (Zhang et al., 2026). Recent benchmarks (Ding et al., 2025; Peng et al., 2026; Li et al., 2026) have exposed a substantial gap in this capability, showing that repository specifications often encode numerous fine-grained and interdependent behavioral requirements that must be consistently realized across the entire codebase, while even advanced coding agents still struggle to produce functionally correct repositories (Luo et al., 2026).

In this paper, we focus on the problem of long-horizon generation from natural-language task specifications to deployable and executable code repositories. We use the term *intent* to denote a requirement that constrains an externally verifiable property of the delivered repository. To understand how existing systems fail to preserve such intents during repository generation, we run OpenHands (Wang et al., 2025a) with GPT-5 and analyze the delivered repositories. The tasks of each

[1] Code available at https://anonymous.4open.science/r/sira-agent/

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.



(a) mechanicalsoup (b) freezegun

Figure 1: Representative failures of intent alignment from completed OpenHands trajectories on NL2Repo-Bench analysis tasks using GPT-5. **(a)** The task specification requires the additional arguments of `links()` to be forwarded to `find_all()`, but the delivered method copies the lookup branches of the helper written immediately before it and never uses these arguments. **(b)** The implementation treats the frozen time as local time rather than UTC. The agent's self-written test encodes the same interpretation, allowing its own test suite to pass even though the benchmark test fails.

benchmark are divided into *analysis tasks* and *evaluation tasks* with no overlap. This study uses only analysis tasks, with all experimental details provided in Appendix A. Figure 1 illustrates two representative specification-alignment failures. Our analysis of benchmark test cases in Figure 2 shows that these are not isolated cases. Among failed test cases that reach the behavior under test, 42.3% on RepoGenesis and 62.9% on NL2Repo contradict behavior determined by the task specification.

Our investigation identifies several critical challenges inherent to this problem: 1) *Unequal criticality across specification intents.* Repository specifications mix routine implementation guidance with fine-grained behavioral contracts, such as exact values, boundary conditions, and output schemas. These details may be locally easy to substitute with plausible framework defaults or implementation conventions, despite directly determining externally observable behavior. In our motivation study, 73% and 65% of specification violations on RepoGenesis and NL2Repo respectively arise from convention adoption or specifically contradicting implementations rather than simple omission (Figure 2b). This motivates identifying intents whose later resurfacing is particularly important, rather than treating the specification as a uniform reminder target. 2) *Intent salience decay under long-horizon generation.* As implementation proceeds across files, agents must continue applying the original requirements alongside an expanding context of generated code. In Figure 1a, a method follows the lookup pattern of the helper written just before it and drops the argument forwarding that the specification states explicitly. Prior work also reports that LLMs can rely excessively on earlier answers and fail to recover from mistaken assumptions in multi-turn conversa-

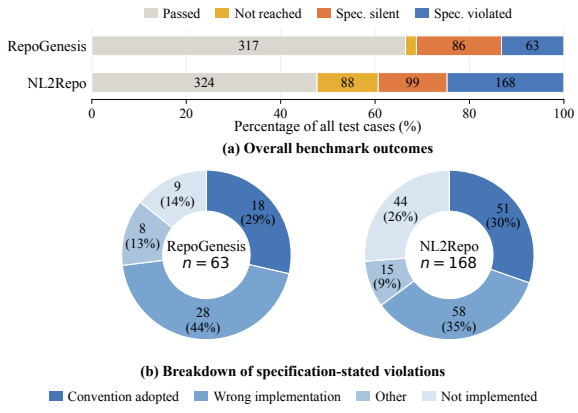


Figure 2: Analysis of benchmark test cases on the repositories delivered by OpenHands with GPT-5. **(a)** Classification of benchmark test-case outcomes. **(b)** Root causes of specification violations. Details are provided in Appendix A.

tions (Laban et al., 2026). This highlights the difficulty of preserving specification intent salience throughout long-horizon generation. 3) *Lack of independent specification-conformance feedback.* Unlike issue resolution (Yang et al., 2024) or program repair (Bouzenia et al., 2025), repository generation usually provides no externally authored executable oracle during construction. When an agent writes both code and tests, its tests can share mistaken assumptions or accept behavior that the specification excludes (Tan et al., 2026). Consequently, passing self-written tests does not ensure specification conformance (Figure 1b).

To address these challenges, we introduce **SIRA** (**S**elf-**I**mproved **R**etrieval-based Intent **A**lignment), a runtime augmentation framework that continuously grounds repository generation in the original task specification. SIRA is designed around two closely related objectives: identifying which specification intents warrant explicit resurfacing and restoring them when concrete implementation decisions can be checked against the specification. Accordingly, SIRA consists of two coordinated mechanisms. First, *self-improved key intent extraction* addresses unequal intent criticality by constructing a compact intent index, using cross-task experience from prior specification violations to prioritize behaviorally critical and repeatedly fragile requirements. Second, *retrieval-driven post hoc grounding* addresses intent salience decay and the lack of independent conformance feedback. Whenever a file is created, SIRA retrieves file-relevant intents and prompts the agent to inspect the concrete implementation against these retrieved intents. Because this grounding occurs only after file generation, it does not interfere with the agent's preceding generation actions, while still providing a local correction opportunity independent of the agent's self-written tests before deviations propagate across the repository. SIRA requires neither a separate verifier nor modifications to the underlying execution protocol. We evaluate SIRA on repository generation tasks from each of two representative repository-generation benchmarks, NL2Repo-Bench (Ding et al., 2025) and RepoGenesis (Peng et al., 2026), using two model backbones. SIRA achieves the highest functional correctness across all four benchmark–backbone settings, outperforming the strongest baseline in each setting by an average of 5.28 points in NL2Repo-Bench Score and 9.63 points in RepoGenesis Pass@1. Ablation studies support the contributions of experience-guided intent extraction and file-level post hoc grounding, suggesting that both the selection of intents to resurface and the timing of their delivery matter for repository generation.

2 METHOD

Problem Formulation. We formulate end-to-end repository generation as a sequential decision-making interaction process, in which a coding agent $\mathcal{A}$ iteratively interacts with a development environment $\mathcal{E}$ to construct a complete and executable repository from a natural-language task specification x. At step t, the agent selects an action a_t conditioned on its interaction history $h_t = (x, a_{<t}, o_{<t})$,

$$a_t \sim \mathcal{A}(\cdot \mid h_t), \qquad (s_t, o_t) = \mathcal{E}(s_{t-1}, a_t), \tag{1}$$

where s_t denotes the environment state after executing a_t, and o_t is the resulting observation. The interaction induces a trajectory

$$\tau_{\mathcal{A}}(x) = \big((a_t, o_t)\big)_{t=1}^{T}, \tag{2}$$

where T denotes the number of interaction steps. The final repository is obtained from the environment state after T interaction steps:

$$R_T = \mathrm{Repo}(s_T), \tag{3}$$

where $\mathrm{Repo}(\cdot)$ extracts the repository from the environment state. The objective is to produce a final repository R_T whose externally observable behavior satisfies the requirements specified in x, denoted by $R_T \models x$. Moreover, we use the term *intent* to denote an explicit requirement or constraint on an externally verifiable property of the delivered repository, including functional behavior, interface contracts, packaging, and deployability. Intents specify *what* the repository is expected to satisfy while remaining agnostic to *how* the corresponding behavior is internally implemented.

SIRA Overview. To improve the agent's alignment with these task intents, we propose SIRA, a runtime augmentation framework that operates without modifying the underlying coding agent. As illustrated in Figure 3, SIRA first extracts a compact set of key intents $\mathcal{I}_x$ from the task specification x using domain-specific cross-task experience g_d. During repository generation, whenever the agent creates a new source file f_t, SIRA retrieves the intents most relevant to that file and grounds

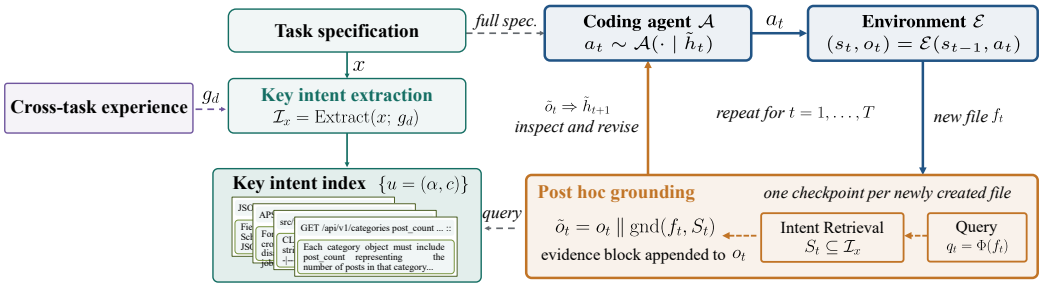


Figure 3: Overview of SIRA. Given a task specification x, SIRA first extracts a compact set of key intents $\mathcal{I}_x$ using domain-specific experience g_d. During repository generation, the creation of a new file f_t triggers post hoc grounding: SIRA retrieves file-relevant intents from $\mathcal{I}_x$ and augments the agent's context with them. The augmented observation is returned to the coding agent, which can inspect and revise the newly created file accordingly.

them into the current environment observation. The augmented observation is then returned to the agent, prompting it to inspect and revise the newly created file when necessary. We detail the intent extraction and post hoc grounding mechanisms in the following sections. All prompts used in our proposed method are provided in Appendix E.3 for reference.

2.1 Self-Improved Key Intent Extraction

Cross-Task Experience. Let $\mathcal{D}_d = \{(x^{(j)}, R^{(j)})\}_{j=1}^{m}$ denote completed tasks of domain d, each a task specification paired with its final repository. To accommodate repository generation tasks across different application domains, SIRA associates each domain d with extraction guidance g_d. Rather than being manually crafted, this guidance is derived from discrepancies observed in previously generated repositories and reused to improve intent extraction on subsequent tasks, which we refer to as *self-improvement*. During the first stage, we use an LLM to examine the specification $x^{(j)}$ together with the delivered repository $R^{(j)}$, producing a set of *findings* $\mathcal{F}^{(j)}$ for each task j. Each finding identifies an important task intent that is not adequately realized in the repository and is retained only if it is judged transferable across tasks within the same domain. The second stage aggregates the transferable findings across tasks within the same domain and consolidates them into domain-specific extraction guidance:

$$g_d = \mathcal{S}\left(\bigcup_{j=1}^{m} \mathcal{F}^{(j)}\right), \tag{4}$$

where $\mathcal{S}(\cdot)$ denotes the LLM-based summarization process that consolidates task-level findings into domain-specific extraction guidance. The resulting g_d consists of a compact set of directives that highlight intent categories warranting greater attention during extraction, together with the critical details that should be preserved. Each guidance g_d is applied only to tasks of domain d, since the kinds of intent that agents repeatedly fail to realize differ with the type of software being built. The resulting guidance for each domain is provided in Appendix G.

Intent Extraction. Given guidance g_d, SIRA condenses the task specification x into an intent index $\mathcal{I}_x$ using a single model call before repository generation begins, *i.e.*,

$$\mathcal{I}_x = \text{Extract}(x; g_d), \qquad u = (\alpha, c) \in \mathcal{I}_x, \tag{5}$$

where each *intent unit* u pairs a retrieval anchor α with a self-contained intent c. The anchor is used to facilitate reliable retrieval of the intent during the generation stage, thereby forming a dual counterpart to the file-side query in Section 2.2. Since retrieval must match specification text against code surface rather than against other text, each unit is anchored on the artifact that owns the requirement instead of on its prose, with a descriptive anchor used only when no such owner is named. Each intent describes only one semantically complete behavior. Meanwhile, all extracted units must be explicitly supported by the current specification.

The resulting index is deliberately non-exhaustive and is not intended to summarize x. Since the agent has already received the complete task specification, $\mathcal{I}_x$ instead retains requirements whose

later reminder is likely to prevent important properties from being omitted, partially realized, or implemented incorrectly. In particular, it prioritizes precise behavioral constraints, cross-component dependencies, and requirements whose relevance extends beyond a single local implementation context. This lightweight design aims to inject high-quality intent into the generation trajectory while preserving the agent's operational efficiency.

2.2 Retrieval-Driven Post Hoc Grounding

Post Hoc Checkpoint. SIRA restricts intent grounding to a single, precisely defined point in the trajectory, with the grounding checkpoint triggered upon the first successful creation of a supported file. This point is chosen because it is the earliest step at which two conditions are simultaneously satisfied: (1) an implementation decision has become externally observable in the form of a concrete file; and (2) any deviation introduced by that decision is still confined to a single file and has not yet propagated to other files subsequently developed based on it. Implementation details of the checkpoint trigger are provided in Appendix C.

Intent Retrieval. At each checkpoint, SIRA constructs a file-specific query $q_t = \Phi(f_t)$ from the newly created file. For source files, Φ captures a lightweight structural representation, including the file path and its declared interfaces (*e.g.*, imports, functions, and classes), while excluding implementation details such as function bodies and constants. This design makes the query reflect the responsibilities exposed by the file rather than its potentially misaligned implementation. For files without explicit program structure, such as configuration files, SIRA instead uses the file path together with a bounded content representation.

SIRA retrieves relevant intent units through complementary dense retrieval using C2LLM-0.5B (Qin et al., 2025) and lexical retrieval using BM25, a common hybrid retrieval implementation (Kalra et al., 2025; Liang et al., 2026; Kim et al., 2026). Given q_t, let $\mathcal{D}_t$ and $\mathcal{B}_t$ denote the candidate sets returned by dense and lexical retrieval, respectively, and let $r_D(u)$ and $r_B(u)$ denote the corresponding ranks of intent unit u. We combine the two rankings using reciprocal rank fusion:

$$s(u, q_t) = \frac{\mathbf{1}[u \in \mathcal{D}_t]}{\kappa + r_D(u)} + \frac{\mathbf{1}[u \in \mathcal{B}_t]}{\kappa + r_B(u)}, \tag{6}$$

where $\mathbf{1}[\cdot]$ denotes the indicator function, taking value 1 when u is retrieved by the corresponding route and 0 otherwise, and κ is the rank-fusion constant. Dense retrieval captures semantic correspondence between file responsibilities and behavioral requirements, while lexical retrieval preserves exact matches on identifiers and specification terms. SIRA further filters repeatedly surfaced intents and selects a small set of high-relevance candidates for grounding. If no sufficiently relevant intent is retrieved, no additional intent is injected, avoiding unnecessary interference with the agent's generation process. Further implementation details are provided in Appendix D.

Grounded Revision. The selected intents S_t, together with the path of the newly created file f_t, form a *grounding block* appended to the observation corresponding to its creation,

$$\tilde{o}_t = o_t \parallel \mathrm{gnd}(f_t, S_t), \tag{7}$$

where $\parallel$ denotes textual concatenation and $\mathrm{gnd}(\cdot)$ constructs the grounding block. The agent then continues from the augmented history $\tilde{h}_{t+1} = (x, a_{\le t}, \tilde{o}_{\le t})$. A fixed instruction added once to the system prompt directs the agent to assess the applicability of the retrieved intents, compare applicable intents against the implementation, and correct clear mismatches in the file or directly related files. Collectively, the grounding block and the instruction guide implementation review against specification intents and encourage revisions only when an applicable intent reveals a clear deviation, helping avoid unnecessary changes for intents that are irrelevant or already satisfied. Grounding thus adds a local self-correction step inside the original trajectory rather than a separate verification.

3 Experiment

3.1 Experimental Setup

Implementation. We implement SIRA on top of OpenHands (SDK v1.21.0), retaining the base agent's tool set, condenser, and execution protocol, and using the same backbone model for intent

Table 1: Evaluation results on RepoGenesis and NL2Repo-Bench. On RepoGenesis, Pass@1 is the pass rate of hidden API tests, DSR is the deployment success rate, and AC is the API coverage. On NL2Repo-Bench, Ave. Score is the average test pass rate. Cost denotes the average API cost (USD) per task. ↑ indicates higher is better, and ↓ indicates lower is better.

Model	Agent	RepoGenesis				NL2Repo-Bench	
		Pass@1 ↑	DSR ↑	AC ↑	Cost ($) ↓	Ave. Score ↑	Cost ($) ↓
GPT-5	Mini SWE Agent	61.75±0.81	86.36±0.00	95.83±0.66	0.43	29.78±0.42	1.37
	OpenCode	58.46±0.68	89.39±2.62	95.83±0.66	0.27	33.99±0.59	0.55
	OpenHands	67.43±1.09	95.45±0.00	95.45±0.00	0.40	31.74±0.53	0.72
	Codex	54.85±0.80	86.36±0.00	95.18±1.56	0.26	29.97±0.43	0.63
	Copilot	57.50±0.87	87.88±2.62	**96.59±0.00**	0.27	36.38±0.27	0.54
	CodeS	13.71±0.49	25.76±2.62	88.40±1.87	1.64	31.54±0.56	2.15
	SIRA	**79.18±0.98**	**98.48±2.62**	**96.59±0.00**	0.38	**41.84±0.44**	0.94
DeepSeek-V4-Pro	Mini SWE Agent	69.63±0.49	95.45±0.00	95.69±0.00	0.09	33.11±0.25	0.23
	OpenCode	65.23±0.98	96.97±2.62	93.86±0.00	0.14	35.39±0.53	0.24
	OpenHands	71.95±0.69	90.91±4.55	93.56±0.00	0.16	34.35±0.52	0.26
	Claude Code	72.92±0.94	**98.48±2.62**	**97.73±1.97**	0.10	34.21±0.52	0.28
	Copilot	65.79±0.79	95.45±0.00	96.21±0.66	0.18	38.63±0.47	0.31
	CodeS	20.04±0.62	27.27±4.55	74.00±2.12	0.13	28.82±0.36	0.28
	SIRA	**80.43±0.85**	**98.48±2.62**	95.41±2.04	0.16	**43.72±0.34**	0.29

extraction and repository generation. Intent extraction is a single LLM call before generation, and grounding reuses the agent's *observation* channel, so each task remains a single agent run. All experiments use two backbones, GPT-5 and DeepSeek-V4-Pro, with reasoning effort set to high. Except for cost, every reported number is the mean and sample standard deviation over three independent runs. Other experimental configurations are reported in Appendix E.

Datasets. We evaluate SIRA on two end-to-end repository-generation benchmarks, NL2Repo-Bench (Ding et al., 2025) and RepoGenesis (Peng et al., 2026). NL2Repo-Bench targets library-style Python repositories specified by long task documents, whereas RepoGenesis targets deployable web microservices verified by containerized deployment and hidden API tests. We evaluate on 22 NL2Repo-Bench tasks stratified over five application domains and on the 22 Python tasks of RepoGenesis, and accumulate extraction experience from tasks disjoint from these evaluation sets. Appendix E.4 describes the datasets in detail.

Baselines. We compare SIRA with six representative coding agents, *i.e.*, OpenHands (Wang et al., 2025a), mini-SWE-agent (Yang et al., 2024), OpenCode, Codex, Claude Code, and GitHub Copilot, as well as CodeS (Zan et al., 2026), a method designed specifically for repository generation. OpenHands shares the scaffold of SIRA, and the other agents test whether SIRA remains competitive beyond it. All systems receive the same public task specification, and have no access to hidden tests or human feedback.

3.2 Main Results

Table 1 reports the end-to-end results on both benchmarks. We first compare SIRA with OpenHands, which shares the same scaffold, tool set, and backbone, so that the difference reflects the runtime augmentation introduced by SIRA rather than differences in agent design. SIRA improves RepoGenesis Pass@1 by 11.75 and 8.48 points and NL2Repo-Bench Ave. Score by 10.10 and 9.37 points under GPT-5 and DeepSeek-V4-Pro, respectively. The consistent gains across benchmarks and backbones suggest that grounding key intents at file creation helps the agent correct deviations that it would otherwise carry into the final repository. These gains come at a comparable cost, as SIRA is no more expensive than OpenHands on RepoGenesis and adds at most 0.22 USD per task on NL2Repo-Bench.

Beyond its own scaffold, SIRA remains highly competitive with independently developed agents. It achieves the best Pass@1 and Ave. Score in all four settings and exceeds the strongest baseline by 11.75 and 7.51 Pass@1 points on RepoGenesis and by 5.46 and 5.09 points on NL2Repo-Bench. In contrast, no baseline leads consistently, with OpenHands and Claude Code leading RepoGenesis under the two backbones and Copilot leading NL2Repo-Bench. CodeS falls far behind the

Table 2: Ablation of the two design axes of SIRA with GPT-5. **Grounding Content** denotes the information provided to the agent, while **Point** specifies when grounding is applied: *file* after each file is generated, and *end* once at the end of repository generation. **Rel. Cost** denotes the per-task API cost relative to OpenHands, averaged over the two benchmarks.

Variant	Grounding Content	Point	RepoGenesis			NL2Repo-Bench	Rel. Cost
			Pass@1	DSR	AC	Ave. Score	
OpenHands	–	–	67.43±1.09	95.45±0.00	95.45±0.00	31.74±0.53	1.00×
What to ground? (grounding point fixed at file creation)							
w/o task-specific content	generic reminder	file	68.61±0.88	95.45±2.99	96.21±0.66	33.65±0.30	1.11×
w/o intent index	spec. chunks	file	66.08±0.94	92.42±2.62	93.86±0.26	34.81±1.10	1.34×
w/o experience	key intents	file	77.09±0.61	96.97±2.63	**96.59**±0.00	37.28±0.33	1.23×
When to ground? (grounding point moved to the end)							
deferred spec. review	full spec.	end	74.33±0.90	**98.48±2.62**	96.43±0.21	34.86±0.46	2.02×
deferred intent review	key intents	end	75.91±1.03	95.45±0.00	95.45±0.00	35.06±0.28	1.79×
SIRA	key intents	file	**79.18±0.98**	**98.48±2.62**	**96.59±0.00**	**41.84±0.44**	1.13×

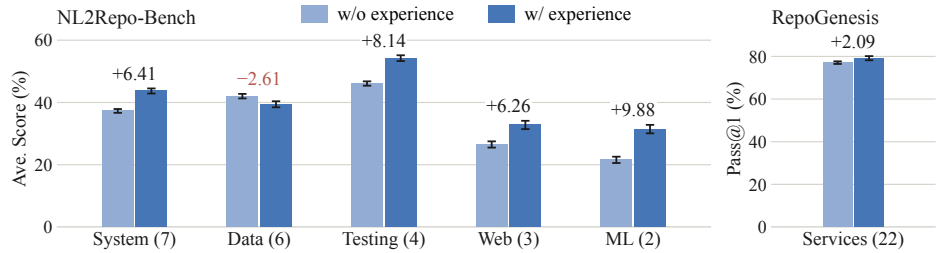


Figure 4: Per-domain effect of cross-task experience under GPT-5. Each group compares SIRA without (light) and with (dark) experience guidance. Bars are means over three runs, error bars are one standard deviation, and the number above each group gives the difference. Parentheses give the number of evaluation tasks.

other systems, especially on RepoGenesis. Its fixed workflow commits to a predefined generation pipeline without any verification, so it produces repositories that resemble the expected structure but cannot actually be deployed and run. This suggests that an elaborately designed workflow is less effective than a coding agent that can execute and revise its own code. Pass@1 and Ave. Score directly measure functional correctness and serve as the primary metrics, whereas DSR and AC are nearly saturated for most systems. The remaining differences in AC stem from its string-matching evaluation of endpoint declarations and do not reflect the functionality of the generated repositories.

3.3 Ablation Study

Table 2 examines grounding content and timing under GPT-5. SIRA achieves the highest functional correctness on both benchmarks with only 13% higher API cost than OpenHands on average. With grounding fixed at file creation, generic reminders yield only modest gains over OpenHands, while retrieving specification chunks produces mixed results despite higher cost. Removing experience guidance reduces Pass@1 by 2.09 points and Ave. Score by 4.56 points relative to SIRA. These results support selective intent extraction and indicate that cross-task experience improves the effectiveness of the resulting grounding content. Relocating grounding to the end requires adapting its delivery (See Appendix F for details). Delivering the same key intents at the end trails SIRA by 3.27 Pass@1 points and 6.78 Ave. Score points while raising relative cost from 1.13× to 1.79×. Reviewing the full specification at the end also underperforms SIRA at 2.02× the baseline cost. These results suggest that grounding intents at file creation is more effective and token-efficient than postponing review until repository completion.

3.4 Cross-Task Self-Improvement

We further examine whether the gain from experience guidance holds consistently across domains. Figure 4 compares SIRA with and without experience guidance on the five NL2Repo-Bench do-

Two adjacent response contracts in the task specification

```
GET /api/v1/files        → files, page, page_size, total, total_pages
GET /api/v1/files/search → files, query, page, page_size, total
```

Intent grounded into the creation observation of `app/main.py` (excerpt)

"... response must include files, **query (echoing q)**, page, page_size, and total."

At file creation

```
@app.get(..., response_model=
      FileListResponse)
async def search_files(q, ...):
  ...
  return FileListResponse(
    files=files,
    page=page, page_size=page_size,
    total=total,
    total_pages=total_pages(...))
```

After grounding

```
class SearchResponse(BaseModel):
  files:  List[FileItem]
  query:  str  ...
@app.get(..., response_model=
      SearchResponse)
  return SearchResponse(
    files=files, query=q,
    page=page, page_size=page_size,
    total=total)
```

Figure 5: Post hoc grounding is followed by a targeted contract repair. File_Relay under GPT-5. The search endpoint first reuses the file-list response model and drops the `query` field required by the specification, and is revised to echo the query after the corresponding intent is surfaced at file creation.

mains and on RepoGenesis under GPT-5. Experience improves five of the six domains, and every gain exceeds the run-to-run standard deviation of both conditions. Testing and Machine Learning gain 8.14 and 9.88 points, and the improvement is not confined to library-style repositories, since RepoGenesis, whose services are verified through deployment and hidden API tests, gains 2.09 Pass@1 points. The only decline is Data Analysis & Processing with 2.61 points. It is driven by two of the six tasks in the domain, one of which declares an incomplete dependency set so that its test module fails to import and none of its 172 test cases runs, while the remaining four tasks improve on average. These results indicate that guidance derived from prior tasks of a domain transfers to unseen tasks of the same domain, and that this transfer holds in most of the evaluated domains.

3.5 Agent Response to Grounding

We examine how the agent responds to each grounding checkpoint in the GPT-5 runs, considering its actions after the grounding block becomes visible and before the first subsequent project execution, so that the observed revisions precede runtime or test feedback. Detailed classification criteria are provided in Appendix J. As shown in Table 3, 20.1% of RepoGenesis checkpoints and 17.8% of NL2Repo-Bench checkpoints are followed by a qualifying revision within this pre-execution window. Among these revisions, more than a third occur on a related file rather than the file that triggered the checkpoint, showing that responses to grounding can extend beyond the newly created file. Most checkpoints produce no qualifying revision, which is consistent with the grounding policy instructing the agent to make no change when a retrieved intent is irrelevant or already satisfied. Overall, grounding leaves most of the immediate generation trajectory unchanged while providing an early opportunity for revisions to the target file or related components before runtime feedback becomes available. Furthermore, Figure 5 shows such a revision restoring the behavior required by the specification.

Table 3: Agent reactions to grounding checkpoints under GPT-5, in percent of 174 RepoGenesis and 354 NL2Repo-Bench checkpoints.

Reaction	RepoGenesis	NL2Repo
Revised	20.1%	17.8%
target file	12.6%	9.3%
related file	7.5%	8.5%
No revision	79.9%	82.2%

3.6 Case Study

Figure 5 shows a checkpoint from File_Relay under GPT-5. The specification gives the search endpoint its own response schema, which omits `total_pages` and adds a `query` field echoing the submitted term. When creating `app/main.py`, the agent reuses the listing response model for search and drops `query`, the same pattern-following failure as in Figure 1a. After SIRA grounds the corresponding intent, the agent defines a dedicated `SearchResponse` model in the related file

`app/schemas.py` and revises the endpoint to return `query=q`. The service has not been executed at this point, so the revision cannot come from runtime or test feedback. This case illustrates how post hoc grounding can recover specification-defined behavior from a misaligned implementation before the deviation propagates further. The full trace is given in Appendix K.

4 Related Work

LLM-based Coding Agents. Modern coding agents pair an LLM with an executable environment and operate on a codebase through repeated action and observation loops (Wang et al., 2024). Driven largely by issue-resolution benchmarks and efforts to scale their construction (Jimenez et al., 2024), a substantial body of work has sought to make this loop more effective through interfaces between agents and computers and open scaffolds (Yang et al., 2024; Wang et al., 2025a), structured localize, repair, and validate pipelines that remove autonomous control flow (Xia et al., 2025), multi-agent role decomposition (Hong et al., 2024; Qian et al., 2024), inference-time search over candidate trajectories (Antoniades et al., 2025), and large-scale trajectory synthesis for training agent models (Pan et al., 2025a; Yang et al., 2025). A parallel line specializes agents for individual stages of the development pipeline, including program repair (Zhang et al., 2024; Bouzenia et al., 2025), test generation (Ahmed et al., 2025), and environment configuration (Hu et al., 2025a). These settings rely on an executable corrective signal, such as a failing test or a bug report, which repository generation lacks, and self-authored tests share the implementation's reading of the specification (Olausson et al., 2024; Huang et al., 2024). SIRA instead re-anchors the agent to its own specification once implementation decisions become observable, without altering the agent or adding an external verifier.

Repository-Level Code Generation. Repository-level code generation was initially studied over an *existing* codebase, where repository context supports completion, feature implementation, or coordinated multi-file edits (Liu et al., 2024; Wu et al., 2024; Bairi et al., 2024; Wang et al., 2025b; Li et al., 2025b; Le Hai et al., 2025; Liang et al., 2025; Li et al., 2024; Wang et al., 2026b). Recent work further studies how to select repository context efficiently for generation (Wang et al., 2026a). Generating an entire repository from a natural-language specification is a more recent and considerably harder setting. Existing methods are predominantly *plan-first*, compiling the specification into an intermediate plan and then realizing it file by file (Zan et al., 2026; Hu et al., 2025b; Luo et al., 2026). Benchmarks for this setting (Liu et al., 2025; Zhao et al., 2025; Ding et al., 2025; Peng et al., 2026; Li et al., 2026; Liu et al., 2026) show that even strong agents remain far from producing functionally correct repositories, and failures concentrate in the fine-grained behavior prescribed by the specification rather than in producing a runnable artifact (Ding et al., 2025; Peng et al., 2026). SIRA is complementary to this line of work. Unlike retrieval that provides *code* to condition generation (Le Hai et al., 2025; Liang et al., 2025; Wang et al., 2026a), it retrieves *specification intents* to audit written code. Unlike plan-first methods, it leaves the agent's plan untouched and resurfaces intents post hoc while any deviation is still local.

5 Conclusion

In this paper, we presented SIRA, a runtime augmentation framework that keeps coding agents aligned with task intents during end-to-end repository generation. SIRA extracts a compact index of key intents under guidance derived from prior specification violations, and grounds file-relevant intents into the agent's observation whenever a new file is created. Compared with six representative coding agents and the dedicated repository-generation method CodeS, SIRA achieves the best functional correctness across all four benchmark–backbone settings. These gains are achieved at a computational cost comparable to that of the base agent, and ablations attribute them to both the selection of intents and the timing of grounding. These results suggest that keeping coding agents anchored to the task specification throughout the generation process is an effective approach that complements stronger planning and verification. In future work, we will explore more precise intent-to-code retrieval and explicit violation detection to further improve intent grounding, and extend SIRA to other programming languages.

Reproducibility Statement

To support reproducibility, Section 2 describes the SIRA framework and its main components, and Section 3 specifies the experimental setup and evaluation protocol. Appendices B–D provide the algorithmic procedure, checkpoint implementation, and retrieval details; Appendix E reports the hardware and software environment, baseline configurations, prompts, dataset partitioning, and backbone selection; Appendix F documents the ablation implementations; and Appendix G provides the cross-task extraction guidance used in the experiments. Anonymous source code is available through the repository linked on the first page.

AI Use Statement

Generative AI tools were used solely to aid language editing and polish the manuscript for clarity and readability. They were not used for literature retrieval or discovery, research ideation or execution, drafting substantive sections of the paper, generating synthetic datasets, or proving mathematical claims. All AI-assisted edits were reviewed by the authors, who take full responsibility for the final content of the paper.

References

Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. Otter: Generating tests from issues to validate SWE patches. In *International Conference on Machine Learning (ICML)*, 2025.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Yang Wang. SWE-Search: Enhancing software agents with Monte Carlo tree search and iterative refinement. In *International Conference on Learning Representations (ICLR)*, 2025.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. CodePlan: Repository-level coding using LLMs and planning. In *Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE)*, 2024.

Islem Bouzenia and Michael Pradel. You name it, i run it: An llm agent to execute tests of arbitrary projects. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1054–1076, 2025.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. RepairAgent: An autonomous, LLM-based agent for program repair. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2025.

Pengyu Chang, Yixiong Fang, Silin Chen, Yuling Shi, Beijun Shen, and Xiaodong Gu. Test vs mutant: Adversarial llm agents for robust unit test generation. *arXiv preprint arXiv:2602.08146*, 2026.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, et al. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*, 2025.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations (ICLR)*, 2024.

Ruida Hu, Chao Peng, Xinchen Wang, and Cuiyun Gao. Repo2Run: Automated building executable environment for code repository at scale. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025a.

Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. Self-evolving multi-agent collaboration networks for software development. In *International Conference on Learning Representations (ICLR)*, 2025b.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations (ICLR)*, 2024.

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 (ICLR)*, 2024.

Jushaan Singh Kalra, Xinran Zhao, To Eun Kim, Fengyu Cai, Fernando Diaz, and Tongshuang Wu. Mor: Better handling diverse queries with a mixture of sparse, dense, and human retrievers. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 11982–12001, 2025.

Joeun Kim, Seunghyouk Yoon, Xuan-Bach Le, Youngeun Nam, Doyoung Kim, Hwanjun Song, and Jae-Gil Lee. Qudar: Query-wise dual-perspective adaptive retrieval. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 38662–38679, 2026.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. LLMs get lost in multi-turn conversation. In *International Conference on Learning Representations (ICLR)*, 2026.

Nam Le Hai, Dung Manh Nguyen, and Nghi DQ Bui. On the impacts of contexts on repository-level code generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1496–1524, 2025.

Han Li, Yuling Shi, Shaoxin Lin, Xiaodong Gu, Heng Lian, Xin Wang, Yantao Jia, Tao Huang, and Qianxiang Wang. Swe-debate: Competitive multi-agent debate for software issue resolution. *arXiv preprint arXiv:2507.23348*, 2025a.

Jia Li, Ge Li, Xuanming Zhang, Yunfei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. Evocodebench: An evolving code generation benchmark with domain-specific evaluations. *Advances in Neural Information Processing Systems*, 37:57619–57641, 2024.

Jia Li, Hongyi Deng, Yiran Zhang, Kechi Zhang, Tianqi Shao, Tiankuo Zhao, Weinan Wang, Zhi Jin, Ge Li, Yang Liu, et al. Realbench: A repo-level code generation benchmark aligned with real-world software development practices. *Proceedings of the ACM on Software Engineering*, 3 (FSE):1735–1756, 2026.

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 17160–17176, 2025b.

Shanchao Liang, Nan Jiang, Lin Tan, et al. Can language models replace programmers for coding? repocod says 'not yet'. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 24698–24717, 2025.

Xiaojian Liang, Lushan Song, Shishuai Du, Weicheng Zhu, Tan Faith, Jun Jie Sim, Haibing Jin, Zhenghao Wu, Yingting Liu, Xin Zhang, et al. Pisces: Cryptography-based private retrieval-augmented generation with dual-path retrieval. In *International Conference on Learning Representations*, volume 2026, pp. 61885–61904, 2026.

Jingjing Liu, Silin Li, Zeming Liu, Zihao Cheng, Yuhang Guo, Yuanfang Guo, Yunhong Wang, and Haifeng Wang. Towards multi-language repository-level code generation: From-scratch to guided tasks. *Neurocomputing*, pp. 133204, 2026.

Kaiyuan Liu, Youcheng Pan, Yang Xiang, Daojing He, Jing Li, Yexing Du, and Tianrun Gao. ProjectEval: A benchmark for programming agents automated evaluation on project-level code generation. In *Findings of the Association for Computational Linguistics (ACL Findings)*, 2025.

Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations (ICLR)*, 2024.

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 (ICLR)*, 2026.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *International Conference on Learning Representations (ICLR)*, 2024.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*, 2024.

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 *International Conference on Machine Learning (ICML)*, 2025a.

Rangeet Pan, Raju Pavuluri, Ruikai Huang, Rahul Krishna, Tyler Stennett, Alessandro Orso, and Saurabh SInha. Saint: Service-level integration test generation with program analysis and llm-based agents. *arXiv preprint arXiv:2511.13305*, 2025b.

Zhiyuan Peng, Xin Yin, Pu Zhao, Fangkai Yang, Lu Wang, Ran Jia, Xu Chen, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Repogenesis: Benchmarking end-to-end microservice generation from readme to repository. *arXiv preprint arXiv:2601.13943*, 2026.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

Jin Qin, Zihan Liao, Ziyin Zhang, Hang Yu, Peng Di, and Rui Wang. C2llm technical report: A new frontier in code retrieval via adaptive cross-attention pooling. *CoRR*, abs/2512.21332, 2025. doi: 10.48550/ARXIV.2512.21332. URL https://doi.org/10.48550/arXiv.2512.21332.

Yuhao Tan, Zhibang Yang, Fangkai Yang, Yuan Yao, Yu Kang, Lu Wang, Pu Zhao, Xin Zhang, Xiaoxing Ma, Qingwei Lin, et al. Beyond fail-to-pass: Iterative hardening of co-generated bug reproduction tests and fixes. *arXiv preprint arXiv:2607.19843*, 2026.

Peiding Wang, Li Zhang, and Fang Liu. Mrcoder: An efficient context selecting approach for repository-level code generation. *arXiv preprint arXiv:2607.26805*, 2026a.

Peiding Wang, Li Zhang, Fang Liu, Taichuan Li, and Yinghao Zhu. Codespec: Dual executable specifications for agentic long-horizon feature development. *arXiv preprint arXiv:2607.26777*, 2026b.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *International Conference on Machine Learning (ICML)*, 2024.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025a.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. RLCoder: Reinforcement learning for repository-level code completion. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2025b.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. Repoformer: Selective retrieval for repository-level code completion. In *International Conference on Machine Learning (ICML)*, 2024.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. In *Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE)*, 2025.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

John Yang, Kilian Lieret, Carlos E. 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 (NeurIPS), Datasets and Benchmarks Track*, 2025.

He Ye, Aidan ZH Yang, Chang Hu, Yanlin Wang, Tao Zhang, and Claire Le Goues. Adverintent-agent: Adversarial reasoning for repair based on inferred program intent. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1398–1420, 2025.

Daoguang Zan, Ailun Yu, Wei Liu, Dong Chen, Bo Shen, Yafen Yao, Wei Li, Xiaolin Chen, Yongshun Gong, Bei Guan, et al. Codes: Natural language to code repository via multi-layer sketch. *ACM Transactions on Software Engineering and Methodology*, 35(7):1–26, 2026.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024.

Zhaoxi Zhang, Yiming Xu, Jiahui Liang, Weikang Li, Xiaoshuai Chen, Liwei Qian, Xin Pei, Jizhou Huang, Run Sun, and Yunfang Wu. Repozero: Can llms generate a code repository from scratch? *arXiv preprint arXiv:2605.07122*, 2026.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T. Chiu, Claire Cardie, Matthias Gallé, and Alexander M. Rush. Commit0: Library generation from scratch. In *International Conference on Learning Representations (ICLR)*, 2025.

A MOTIVATION STUDY

We conduct a preliminary experiment with OpenHands and GPT-5 on analysis tasks to explore the main causes of failure when existing coding agents generate repositories end to end. We analyze the benchmark test cases executed on the delivered repositories. For each failed test case, we examine whether the failure reaches the behavior under test and whether the task specification clearly defines the expected behavior. For failures that contradict the specification, we further inspect the delivered code to characterize how the implementation deviates from the stated requirement.

A.1 SETUP

We run OpenHands with GPT-5 on each task and collect the delivered repositories together with their evaluation logs. All tasks in this study are analysis tasks, which have no overlap with the evaluation tasks used in Section 3. On RepoGenesis, we sample 30 of the 38 Python analysis tasks, which contain 477 test cases. On NL2Repo, we select 10 of them stratified by difficulty, namely three Easy tasks (autorccar, coverage_shield, and trimming), five Medium tasks (arguably, flasky, freezegun, mechanicalsoup, and stamina), and two Hard tasks (synthetic and unidecode). These tasks cover all five application domains and contain 679 test cases. A benchmark test case, hereafter a test case, corresponds to one pytest node, and each instance of a parametrized test is counted separately.

All labels are assigned through manual analysis, using the task specification, the source code of the test case, the failure output of the benchmark harness, and the delivered repository as evidence. The available evidence differs between the two benchmarks. On RepoGenesis, the benchmark test suite is available, so whether a test case is determined by the specification is judged from the specification and the test source alone, without the failure output. On NL2Repo, the harness reports only failed test cases, so only these are annotated and passed test cases are counted from the evaluation report.

A.2 CLASSIFICATION OF BENCHMARK TEST CASES

Each test case is assigned to exactly one of four categories, which are decided in the order below.

- **Passed.** The test case passes on the delivered repository.
- **Not reached.** The test case fails before the behavior under test is exercised, for example due to startup, import, collection, or fixture errors.
- **Spec. silent.** The behavior under test is exercised and the test case fails, but the specification does not determine the expected outcome, e.g., due to an omission or ambiguity in the specification (Ding et al., 2025).
- **Spec. violated.** The behavior under test is exercised and the test case fails, and the expected outcome is determined by the specification, whether stated directly or implied by its general requirements.

A.3 ROOT CAUSES OF SPECIFICATION VIOLATIONS

For Spec. violated test cases, we further inspect the delivered repository to identify the underlying cause of each violation. The classification distinguishes whether the governed functionality is absent, follows an otherwise plausible implementation convention that conflicts with a more specific requirement, or contains requirement-specific logic that implements the requirement incorrectly.

- **Not implemented.** No code in the repository handles the situation that the requirement governs.
- **Convention adopted.** The relevant functionality is implemented using a plausible default, common convention, or locally natural implementation pattern, but this choice conflicts with a more specific requirement in the specification. Satisfying the requirement typically requires only a localized adjustment or override to the otherwise conventional implementation, as illustrated in Figure 1a.
- **Wrong implementation.** The implementation contains logic specifically intended to realize the stated requirement, but that logic encodes behavior that contradicts the specifi-

Table 4: Analysis of benchmark test cases. The upper block reports the number of test cases and their shares of all test cases. The lower block reports the number of Spec. violated test cases by root cause and their shares of all Spec. violated test cases.

	RepoGenesis	NL2Repo
All test cases	477	679
Passed	317 (66.5%)	324 (47.7%)
Not reached	11 (2.3%)	88 (13.0%)
Spec. silent	86 (18.0%)	99 (14.6%)
Spec. violated	63 (13.2%)	168 (24.7%)
Not implemented	9 (14.3%)	44 (26.2%)
Convention adopted	18 (28.6%)	51 (30.4%)
Wrong implementation	28 (44.4%)	58 (34.5%)
Other	8 (12.7%)	15 (8.9%)

cation. Satisfying the requirement therefore requires correcting the requirement-specific implementation, as illustrated in Figure 1b.

- **Other.** None of the above holds after inspecting the source code. In most cases, the requirement-specific logic is implemented correctly, but execution fails due to an unrelated defect before reaching the relevant implementation branch.

A.4 Results and Analysis

Table 4 shows that OpenHands passes 66.5% of the test cases on RepoGenesis and 47.7% on NL2Repo. Among all test cases, 2.3% and 13.0%, respectively, fail before reaching the behavior under test, while 18.0% and 14.6% fail after reaching the tested behavior but concern outcomes that are not determined by the specification. Distinguishing these cases from specification violations shows that violations account for a substantial part of functional failures. Among failed test cases that reach the behavior under test, 42.3% on RepoGenesis and 62.9% on NL2Repo contradict behavior determined by the task specification. Thus, even when execution reaches the relevant behavior, a considerable fraction of failures reflect discrepancies between the delivered implementation and the specified requirements.

The root-cause analysis further shows that most specification violations are not caused by complete omission of the governed functionality. Not implemented accounts for only 14.3% of specification violations on RepoGenesis and 26.2% on NL2Repo. In contrast, convention adopted and wrong implementation together account for 73.0% and 64.9%, respectively. In these cases, the repository contains an implementation for the relevant behavior, but the concrete implementation choice conflicts with the stated requirement. The former reflects locally plausible defaults, conventions, or surrounding implementation patterns that override a more specific task constraint, whereas the latter reflects requirement-specific logic that realizes the intended behavior incorrectly.

Although the share of specification violations differs between the two benchmarks, the distribution of their root causes is consistent across these different repository-generation settings. On both benchmarks, wrong implementation is the most frequent root cause and convention adopted is the second. Taken together, they show that a substantial fraction of specification violations occur not because the relevant functionality is absent, but because an existing implementation realizes the specified behavior incorrectly. In such cases, the key discrepancy is therefore between the concrete implementation decisions made by the agent and the behavior required by the task specification. This observation motivates us to investigate how repository-generation agents can better preserve specification-defined requirements throughout the generation process.

A.5 Threats to Validity

The analysis relies on manual inspection and may therefore be affected by subjective judgment, particularly in borderline cases where the specification or observed failure admits multiple interpretations. The reported proportions should therefore be interpreted as descriptive evidence rather than exact estimates of failure prevalence.

B ALGORITHM DETAILS

Algorithm 1 presents the complete procedure of SIRA, which consists of three stages, namely cross-task experience induction, key intent extraction, and repository generation with post hoc grounding.

Lines 1 to 3 derive the extraction guidance g_d for domain d (Section 2.1). $\mathrm{Analyze}(\cdot)$ denotes the LLM-based analysis that compares a completed specification with its delivered repository and retains transferable findings, and $\mathcal{S}(\cdot)$ consolidates these findings into g_d. This stage is executed once per domain on its analysis tasks, which are disjoint from the evaluation tasks, and g_d is shared by all tasks of that domain. Line 4 then constructs the intent index $\mathcal{I}_x$ with a single model call before generation begins.

Lines 5 to 17 retain the interaction loop of the base agent and modify only the observation returned at a grounding checkpoint (Section 2.2). The set $\mathcal{V}$ records files that have already triggered a checkpoint, so each file is grounded at most once. $\mathrm{Trigger}(\cdot)$ accepts a newly created file only if its suffix or name indicates executable or deployment semantics and its content contains implementation. For each accepted file, $\mathrm{Dense}(\cdot)$ and $\mathrm{BM25}(\cdot)$ rank the intent units against the file-side query q_t, and $\mathrm{RRF}(\cdot)$ fuses the two rankings. The counter $n(u)$ records the number of injections of unit u within the task, and units that reach the cap $N_{\max}$ are removed from the candidate set C_t. $\mathrm{TopK}(\cdot)$ returns the top $k(f_t)$ units from the remaining fused ranking, where $L(f_t)$ is the line count of f_t, ℓ is the number of lines that entitles a file to one additional unit, and $k_{\max}$ bounds the number of units per checkpoint. All experiments use $N_{\max} = 4$, $k_{\max} = 5$, and $\ell = 100$. If the remaining ranking is empty, the observation is returned unchanged. The same holds for all other observations, including those produced by later edits to existing files.

Algorithm 1 SIRA

Input: specification x of domain d, completed tasks $\mathcal{D}_d$, coding agent $\mathcal{A}$, environment $\mathcal{E}$ with initial state s_0
Output: repository R_T

Cross-task experience (once per domain)
1: **for** each $(x^{(j)}, R^{(j)}) \in \mathcal{D}_d$ **do**
2: $\quad \mathcal{F}^{(j)} \leftarrow \mathrm{Analyze}(x^{(j)}, R^{(j)})$ ▷ retain transferable findings
3: $g_d \leftarrow \mathcal{S}\big(\bigcup_j \mathcal{F}^{(j)}\big)$
Key intent extraction
4: $\mathcal{I}_x \leftarrow \mathrm{Extract}(x; g_d)$ ▷ single model call
Repository generation with post hoc grounding
5: add the fixed grounding instruction to the system prompt of $\mathcal{A}$
6: $n(u) \leftarrow 0$ for all $u \in \mathcal{I}_x$, $\quad \mathcal{V} \leftarrow \emptyset$, $\quad \tilde{h}_1 \leftarrow (x)$, $\quad t \leftarrow 1$
7: **repeat**
8: $\quad a_t \sim \mathcal{A}(\cdot \mid \tilde{h}_t)$, $\quad (s_t, o_t) \leftarrow \mathcal{E}(s_{t-1}, a_t)$, $\quad \tilde{o}_t \leftarrow o_t$
9: $\quad$ **if** a_t creates a file $f_t \notin \mathcal{V}$ **and** $\mathrm{Trigger}(f_t)$ **then**
10: $\quad\quad \mathcal{V} \leftarrow \mathcal{V} \cup \{f_t\}$
11: $\quad\quad q_t \leftarrow \Phi(f_t)$
12: $\quad\quad C_t \leftarrow \mathrm{RRF}\big(\mathrm{Dense}(q_t, \mathcal{I}_x), \mathrm{BM25}(q_t, \mathcal{I}_x)\big) \setminus \{u \mid n(u) \geq N_{\max}\}$
13: $\quad\quad S_t \leftarrow \mathrm{TopK}\big(C_t, k(f_t)\big)$, $\quad k(f_t) = \min\big(k_{\max}, \lceil L(f_t)/\ell \rceil\big)$
14: $\quad\quad$ **if** $S_t \neq \emptyset$ **then**
15: $\quad\quad\quad \tilde{o}_t \leftarrow o_t \,\|\, \mathrm{gnd}(f_t, S_t)$, $\quad n(u) \leftarrow n(u) + 1$ for all $u \in S_t$
16: $\quad \tilde{h}_{t+1} \leftarrow (\tilde{h}_t, a_t, \tilde{o}_t)$, $\quad t \leftarrow t + 1$
17: **until** a_{t-1} is the finish action
18: **return** $R_T = \mathrm{Repo}(s_T)$ with $T = t - 1$

C CHECKPOINT TRIGGER

A grounding checkpoint is triggered upon observing a file editing action only if the action successfully creates a path that did not previously exist. Each path therefore triggers at most once, and later edits of the same file never trigger again, so the number of checkpoints is bounded by the number of newly created files rather than by the length of the trajectory. Before retrieval, the newly created file is subject to two local filters. The *type* filter matches the created path against a fixed whitelist of suffixes and file names associated with executable or deployment semantics, such as Python source files and Dockerfiles, excluding documentation and data files from further processing. The *content* filter

then removes files that contain no implementation: a Python module whose syntax tree is empty after removing a leading docstring, as in an empty package initializer, or a script or configuration file whose content is blank.

Two cases are handled separately. Files without program structure, such as configuration files, skip structural analysis and are represented by their path together with a bounded prefix of their content. Files materialized outside the editing tool, for example through shell redirection, produce no checkpoint, which is a coverage boundary of the current implementation.

D Retrieval Details and Example

At each grounding checkpoint, SIRA maps the newly created file f_t to the selected intents S_t in three steps, namely query construction, hybrid ranking over the intent index $\mathcal{I}_x$, and budgeted selection. Ranking compares an indexed passage of each intent unit with a query rendered from the file, and the templates of both texts are shown below. The parameter values are listed in Table 5 and are fixed across both benchmarks and both backbones.

Intent unit $u = (\alpha, c)$

Indexed passage

```
Implementation responsibility: <α>
Task intent: <c>
```

Delivered text

```
<α> :: <c>
```

File-side query $q_t = \Phi(f_t)$

```
Created file: <relative path>
Imports:
<one imported module per line>
Declarations:
<one declaration per line>
```

Intent Index. Each intent unit $u = (\alpha, c) \in \mathcal{I}_x$ is indexed as a single passage, so the retrieval corpus consists exactly of the extracted units and the specification itself is never chunked. The passage renders the anchor and the intent as two labelled fields. This representation states each unit as the responsibility of an implementation artifact and matches the form of the file-side query. A selected unit is instead delivered to the agent in the original form produced by extraction.

File-Side Query. For a Python file, $\Phi(f_t)$ parses the syntax tree and fills the query template. The declaration block contains public top-level functions and classes together with all methods of public classes. Each entry is rendered as its signature with decorators inlined on the same line, so route paths and dependency injections stay attached to the functions they qualify, while function bodies, constants, and comments are omitted. Entries are deduplicated in source order, and the rendered query is truncated to a maximum length. Shell scripts and configuration files have no parsed structure, so their query consists of the file path followed by a bounded prefix of the file content.

Hybrid Ranking. The dense route follows the prompt format of C2LLM, with the file-side query encoded using the query-side instruction *Given a newly created code file, retrieve the task requirements that the file should satisfy*, while each indexed passage is encoded with the document-side prefix *Retrieved Answer:*. Units are ranked by cosine similarity, and units below a minimum cosine score are discarded. The lexical route ranks units by BM25 and retains only units with a positive score. Its tokenizer splits identifiers at underscores, case boundaries, and path separators while keeping the whole identifier, so a declaration such as `delete_stored_file` also matches specification text that contains only `delete` or `stored`. Each route returns a bounded number of candidates, and the two rankings are combined by the reciprocal rank fusion in Section 2.2.

Budgeted Selection. SIRA maintains a counter $n(u)$ of the injections of each unit within the current task and removes units with $n(u) \geq N_{\max}$ from the fused ranking, which prevents a broadly worded unit from being injected at every checkpoint. The top $k(f_t)$ remaining units form S_t, where

$$k(f_t) = \min\big(k_{\max}, \lceil L(f_t)/\ell \rceil\big), \tag{8}$$

$L(f_t)$ is the number of lines of f_t, ℓ is the number of lines that entitles a file to one additional unit, and $k_{\max}$ bounds the number of units per checkpoint. A longer file thus implements more behavior and is checked against more intents. If no unit remains, the observation is returned unchanged and the counters are not updated.

Table 5: Retrieval and selection parameters of SIRA.

Stage	Parameter	Value
Query construction	Content prefix for files without structure	2,000 characters
Dense route	Embedding model	C2LLM-0.5B
	Minimum cosine score	0.4
	Maximum candidates	40
Lexical route	BM25 k_1, b	1.2, 0.75
	Maximum candidates	40
Fusion	Rank fusion constant κ	60
Selection	Lines per additional unit ℓ	100
	Units per checkpoint $k_{\max}$	5
	Injections per unit within a task $N_{\max}$	4

Example. We illustrate how intent anchors are paired with the file-side query using a checkpoint from the NL2Repo-Bench task math-verify under GPT-5, whose intent index contains 14 units. The agent creates `src/math_verify/parser.py` with 329 lines, so $k(f_t) = \min(5, \lceil 329/100 \rceil) = 4$ units are selected. The left panel shows the query rendered for this file, with the imported modules listed together and the signature of `parse` shortened. The right panel shows the grounding block appended to the creation observation, with all units except the last abbreviated.

File-side query

```
Created file: src/math_verify/parser.py
Imports:
__future__ re dataclasses typing sympy
  sympy.matrices.matrices .errors .utils
Declarations:
@dataclass(frozen=True) class
  LatexExtractionConfig
@dataclass(frozen=True) class
  ExprExtractionConfig
@dataclass(frozen=True) class
  StringExtractionConfig
parse(pred: str, extraction_config: ...,
  fallback_mode: ..., extraction_mode: ...,
  parsing_timeout: int=5)
```

Grounding block

```
<INTENT_VIEW_EVIDENCE>
Created file: src/math_verify/parser.py
Candidate task intents:
- math_verify.parser.parse() :: ...
- math_verify/__init__.py :: Provide a unified API: from
  math_verify.parser import parse,
  LatexExtractionConfig, ...
- math_verify.parser.parse(parsing_timeout) :: ...
- math_verify.parser.LatexExtractionConfig :: Provide
  fields: try_extract_without_anchor=True;
  boxed_match_priority (0 highest, 50 default, -1 disables
  boxed extraction); extract_last_expression to prefer the
  final expression; boxed priority must control ordering
  when multiple candidates exist and support multiple
  \boxed{...} instances.
</INTENT_VIEW_EVIDENCE>
```

Every selected unit is anchored on an artifact that the query declares, namely `parse` and `LatexExtractionConfig`, or on the package entry that exports both. The pairing therefore rests on artifact names shared by the specification and the code rather than on the wording of either side. The last unit illustrates what each side contributes. On the specification side, `LatexExtractionConfig` is introduced among the configuration classes with a single field, whereas the statement that the value -1 of `boxed_match_priority` disables boxed extraction appears nearly 600 lines later in a section on boxed environment priority. The anchor gathers both statements under the class, so they are retrieved together once a file declaring the class is created. On the code side, the created file treats the value -1 as an ordinary priority in its private helper `_extract_latex_segments`, so boxed content is still extracted. The helper is absent from the query, yet the unit reaches the file through the class whose field the helper reads. Because the query represents the responsibility of the file rather than its implementation, the grounding block delivers the very requirement that the implementation misses.

E Experimental Detail

E.1 Hardware and Software Environment

All experiments were conducted on a Linux workstation running Ubuntu 24.04 LTS, equipped with an Intel Core i5-13400F CPU, 32 GB of memory, and 1 TB of storage. The software environment consists of Python 3.11, PyTorch 2.13.0, and Docker 29.6, and both agent execution and benchmark

evaluation were performed in isolated Docker containers. LLM inference was carried out via the Azure API, spanning an operational period of approximately four weeks.

E.2 Baseline Setup

Shared protocol. All baselines follow a common protocol so that the comparison isolates the agent design. Each task is generated by a single non-interactive run in a fresh Docker container built from `python:3.13-slim`, which matches the runtime of the OpenHands agent server, with 2 CPU cores and 4 GB of memory. The container mounts only the task workspace and a per-task agent home, so user-level configuration, memories, and project instruction files never enter the agent context. Because the container serves as the sandbox, every agent runs with all tool permissions granted and without human approval. On RepoGenesis, the workspace contains the task files with all tests removed, and the prompt embeds the README and the reference `requirements.txt` in the same template for all systems. On NL2Repo-Bench, the workspace contains only `start.md`, and test commands and test file lists are never copied. All systems reach the same model deployments, `gpt-5-20250807` and DeepSeek-V4-Pro, through a local forwarding service with reasoning effort set to high. For DeepSeek-V4-Pro, thinking mode is enabled and the maximum output length is 16,384 tokens for every system. The delivered workspace is evaluated as is without post-processing. Unless stated otherwise, each agent keeps its default system prompt, tool set, and context management. Table 6 lists the versions.

OpenHands. We use OpenHands SDK v1.21.0 with the default CLI agent preset, which is also the base agent of SIRA. It keeps the default tool set and the LLM summarizing condenser, which retains the first 4 events and condenses the history once it exceeds 80 events. Each run is limited to 500 iterations.

mini-SWE-agent. We use mini-SWE-agent v2.4.6 with the upstream `DefaultAgent` and `mini.yaml` configuration. The agent acts only through a single bash tool over a linear message history. We set the step limit to 500 and the timeout of each command to 600 seconds.

OpenCode. We run OpenCode v1.18.10 in non-interactive mode. The backbone is registered as a custom provider whose context and output limits follow the model, namely 272K and 65,536 tokens for GPT-5 and 1M and 16,384 tokens for DeepSeek-V4-Pro. Automatic updates and online model catalog fetching are disabled.

Codex. We run Codex CLI v0.144.5 through `codex exec`. We provide the model catalog entry of GPT-5 so that Codex uses the correct context window and its native patch editing tool, which it otherwise withholds for unrecognized models. Memories, multi-agent delegation, and tool search are disabled. Codex is evaluated only with GPT-5 because it communicates only through the Responses API and edits files with a grammar-constrained patch tool, whereas DeepSeek-V4-Pro is served only through Chat Completions without constrained decoding, so adapting Codex to DeepSeek-V4-Pro would introduce implementation effects that cannot be attributed to the agent itself.

Claude Code. We run Claude Code v2.1.223 in print mode with permission checks skipped. Claude Code supports only the Anthropic Messages API, so a protocol bridge forwards its requests to DeepSeek-V4-Pro and preserves thinking content across turns. All model roles, including the lightweight auxiliary model, are mapped to DeepSeek-V4-Pro. We keep the default 200K context window that Claude Code assigns to unrecognized models, together with its default compaction. We do not evaluate GPT-5 with Claude Code because doing so would require additional adaptation of its Responses API semantics to Claude Code's Anthropic-specific interaction protocol, introducing adapter-specific effects.

GitHub Copilot. We run GitHub Copilot CLI v1.0.77 in prompt mode with all tools allowed and a user-supplied model provider. Offline mode disables web tools and the GitHub MCP server, which reach beyond the workspace and are unavailable to the other baselines. With GPT-5, Copilot uses a built-in model profile whose prompt budget matches the 272K context window of GPT-5, while all

requests are served by GPT-5. With DeepSeek-V4-Pro, Copilot runs in autopilot mode without user queries or custom instructions, with the prompt budget set to 983,616 tokens.

CodeS. We use the prompt-engineering variant of CodeS from its official implementation. It builds a repository through three fixed LLM stages, RepoSketcher, FileSketcher, and SketchFiller, without tool use or execution feedback. We keep the upstream prompts and response parsing and apply only compatibility fixes. SketchFiller receives the complete task specification instead of the upstream summary, which keeps only the text before the second heading and covers less than 2% of our specifications. We also enable generation of asynchronous functions and non-Python files, which most RepoGenesis services require.

Table 6: Versions and backbones of the baseline systems.

System	Version	Backbones
OpenHands	SDK 1.21.0	GPT-5, DeepSeek-V4-Pro
mini-SWE-agent	2.4.6	GPT-5, DeepSeek-V4-Pro
OpenCode	1.18.10	GPT-5, DeepSeek-V4-Pro
Codex CLI	0.144.5	GPT-5
Claude Code	2.1.223	DeepSeek-V4-Pro
GitHub Copilot CLI	1.0.77	GPT-5, DeepSeek-V4-Pro
CodeS	commit `0b624ab`	GPT-5, DeepSeek-V4-Pro

E.3 PROMPT

SIRA uses prompts for key intent extraction, cross-task guidance induction, and post hoc grounding. Task analysis identifies implementation problems, and guidance induction converts reusable findings into domain-specific extraction guidance. Grounding uses a fixed system policy and file-creation observations. In these prompts, braced placeholders denote dynamic inputs. For example, the {`cross_task_experience`} placeholder is replaced by the cross-task extraction experience block when guidance is available and omitted otherwise. The {`domain_line`} placeholder contains the complete domain description line. Other placeholders are named after their inputs.

Prompt E.1: Key Intent Extraction

Construct a compact, deliberately non-exhaustive intent index for a repository-generation agent. This index is not a complete representation of the task. It should contain only the task intents whose later reminder is most likely to prevent an important requirement from being omitted, only partly realized, or implemented incorrectly. The agent has already received the complete task document before beginning work. Read the complete source document enclosed in the <`task_document`> element after these instructions, and extract the intents that satisfy the selection and organization rules defined here.

An *intent unit* is a concise, source-grounded implementation intention organized around one coherent behavior, artifact, or cross-component invariant. It is not an exhaustive summary of a section or the task document. Each unit is indexed as an independent retrieval passage. After the agent first creates a file, the system uses signals from the file's path and content to retrieve matching units and injects their original text into the agent's context, prompting it to review the file and immediately correct any clear conflict.

{`cross_task_experience`}

Selection rules:

- Select important specifications that are easy to forget, misunderstand, only partly realize, or implement incorrectly. When cross-task experience is provided, use it to recognize the kinds of intent that prior generated repositories repeatedly failed to realize well.
- Prefer important intent hidden in negative paths, examples, prose, or distant sections, and intent that must remain consistent across files or components.
- Omit ordinary main-path behavior already made unmistakable by the task, facts a competent implementation is very likely to satisfy by default, exhaustive field-by-field or endpoint-by-endpoint restatements, and documentation that merely repeats implementation contracts represented elsewhere.

- Omit aspirational or non-checkable quality prose, instructions merely telling the agent to analyze or test the task, and generic repository-completeness reminders.

Content organization rules:

- Every intent must be supported by the current task document. Do not add a file, symbol, behavior, value, default, risk, or relationship merely because it is common in similar repositories.
- State each task fact once in its best-fitting intent.
- Keep together only facts needed to express one coherent key intent. Split a long owner contract when it contains independently retrievable intentions, but do not fragment one schema, enum, supported-value set, or tightly coupled behavior into isolated fields and literals.
- Begin every unit with the strongest explicit, source-supported API, component, artifact, command, workflow, or invariant likely to occur in an implementing file's path, imports, declarations, or code. This is the retrieval anchor. If the task names no such owner, use a concise descriptive anchor grounded in its wording.
- Make every unit self-contained and preserve the exact context needed to act on it. Normally keep a unit within 120 words; exceed this limit only when shortening an indivisible exact set would make it ambiguous.

Before answering, silently examine each candidate and retain it only if it is source-backed, important to task correctness, plausibly easy to lose during a long repository-generation trajectory, and not duplicated by another unit. It is correct to omit explicit requirements that are obvious, low-risk, or already strongly reinforced by the main task structure. Do not output this analysis.

Output rules:

- Output exactly one intent unit per physical line using: `<retrieval anchor> :: <key intent>`
- Do not output JSON, Markdown fences, bullets, numbering, headings, commentary, or blank lines inside a unit.
- Keep each line self-contained and do not refer to "above" or "below".
- Output exactly `<NO_INTENT_UNITS>` only when the document contains no key software implementation intents under these rules.

`<task_document>`

`{task_document}`

`</task_document>`

Prompt E.2: Cross-Task Extraction Experience Block

Cross-task extraction experience:

- The guidance below summarizes kinds of intent that prior generated repositories in this domain repeatedly failed to implement correctly. Use it only to prioritize and organize requirements explicitly present in the current task document.
- Never emit an intent, value, behavior, file, or relationship unless the current task document itself supports it, and never copy the guidance itself as an intent unit.

`{experience}`

Prompt E.3: Task Analysis, First Stage of Self-Improvement

Compare the complete task prompt with the generated repository implementation view. The view contains the project file tree and the full contents of its source, template, script, and project configuration files. A path shown only in the tree exists, but its contents are intentionally not part of this review; do not infer what that file contains or report a problem that depends on inspecting it.

Identify important implementation intentions that the repository did not implement correctly. A problem may be an omission, a partial implementation, incorrect behavior, an interface mismatch, or an inconsistency across files.

Judge only requirements that can be checked from the file tree or the expanded implementation files. Do not guess why the generator made an error. Report a problem only when the task requirement and the repository evidence are both clear.

Set reusable to true only when the problem reflects a kind of task intent whose recurrence could usefully guide key-intent extraction for other repository-generation tasks. A repository-specific name or value may appear in the evidence, but a purely local accident is not reusable.

Return only a JSON array. Each element must have exactly four fields:

- "problem": what important task intention was not implemented correctly;
- "task_evidence": where and how the task prompt requires it;
- "repository_evidence": where and how the generated repository conflicts with or omits it;
- "reusable": a JSON boolean.

Use those exact field names; do not rename them to requirement, conflict, evidence, or location. Return [] when no sufficiently evidenced problem exists. Do not output Markdown fences, commentary, diagnoses of root cause, implementation advice, or any field other than the four fields above.

<task_prompt>

{task_prompt}

</task_prompt>

<repository_implementation_view>

{repository_implementation_view}

</repository_implementation_view>

Prompt E.4: Guidance Induction, Second Stage of Self-Improvement

{domain_line}

The records below are evidence-grounded problems found by comparing completed task prompts with their generated repositories. Derive reusable guidance for selecting key intents from unseen task prompts in this domain. The guidance will be consumed by an intent extractor, not by a repository implementation agent.

Produce concise bullet points, one for each distinct transferable pattern supported by the records. Each bullet must identify a kind of intent explicitly stated in a task document that deserves higher priority during extraction, and state which task-specified details or relationships should be preserved together when that kind of intent appears. Phrase each bullet as extraction guidance in the form "Prioritize intents about ...; preserve ...". Describe the information that should be extracted, not behavior that every future repository must implement. Do not prescribe a repair or concrete implementation mechanism.

A transferable pattern does not have to appear in more than one task: this stage pools reusable records instead of applying a frequency threshold. Include a pattern only when at least one provided record supports it and its abstraction can apply beyond that record. Consolidate semantically overlapping records, but do not merge unrelated patterns merely to shorten the result, and do not target a fixed number of points.

Generalize beyond individual records: do not copy a task name, repository path, API or symbol name, framework name, or concrete value. Do not infer why an error occurred, mention the analysis process, or add generic advice that would be true without the records. The unseen task prompt remains the only authority for its actual requirements.

Output exactly <NO_EXPERIENCE> if the records do not support reusable extraction guidance. Otherwise output only the bullet points, with no heading or commentary.

<analysis_records>

{analysis_records}

</analysis_records>

The following policy is appended once to the agent system prompt.

Prompt E.5: Post Hoc Grounding

<INTENT_VIEW_POST_CREATE_REVIEW_POLICY>

After a successful file creation, its tool observation may contain an <INTENT_VIEW_EVIDENCE> block appended by the environment. The content inside that block is used to correct any implementation errors that **may** exist in the corresponding file.

Whenever such evidence appears:

1. Determine whether each candidate task intent applies to the just-created file.
2. If an intent applies, compare the implementation with the complete intent. Immediately correct clear mismatches in that file or directly related files.
3. If an intent is irrelevant or already satisfied, make no change for it.
4. Do not produce a separate review report. Continue the original repository task. Only after all repository code has been generated, verify that it can be installed, deployed, or executed successfully.

</INTENT_VIEW_POST_CREATE_REVIEW_POLICY>

At a grounding checkpoint, the following block is appended to the observation of the file creation tool call.

Prompt E.6: Grounding Observation

<INTENT_VIEW_EVIDENCE>

Created file: {relative_path}

Candidate task intents:

- {intent_unit}

</INTENT_VIEW_EVIDENCE>

E.4 Dataset Preparation

Tasks of each benchmark are divided into two disjoint sets, namely analysis tasks and evaluation tasks. *Evaluation tasks* produce all reported results of SIRA and the baselines. *Analysis tasks* are never used to report results. They are used only for the motivation study in Appendix A and as the completed tasks $\mathcal{D}_d$ from which the extraction guidance g_d is induced. No model parameters are updated on analysis tasks, and guidance induction uses only their task specifications and delivered repositories without accessing any test suite. Table 7 summarizes the resulting partition.

NL2Repo-Bench. NL2Repo-Bench contains 104 Python tasks and reports nine application categories with the number of tasks in each, but does not release the category of individual tasks. We therefore assign each task to one category according to the primary purpose stated in its task document, subject to the official category sizes. Five categories are retained as domains, namely System Tools, Data Analysis & Processing, Testing, Web Development, and Machine Learning, which together contain 72 tasks. Batch File Processing is excluded because its five tasks are too few to be divided within the domain. Database Interaction and Networking Tools are excluded because several of their tasks depend on external databases or network services, which makes evaluation outcomes sensitive to the execution environment. Utility Libraries is excluded because its boundary with System Tools and Data Analysis & Processing is ambiguous. Within each domain, roughly 30% of the tasks are held out as evaluation tasks and the rest become analysis tasks, with the official easy, medium, and hard levels kept in similar proportions across the two sets. The partition yields 22 evaluation tasks with 7, 9, and 6 tasks at the three levels and 50 analysis tasks with 15, 21, and 14.

RepoGenesis. RepoGenesis provides an official split into RepoGenesis (Verified) and RepoGenesis (Train). The Verified split contains 30 repositories whose test suites are adjudicated by human experts and serves as the evaluation benchmark. The Train split contains 76 repositories released for model training, whose test cases are validated only by LLM consensus. We follow this split, using the 22 Python repositories of the Verified split as evaluation tasks and the 38 Python repositories

of the Train split as analysis tasks. Java repositories are excluded from both sets so that the two benchmarks are compared under a single language. All repositories are web microservices and are therefore treated as a single domain.

Table 7: Partition of analysis and evaluation tasks. Ana. and Eval. denote the numbers of analysis and evaluation tasks. Evaluation tasks of NL2Repo-Bench are listed by name.

Domain	Ana.	Eval.	Evaluation tasks
NL2Repo-Bench			
System Tools	17	7	autojump, decouple, gitingest, pyautogui, schedule-master, tqdm, typing_extensions
Data Analysis & Processing	12	6	fuzzywuzzy, jsonlines, markdownify, pdfplumber-stable, tablib, xlrd
Testing	9	4	math-verify, pytestify, python-pytest-cases, uiautomator
Web Development	7	3	bleach, flask-restful, requests-html
Machine Learning	5	2	cherry, deslib
RepoGenesis			
Microservice	38	22	Python repositories of RepoGenesis (Verified)

E.5 Backbone Selection

We use GPT-5 and DeepSeek-V4-Pro, which cover one proprietary and one open-weight backbone and are both widely reported on these benchmarks. We exclude more recent models such as GPT-5.4 because they frequently satisfy NL2Repo-Bench tasks by retrieving the original repository rather than implementing the specification, so their scores no longer reflect generation capability. We further conduct trajectory inspection to confirm that neither backbone used in our experiments took this shortcut.

F Ablation Implementation Details

Every variant in Table 2 produces a repository within a single agent run and, unless otherwise noted, shares the configuration of SIRA, namely the GPT-5 backbone with high reasoning effort and the OpenHands scaffold with its default tool set and condenser. SIRA and the non-deferred variants use a limit of 500 iterations per task, whereas the two Deferred Review variants use a limit of 1,000 iterations. Variants under *What to ground* modify only the grounding content and keep all other components identical to SIRA. Variants under *When to ground* cannot relocate the grounding point in isolation and require the adaptations described below. Relative cost divides the average per-task API cost of a variant by that of OpenHands on each benchmark and averages the two ratios. For SIRA the ratios are $0.95\times$ on RepoGenesis and $1.31\times$ on NL2Repo-Bench.

w/o Task-Specific Content. The variant preserves the checkpoint trigger, the file filters, and the observation channel of SIRA, and substitutes a fixed reminder for the grounding block. The reminder states the path of the newly created file and instructs the agent to review it against the task specification, and it carries no content derived from the task. The system instruction retains the same clause that defers installation, deployment, and execution until all repository code has been generated, and differs from that of SIRA only in asking the agent to check the newly created file against the task specification in general, without referring to any specific intent. Because no candidate is subject to relevance filtering, the reminder is injected at every checkpoint, and the number of interventions is therefore no smaller than that of SIRA.

w/o Intent Index. The variant retains the checkpoint and the entire retrieval pipeline and replaces only the retrieval corpus. The task specification is segmented into passages and indexed directly, so each newly created file is matched against the original specification text rather than against extracted intent units. Query construction, hybrid ranking, rank fusion, and the selection budget follow SIRA, and the grounding block delivers the retrieved passages in their original wording. The contrast with SIRA therefore isolates the effect of condensing the specification into a compact intent index.

Table 8: Domain descriptions supplied to guidance induction and the number of directives in the resulting guidance of each domain.

Domain	Description	Directives
System Tools	Reusable command-line tools, configuration loaders, and runtime utilities that other Python programs install and call.	20
Data Analysis & Processing	Libraries that read, convert, and write tabular, textual, or document data across several interchange formats.	22
Testing	Libraries and tools that inspect, transform, or verify other code, and that are themselves driven by a test runner.	15
Web Development	Libraries that build, parse, sanitise, or fetch web content and expose it through a Python API.	17
Machine Learning	Libraries that prepare data, fit or combine models, and report predictions or scores.	22
Microservice	Single backend services that expose an HTTP API over persisted state and must start from one documented command.	15

w/o Experience. The variant removes the cross-task experience block from the extraction prompt, so key intents are extracted from the task specification alone. Indexing, retrieval, budgeted selection, the grounding block, and the system instruction are left unchanged, which attributes the observed difference to the quality of the extracted intents rather than to the grounding mechanism.

Deferred Review. The two deferred variants move the grounding point from file creation to the end of repository generation without leaving the original trajectory. Grounding is triggered when the agent first attempts to terminate the task, at which moment the delivered repository is complete. The grounding block appended at that point supplies the review material together with an instruction to examine the repository against it and to correct clear mismatches, and the agent continues from the augmented history using its existing tools. Deferred specification review supplies the complete task specification, whereas deferred intent review supplies the entire intent index extracted for the task and serves as the closest counterpart of SIRA at the end of generation.

Relocating the grounding point entails several adaptations. Per-file retrieval has no natural counterpart once all files exist, so the index is delivered as a single block rather than as budgeted per-file selections, and the review instruction targets the complete repository rather than one newly created file. The iteration limit is raised to 1,000 so that the review stage is not truncated by the budget consumed during generation. These adaptations provide the deferred variants with more review material and more iterations than SIRA, and are intended to favor rather than handicap the deferred setting.

G Cross-Task Experience

SIRA induces domain-specific extraction guidance g_d for each of the six domains through the two-stage process in Section 2.1. Guidance induction receives a hand-written description of the domain together with the findings pooled from its analysis tasks, and consolidates these findings into a list of directives. Table 8 lists the description and the number of directives for each domain. During key intent extraction, the guidance of the task domain is inserted into the extraction prompt as cross-task experience. Each directive names a kind of intent that deserves higher priority during extraction and the task-specified details that should be preserved together with it. The guidance of each domain is reproduced in its generated order.

Scope of Domain Guidance. The guidance of a domain is applied only to tasks of that domain. Its directives are induced from intents missed in completed tasks of the domain, and many of them name kinds of intent that arise mainly in that domain, such as the mapping from invalid inputs to HTTP status codes in Microservice, per-module command-line entry points in System Tools, and exact layer sizes in Machine Learning. Specifications from other domains seldom contain such intents, so guidance from another domain would give the extractor little to prioritize and would not point to the kinds of intent that the current task tends to lose. A few directives, such as complete dependency declarations and exact public exports, recur across domains and reflect failures common to coding agents rather than to a single domain. The six domains follow the categories of the two benchmarks and are not intended as a general taxonomy of repositories. For a task outside these domains, SIRA

can extract intents without guidance, which still improves Pass@1 from 67.43 to 77.09 and Ave. Score from 31.74 to 37.28 over OpenHands in Table 2.

Extraction Guidance for System Tools

1. Prioritize intents about providing test scripts and example code; preserve the list of functional areas to cover and the requirement that tests automatically verify correctness.
2. Prioritize intents about designing independent command-line interfaces for each functional module; preserve that each module should have its own CLI entry point with clear input/output.
3. Prioritize intents that define explicit function signatures, method interfaces, or data schemas with precise parameters/fields; preserve every specified parameter/field and its type exactly, without ignoring or adding extras.
4. Prioritize intents about installation-time verification of modules; preserve that the install script must check whether all functional modules work correctly.
5. Prioritize intents about completeness of dependency declarations in packaging/install scripts; preserve the requirement to declare core libraries explicitly.
6. Prioritize intents that list API functions with descriptions; preserve that each listed function must have a full implementation consistent with its description, not a stub.
7. Prioritize intents about parsing real external data sources; preserve that the implementation must actually parse the specified files rather than relying on hardcoded data.
8. Prioritize intents about implementing a described algorithm or matching system; preserve that all described components (classes, functions) must be integrated and used in the final logic.
9. Prioritize intents about content processing modes such as intelligent truncation; preserve that the mode must be implemented as described (e.g., partially reading and truncating large files rather than skipping them).
10. Prioritize intents about supporting specific modes of operation; preserve that all enumerated options (e.g., sub-path cloning, branch specification) must be honored.
11. Prioritize intents about sending real notifications via messaging APIs; preserve that the implementation must use actual platform APIs, not print statements.
12. Prioritize intents about integration with external libraries; preserve that integration must use correct base classes, registration mechanisms, and create actual library objects/widgets, not fallback to console.
13. Prioritize intents about CLI parameters supporting environment variable overrides; preserve that every parameter must be overridable via environment variables.
14. Prioritize intents about context managers that alter or redirect state; preserve the described state changes and that original state is fully restored on exit.
15. Prioritize intents about constructor side effects; preserve any listed side effects such as adding to global registries or starting threads.
16. Prioritize intents about module exports as derived from import examples; preserve the exact set of public names and their module locations.
17. Prioritize intents about backward compatibility with specific Python versions; preserve the supported version range and the need for conditional imports or version-specific behavior.
18. Prioritize intents about exact file locations for specific functions; preserve the required module path where the function must be defined.
19. Prioritize intents about catching specific custom exceptions; preserve the list of exception types that must be handled.
20. Prioritize intents about implementing language-level features with correct semantics; preserve behavioral details such as exception propagation for with statements.

Extraction Guidance for Data Analysis & Processing

1. Prioritize intents about build/packaging configuration files (e.g., `setup.py`, `pyproject.toml`) including a test or verification command; preserve the requirement that the file itself must provide a runnable means to verify all functional modules.
2. Prioritize intents about providing example code and test cases that demonstrate specific library functions; preserve the connection between the functions named for demonstration and the requirement for corresponding example/test files or directories.
3. Prioritize intents that include full algorithm definitions, conversion tables, or regex patterns as part of the specification; preserve the entire definition as a unit that must be faithfully implemented, not simplified or replaced with a pass-through.

4. Prioritize intents about behaviour that must differ based on the processing context or a special parameter value (e.g., `'auto'`, inline vs. block context); preserve the described conditional logic that distinguishes the different contexts.
5. Prioritize intents about exact return value structures (e.g., tuple length) documented in the API; preserve that exact structure as the authoritative contract, even if a variant would be convenient.
6. Prioritize intents about function signatures that include specific decorators or annotations; preserve the decorator as part of the required interface, not as an optional detail.
7. Prioritize intents about overloaded type signatures for functions; preserve the requirement that every parameter appear in all overload variants and stub files, not only in the implementation.
8. Prioritize intents that specify multiple alternative conditions triggering a behavior (e.g., short paragraph OR low stopword density); preserve the full set of conditions so that all are handled, not only the first one listed.
9. Prioritize intents about top-level module exports that support `from package import name` usage; preserve the exact public name that must be importable at the package level.
10. Prioritize intents about extracting data from documents or reading specific file formats; preserve the requirement that the implementation must parse the actual document structures or binary format and produce the specified data, not return placeholders, empty collections, or fallback text heuristics.
11. Prioritize intents about exact output formatting rules for generated strings (CLI tables, transliterations, etc.); preserve the provided expected output string as the precise specification to match.
12. Prioritize intents about function parameters that are described in the specification; preserve the requirement that every such parameter must influence the computation or output, not be silently accepted and ignored.
13. Prioritize intents about character set coverage (e.g., handling traditional Chinese characters); preserve the explicit mention of character classes or Unicode ranges that must be supported.
14. Prioritize intents about a specific function whose name and call signature appear in usage examples; preserve that the function must exist as a documented public API with the exact demonstrated signature.
15. Prioritize intents about handling linguistic cases like heteronyms where multiple valid forms exist; preserve the requirement that equivalence or comparison operations account for the multiplicity of valid forms, not just exact character matching.
16. Prioritize intents about data container indexing methods that support access by column name (e.g., `row["A"]`); preserve the string-indexing capability as part of the data type's interface.
17. Prioritize intents about named feature calculators listed in a configuration constant; preserve that every listed calculator must be implemented and produce a result, not silently skipped during extraction.
18. Prioritize intents about machine learning pipeline integration (e.g., sklearn transformers); preserve the transformer lifecycle contract where `transform` extracts features from stored source data, not from the input `X` directly.
19. Prioritize intents about supporting both supervised and unsupervised learning scenarios; preserve the ability to operate without a target variable when the task mentions unsupervised or unlabeled data.
20. Prioritize intents about modular mapping architectures (e.g., per-block Unicode maps); preserve the requirement that transliteration uses dynamically loaded per-block modules, not a monolithic hard-coded lookup.
21. Prioritize intents about warnings or exceptions for specific input categories (unmappable characters, surrogates, private area, etc.); preserve the link between those categories and the requirement to emit diagnostic warnings or exceptions, not silently handle them.
22. Prioritize intents about specific HTML/XML attributes for document conversion (e.g., `colspan`); preserve the requirement that the converter must inspect and faithfully apply the attribute's semantics to the output.

Extraction Guidance for Testing

1. Prioritize intents about exhaustive sets of items that must all be present; preserve the complete list and the requirement that each item is implemented or supported.
2. Prioritize intents about integration with specific named external libraries; preserve the library names and the requirement to call them directly rather than reimplementing their functionality.
3. Prioritize intents about real communication with an external system; preserve the requirement for actual interaction (not stubs) and the relevant protocol.
4. Prioritize intents about custom type preservation through arithmetic or serialization operations; preserve the specific operations and the requirement that the custom type is retained.

5. Prioritize intents about bidirectional or symmetric operations on a class; preserve the methods and parameters that enable reverse application and verification.
6. Prioritize intents about secure processing mechanisms such as safe deserialization; preserve the restricting mechanism and the class that implements it.
7. Prioritize intents about precise behavioral contracts in API methods or features; preserve explicit return types/structures, exception message content, mode-specific algorithm steps, required environment variables or file artifacts, immutability constraints, and state-checking requirements.
8. Prioritize intents about configuration file discovery that traverses parent directories; preserve the functions that must perform upward traversal.
9. Prioritize intents about mandatory deliverable files such as examples, notebooks, and test scripts; preserve the file names and the requirement that they contain described scenario content.
10. Prioritize intents about prescribed project module structure; preserve the mapping of components to specific module files.
11. Prioritize intents about callback or handler chains triggered by specific error conditions; preserve the triggering condition and the sequential execution rule.
12. Prioritize intents about mark or tag propagation from test case definitions to execution; preserve that marks set on case functions must be applied to generated parametrization.
13. Prioritize intents about self-contained verification scripts that must run without prior build or install; preserve the requirement to correctly import from the source tree.
14. Prioritize intents about inter-process data exchange in distributed testing; preserve the variable names and the master-worker data flow.
15. Prioritize intents about signal handling for data cleanup on abnormal termination; preserve the specific signals and the required state-saving action.

Extraction Guidance for Web Development

1. Prioritize intents about project dependency completeness; preserve the full list of required libraries, including test tools and any library directly imported in the source code.
2. Prioritize intents about package verification; preserve that setup.py must provide a command that verifies all functional modules work properly.
3. Prioritize intents about function and constructor parameter defaults; preserve the exact default value expression (constant reference or literal) as specified in the API documentation.
4. Prioritize intents about class interface specifications; preserve every method name and its described purpose for each target class.
5. Prioritize intents about parameters whose descriptions indicate behavioral effects; preserve the full behavioral description, not just the parameter's existence or storage.
6. Prioritize intents about module-level constants; preserve the exact names and literal values as given.
7. Prioritize intents about framework signal integration; preserve the specific signal names that must be connected in error handling or request pipelines.
8. Prioritize intents about required features or subsystems; preserve the complete feature description and any enumerated sub-features (e.g., security protections, async rendering, extension mechanisms, template inheritance, internationalization, loop context properties).
9. Prioritize intents about processing pipeline outputs; preserve that parsers must produce an Abstract Syntax Tree and code generators must consume that tree.
10. Prioritize intents about built-in function sets (filters, tests); preserve the complete list of required functions explicitly listed in the specification.
11. Prioritize intents about adapter or translator class responsibilities; preserve that the designated class must implement the full conversion logic, not stubs that delegate the actual work elsewhere.
12. Prioritize intents about text extraction algorithms; preserve the detailed rules for block separators, whitespace handling, and parameter effects as described.
13. Prioritize intents about method return types and content structures; preserve whether a method returns a generator or a list, and whether text nodes should be plain strings or element wrappers.
14. Prioritize intents about CLI argument parsing; preserve the argument syntax (e.g., accepting multiple values after a single flag) as shown in documented usage examples.
15. Prioritize intents about class inheritance hierarchies; preserve the base classes and inheritance relationships explicitly stated for session classes or similar components.
16. Prioritize intents about Python version compatibility; preserve the claimed minimum supported version and flag any language feature that is incompatible with that version.

17. Prioritize intents about form field specifications; preserve the exact field types (e.g., PageDownField) and the third-party dependencies they imply.

Extraction Guidance for Machine Learning

1. Prioritize intents about exact dimensions and layer sizes of neural network architectures; preserve the numeric values (e.g., input size, hidden-layer sizes) as stated in the specification.
2. Prioritize intents about enumerated sets of actions, commands, or operations that a system must support; preserve the complete list of named actions and any required compound or combined variants.
3. Prioritize intents about the public API surface and top-level import convenience; preserve which classes, functions, or symbols must be accessible via a wildcard import or a flat namespace.
4. Prioritize intents about mandatory dependencies and their declaration; preserve the exact set of library names that the task lists as core or required dependencies.
5. Prioritize intents about built-in pre-trained models or bundled assets that users should load directly; preserve the requirement that actual model files or usable artifacts must be shipped, not placeholders or only URLs.
6. Prioritize intents about supported input data formats and their internal schema; preserve the concrete file types (e.g., .csv, .txt) and the data-to-label association structure that must be handled.
7. Prioritize intents about command-line interface parameters; preserve the full set of arguments the CLI must accept and the semantics of each (e.g., data path, model save path, input text).
8. Prioritize intents about default values for parameters in function and class signatures; preserve the exact default value (numeric, boolean, or enum) as stated in the specification.
9. Prioritize intents about parameters that drive algorithmic behavior; preserve that the parameter must be actively used in the implementation and, when a set of behaviors is specified (e.g., aggregation strategies), preserve that enumerated set.
10. Prioritize intents about standalone utility functions that a module must provide; preserve the required function categories (e.g., graph traversal, training visualization) as named requirements.
11. Prioritize intents about non-packaging responsibilities of build or setup files; preserve verification or testing commands that the file must support beyond simple metadata.
12. Prioritize intents about specific files or modules that must contain designated functions with prescribed roles; preserve the file path, function names, and their core responsibility (e.g., serialization save/load).
13. Prioritize intents about functions that must create or return instances of multiple concrete types; preserve that the function must support all listed variants (e.g., DummyVecEnv and SubprocVecEnv) and not silently hard-code one.
14. Prioritize intents about functions that set global state across several libraries/systems; preserve the full list of targets that must be seeded or configured (e.g., random, numpy, torch, and the environment).
15. Prioritize intents about algorithm-module completeness; preserve that each algorithm must include non-stub implementations of training logic, network architecture, and hyperparameter configuration.
16. Prioritize intents about callback or event systems; preserve the required set of callback classes and the monitoring, saving, early-stopping, and evaluation capabilities they must provide.
17. Prioritize intents about specific methods that must exist on a class; preserve the method name and the described transformation or operation it must perform (including its output types).
18. Prioritize intents about output fields or keys produced by data-processing functions; preserve the exact set of fields (e.g., masks) and that they respect the relevant masking/truncation options.
19. Prioritize intents about feature-level modules or classes described as part of a system; preserve that each named module/class (e.g., a tool-invocation environment) must be implemented, not omitted.
20. Prioritize intents about aggregation or reduction behavior in composite components; preserve that the component must combine or accumulate results from its sub-components rather than silently using only the first.
21. Prioritize intents about configuration parameters that govern control flow or termination; preserve the semantics of special sentinel values (e.g., -1 for unlimited) and that the default behavior matches the intended use case.
22. Prioritize intents about separate concurrency limits for different phases of a process; preserve that distinct parameters (e.g., max_concurrent_generation and max_concurrent_scoring) must be enforced individually.

Extraction Guidance for Microservice

1. Prioritize intents about access control rules that depend on resource ownership or state; preserve the exact conditions and that they apply to every endpoint that returns or exposes those resources, not only single-resource lookups.
2. Prioritize intents about API response schemas; preserve the complete field set, types, and requiredness (no extra fields, no missing fields, and no null values where a non-nullable type is specified).
3. Prioritize intents about request body requirements; preserve that a body schema documented for an endpoint is mandatory unless the specification explicitly marks it as optional.
4. Prioritize intents about required logging coverage; preserve the exact categories of operations that must be logged (e.g., all operations, security events, access audits), and that this is a global requirement across all relevant endpoints and actions.
5. Prioritize intents about a unified error response structure; preserve that the specified error format must be used for every error response, including unhandled internal server errors and validation errors.
6. Prioritize intents about distinct error handling for different validation categories; preserve the mapping between validation error types (e.g., request parameter error vs. data validation failure) and their corresponding HTTP status codes.
7. Prioritize intents about fine-grained, domain-specific error codes; preserve the exact mapping from each invalid input value or condition to its designated error code string.
8. Prioritize intents about operation parameters that control output behaviour; preserve that any option declared as applicable must be honoured in every relevant processing path (e.g., all source-target format combinations).
9. Prioritize intents about error codes that imply cross-cutting mechanisms; preserve that any listed error code (e.g., 429) requires the corresponding behavior to be present and effective across endpoints.
10. Prioritize intents about the deployment start command; preserve that the start script must use only the framework's own built-in run command, without fallback to external servers.
11. Prioritize intents about background processing that continues after the initial request; preserve that such processing must not depend on request-scoped resources and must be self-contained to complete successfully.
12. Prioritize intents about dependency version constraints; preserve the exact version ranges specified for frameworks or libraries, noting that the implementation must be compatible with those versions and their APIs.
13. Prioritize intents about the binding of input parameters to request parts; preserve that fields described within a multipart/form-data or JSON body schema must be extracted from the body, not from query parameters.
14. Prioritize intents about server-managed fields with temporal semantics (e.g., `updated_at`); preserve that their value must reflect the correct point in time after the corresponding operation, regardless of the underlying database backend.
15. Prioritize intents about validation constraints on path or query parameters that appear in response schemas; preserve that any pattern, format, or constraint declared for such a parameter must be enforced at input to ensure responses conform.

H Effect of Cross-Task Experience on Intent Extraction

We compare the intent indexes extracted for the evaluation tasks with and without cross-task experience under GPT-5. The two settings share the task specification, the extraction prompt, and the backbone, and differ only in whether the domain guidance is inserted into the prompt, which corresponds to the w/o experience variant in Table 2.

Table 9: Size of the intent index extracted without (w/o) and with (w/) cross-task experience under GPT-5. Values are averaged over the 22 evaluation tasks of each benchmark.

Benchmark	Setting	Units	Tokens	Tokens per Unit
NL2Repo-Bench	w/o	17.9	1,220	68.3
	w/	17.4	1,236	71.0
RepoGenesis	w/o	12.0	635	53.1
	w/	12.5	710	57.0

Index Size. As shown in Table 9, cross-task experience changes the size of the intent index only marginally. The number of units per task differs by less than one on both benchmarks, and the number of tokens per task increases by 16 on NL2Repo-Bench and by 75 on RepoGenesis. The index also remains compact relative to the specification. On NL2Repo-Bench, whose specifications contain about 17.4K tokens on average, the index amounts to about 7% of the specification. Inserting the guidance adds about 0.7K input tokens to the single extraction call of each task. The influence of experience therefore lies mainly in which intents are selected rather than in how many are retained.

Table 10: Representative differences between the intent indexes extracted for File_Relay without and with cross-task experience under GPT-5. The first column names the directive of the Microservice guidance that corresponds to each difference. Each unit shows its anchor in bold followed by its intent, reproduced verbatim with omissions marked.

Guidance directive	W/o experience	W/ experience
Unified error response structure	**Error Responses** All non-download endpoints return application/json; errors use `{"error":{"code":string,...}}` with codes mapped to HTTP: INVALID_REQUEST 400, ..., INTERNAL_ERROR 500.	**Error Responses** Use the unified JSON error envelope `{"error":{"code":string,...}}` for every error across all endpoints, including validation failures and unhandled internal errors; set Content-Type application/json.
Binding of input parameters to request parts	**POST /api/v1/files** Accept multipart/form-data with required file; description optional (max 500 chars); tags optional comma-separated string parsed to array; ...	**POST /api/v1/files (multipart/form-data)** Read 'file' (required), 'description' (optional, max 500 chars), and 'tags' (optional) from the multipart body; parse 'tags' as a comma-separated string into an array; do not read from query; if 'file' missing or constraints violated, return 400 INVALID_REQUEST.
Server-managed fields with temporal semantics	*No corresponding unit*	**Server-managed timestamps** Set server-generated timestamps: 'created_at' (user registration output) and 'upload_time' (file upload metadata) using current ISO 8601 UTC; clients must not provide these; ensure values reflect the actual operation time.
Required logging coverage	*No corresponding unit*	**Logging coverage** Capture logs for all operations (auth, uploads, downloads, metadata updates, deletions, listings, searches), including actor identity, resource IDs, and outcomes; apply globally for auditing and debugging.

Case Study. We illustrate this influence with File_Relay from RepoGenesis, a file management service whose two indexes contain 15 and 17 units. The task is chosen because the differences between its two indexes correspond closely to individual directives of the Microservice guidance and to explicit statements in the specification. Table 10 lists four representative differences.

Without experience, the extracted units largely follow the organization of the specification, with one unit for each endpoint and one for the error format. Requirements that span several endpoints are therefore stated only partially or omitted. The specification requires standard error responses for all endpoints and includes INVALID_REQUEST and INTERNAL_ERROR among the error codes. The unit extracted without experience restates the envelope and the code mapping, whereas the unit extracted with experience explicitly extends the envelope to validation failures and unhandled internal errors, for which a web framework otherwise returns its own default responses. Likewise, the upload fields are documented under a multipart input schema, and only the unit extracted with experience states that they must be read from the request body rather than from query parameters. The timestamp fields appear only in output schemas, and the logging requirement appears among the implementation requirements. Both become standalone units only when experience is used.

These additions do not substantially enlarge the index. The index extracted with experience no longer contains the unit for the health check endpoint or the username format rules of registration, which the main task structure already reinforces. Each added unit restates a requirement present in the specification, so the guidance affects which requirements are selected and how completely they are stated without introducing requirements absent from the task. The added units also reach the agent at the files responsible for them. For example, the logging unit is retrieved and grounded when the agent creates `app/logging_config.py`.

Table 11: Results of applying SIRA to OpenCode with GPT-5. **Rel. Cost** denotes the per-task token consumption relative to OpenCode.

Method	RepoGenesis		NL2Repo-Bench	Rel. Cost
	Pass@1 ↑	DSR ↑	Ave. Score ↑	
OpenCode	58.46±0.68	89.39±2.62	33.99±0.59	1.00×
OpenCode + SIRA w/o Experience	68.91±0.76	93.94±2.62	38.87±0.48	1.22×
OpenCode + SIRA	**72.73±0.91**	**95.45±0.00**	**42.16±0.51**	1.19×

I Generalization Beyond OpenHands

We further apply SIRA to OpenCode, an independently developed coding agent with its own tool set, file editing protocol, and context management. SIRA is implemented as an OpenCode plugin built on hooks of the agent, leaving its source code unchanged. Intent extraction, experience guidance, the checkpoint filters, retrieval, and all parameters are identical to those used with OpenHands. We evaluate OpenCode, OpenCode with SIRA, and OpenCode with SIRA without experience using GPT-5 under the same protocol as the main experiments. Table 11 reports the results. SIRA improves OpenCode by 14.27 Pass@1 points on RepoGenesis and by 8.17 Ave. Score points on NL2Repo-Bench at 1.19× the token consumption, and OpenCode with SIRA outperforms every GPT-5 baseline in Table 1 on both primary metrics. Without experience, SIRA still improves Pass@1 by 10.45 points and Ave. Score by 4.88 points, and experience contributes a further 3.82 and 3.29 points. These results suggest that the gains of SIRA are not specific to the OpenHands scaffold. Since SIRA interacts with the agent only through the system prompt and the observations of file-creating actions, it can be attached to other agents that provide comparable hooks.

J Classification of Agent Responses to Grounding

Samples. We analyze all grounding checkpoints in the GPT-5 runs of SIRA, comprising 174 checkpoints on RepoGenesis and 354 on NL2Repo-Bench. For each checkpoint, the file named in the grounding block is referred to as the target file.

Observation window. The grounding policy instructs the agent to verify the repository only after all code has been generated. We therefore observe the response to each checkpoint before the generated project is first run. The window starts at the first action taken after the grounding block becomes visible. Tool calls issued in the same model response as the file creation are excluded, since they are decided before the block is observed. The window ends at the first subsequent action that runs the project, such as starting the service, running the test suite, or executing repository code. If the project is not run after the checkpoint, the window extends to the end of the trajectory.

Classification criteria. Each checkpoint is assigned to one of three categories according to the actions within its window.

- **Revised on the target file.** The content of the target file at the end of the window differs from its content at creation.
- **Revised on a related file.** The target file is unchanged, and the agent modifies another existing file in a way that implements or corrects a specific item stated in the grounding block, such as a field, a status code, a parameter, a behavior, or a dependency version. Newly created files are not considered, as they belong to the ongoing generation. Changes without such a correspondence, such as import adjustments or fixes of unrelated errors, are not considered either.
- **No revision.** Neither condition is satisfied. This category also covers checkpoints where the target file is viewed but not changed.

Modifications after the first run. Modifications made after the project is first run are not counted as revisions, since they may be driven by runtime errors or test outcomes. The corresponding checkpoints remain in the analysis and are classified as no revision, so the categories in Table 3 sum to

100%. Actually, the target file is modified only after the first run in 7 checkpoints on RepoGenesis and 23 checkpoints on NL2Repo-Bench.

K Evidence for Intent Alignment During Generation

One grounding checkpoint in the GPT-5 File_Relay trajectory of the experience-guided run is traced from file creation to the final repository. Positions in the trajectory are reported as Δ, the number of agent actions taken after the grounded file creation, counting actions that fail. Source wording is preserved and omissions are marked.

Original task evidence. The specification documents the two endpoints under separate headings with distinct output schemas.

Response schemas of the two endpoints in the task specification

```
GET /api/v1/files         → files, page, page_size, total, total_pages
GET /api/v1/files/search  → files, query, page, page_size, total
```

In the search example the field `query` carries the value `presentation` for the request `q=presentation`, so it echoes the submitted term rather than a stored value. The search endpoint also lists status 400 for an invalid query, and the common error table of the specification binds status 400 to the code `INVALID_REQUEST`.

Implementation at file creation. The agent creates `app/main.py` in a single write of all endpoints. The search handler declares `response_model=FileListResponse` and returns that model with `total_pages` and without `query`. It constrains `q` by `min_length=2` but registers no handler for request validation errors, so a short `q` yields the framework default of 422 instead of the specified 400. The listing handler returns the same model correctly.

Delivered grounding evidence. Four units are selected for this file. The unit stating the search response intent is reproduced in full, while the remaining three, on startup, update authorization and bearer authentication, are abbreviated.

Grounding block appended to the creation observation of `app/main.py`

```
<INTENT_VIEW_EVIDENCE>
Created file: app/main.py
Candidate task intents:
- start.sh :: ...
- GET /api/v1/files/search validation :: Require query param q with length >= 2; enforce page/page_size defaults and ranges as in list;
  invalid q yields 400 INVALID_REQUEST; response must include files, query (echoing q), page, page_size, and total.
- PATCH /api/v1/files/{file_id} authorization :: ...
- BearerAuth across /api/v1/files* :: ...
</INTENT_VIEW_EVIDENCE>
```

The unit carries two requirements that the written code violates, one on the status code for an invalid `q` and one on the fields of the response body. No earlier observation in this trajectory carries either requirement. The `app/schemas.py` checkpoint, where the response models were defined, surfaced authentication and request-body units instead.

Subsequent action sequence. Each row groups consecutive actions that serve one change, and Δ gives their positions in the window.

Δ	Event	Target
0	Creation, search reuses `FileListResponse` and yields 422	app/main.py
1–3	`file_type` column and storage category	app/models.py, app/storage.py
4–5	Validation errors mapped to 400 `INVALID_REQUEST`	app/main.py
6–7	`file_type` persisted and filtered	app/main.py
8	`SearchResponse` insert fails on line range	app/schemas.py
9–11	`SearchResponse` inserted after a file view	app/schemas.py
12	`SearchResponse` imported	app/main.py
13–15	Endpoint revised with `query=q`	app/main.py

The status-code requirement is met at $\Delta = 5$ and the response-body requirement over $\Delta = 13$ to $\Delta = 15$, separated by edits that serve an unrelated file-type feature. The failure at $\Delta = 8$ is an editor line-range error rather than feedback about the grounded intent. Every action in the window is a file edit or view, and the first action that executes repository code is the service start at $\Delta = 26$, so the service is neither run nor tested while the two revisions are made.

Interpretation. The deviation observed here is not an omission. The specification stays in the agent's context throughout, and the search handler departs from it by reusing the response model of the listing handler written earlier in the same file and by leaving the default validation behavior of the framework in place. Both decisions are locally plausible and both contradict a stated contract, which is the convention-adoption failure mode identified in the motivation analysis. A deviation of this form is invisible to execution, since the endpoint returns a well-formed response and raises no error. No oracle available inside the generation loop would have reported it, and no execution had occurred when the revisions were made. What exposes the deviation is the delivery of the violated contract at the point where the corresponding code has just been written, and the delivered unit states that contract at the granularity of a status code and an output schema, which is the granularity the domain guidance marks as priority. Although whether the agent would have revised this endpoint without the grounding block is not observed, the revision is confined to the two items stated in the delivered unit and is completed before any execution, which is the behavior that post hoc grounding is designed to produce.