HARNESS ENGINEERING FOR SOFTWARE ENGINEERING VIA MODULAR EXECUTABLE DEV-PRIMITIVES

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) equipped with terminal access have demonstrated strong capabilities in automating software engineering tasks. However, existing agents remain brittle on long-horizon workflows, where they must repeatedly reconstruct program state scattered across source files, configurations, tests, dependencies, and runtime behavior, leading to increasingly long interaction histories, context explosion, and semantic drift. Large repositories further complicate the identification of task-relevant components. To address these challenges, we introduce **Dev-Primitives** (Development Primitives), a modular and executable abstraction that transforms repository components from passive software artifacts into active participants in software engineering. Each Dev-Primitive pairs a repository artifact with a resident LLM, which gives the artifact an agent-native interface grounded in its own implementation and dependencies, enabling natural-language reasoning, inter-component communication, and localized self-modification. Building on Dev-Primitives, we propose **HERMES**, a **H**arness **E**ngineering framework for software enginee**R**ing via **M**odular **E**xecutable Dev-Primitive**S**, which instantiates these primitives at repository scale through a dependency-aware dynamic activation mechanism and a bug diagnosis mechanism that maps execution evidence back to the components that must be revised. Extensive experiments on four software engineering benchmarks demonstrate that HERMES outperforms matched baseline harnesses by 12.4% on average. Moreover, when paired with strong activation and diagnosis models, HERMES, even with Qwen3-8B Dev-Primitives, remains within 4.5% of the homogeneous GPT-5.6 Sol configuration across all four benchmarks, while reducing inference cost by 26.2% on Terminal-Bench 4.0, highlighting the importance of harness design in software engineering agents.

1 INTRODUCTION

Large language models (LLMs) equipped with terminal access have demonstrated remarkable capability in automating real-world software engineering (Yang et al., 2024; Wang et al., 2025). Existing approaches broadly follow two paradigms. Pipeline-based methods prescribe the control flow in advance, decomposing software repair into stages such as fault localization, patch generation, and validation (Xia et al., 2025; Zhang et al., 2024). Agent-based methods instead place an LLM in an interactive execution environment, where it can inspect repositories, execute commands, modify code, and react to runtime feedback through a general action loop (Wang et al., 2024; 2025). Recent benchmarks have meanwhile expanded from isolated issue resolution to tasks that require sustained interaction with repositories, terminals, tests, and runtime systems (Merrill et al., 2026).

Both paradigms remain brittle once such interactions extend over long horizons. Append-only histories and passive compression lead to context explosion, semantic drift, and degraded reasoning (Liu et al., 2026; Wang et al., 2026a). In software repositories, however, the difficulty is not only the length of the history: the relevant program state is scattered across interdependent source files, configurations, and tests, so a change to one file often requires matching changes in the code that calls it, its configuration, and its tests. Each edit also changes the repository itself, so the agent must repeatedly re-inspect affected files and work out what each change requires elsewhere, a burden that no amount of history compression removes. Recent evaluations expose this clearly: on whole-repository migration, only 5.4% of 520 agent runs complete all evaluation stages (Hong et al., 2026), and similar difficulties appear in long-horizon software evolution and repository generation (Le et al.,

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.

2025; Ding et al., 2025). This raises a natural question: **instead of requiring an agent to repeatedly reconstruct distributed program state, can individual software components reason about their own responsibilities and communicate relevant information directly to one another?**

We answer this question with **Dev-Primitives** (Development Primitives), modular and executable abstractions that transform repository components from passive code artifacts into active participants. Each Dev-Primitive wraps a repository artifact with a resident LLM, and this is what makes the artifact agent-native: because the resident model reads the implementation it is attached to, the component can be addressed in natural language and can answer for its own implementation and dependencies. Through this interface it interprets requests, reasons over its own artifact, communicates requirements and constraints to other primitives, and modifies its implementation when necessary, so that component-specific information and reasoning stay with the artifact that owns them.

Dev-Primitives alone, however, do not determine which components a task should involve: repositories may contain thousands of components, and activating all of them would be wasteful. We therefore introduce **HERMES**, a **H**arness **E**ngineering framework for software enginee**R**ing via **M**odular **E**xecutable Dev-Primitive**S**, which adds two mechanisms. Dynamic activation localizes the components likely to implement the reported behavior, follows their dependencies to the callers, configurations, and tests that may need to change with them, and instantiates only the Dev-Primitives in this candidate set, which modify their artifacts and exchange requirements before the repository is executed. Because a single round often leaves the issue unresolved, and execution reports only that the repository still fails rather than which component should be revised, a bug diagnosis mechanism traces the failure back to the components it implicates and reactivates only those primitives, repeating until the repository passes or a revision budget is exhausted.

Extensive experiments on four software engineering benchmarks show that HERMES consistently improves over existing harnesses across issue resolution, whole-repository refactoring, terminal-based tasks, and DevOps workflows. Under the default GPT-5.6 Sol setting with medium reasoning effort, it improves performance by 12.4 percentage points on average. With strong activation and diagnosis models, HERMES using Qwen3-8B Dev-Primitives stays within 4.5 percentage points of the homogeneous GPT-5.6 Sol configuration while reducing inference cost by 26.2% on Terminal-Bench 4.0. These results demonstrate the importance of harness design in software engineering agents. Our contributions are summarized as follows:

- We introduce **Dev-Primitives**, modular executable interfaces that turn repository components into active software engineering participants capable of localized reasoning, natural-language inter-component communication, and self-modification.
- We propose **HERMES**, a **H**arness **E**ngineering framework for software enginee**R**ing via **M**odular **E**xecutable Dev-Primitive**S**, which instantiates Dev-Primitives at repository scale through a dependency-aware dynamic activation mechanism and a bug diagnosis mechanism that maps execution evidence back to the components that must be fixed.
- We evaluate HERMES on four software engineering benchmarks spanning issue resolution, whole-repository refactoring, terminal-based tasks, and DevOps workflows. HERMES improves over matched baseline harnesses by 12.4 percentage points on average, while Qwen3-8B Dev-Primitives remain within 4.5 percentage points of the homogeneous GPT-5.6 Sol configuration and reduce Terminal-Bench 4.0 inference cost by 26.2%.

2 Related Work

Software Engineering Agents and Long-Horizon Evaluation. SWE-agent (Yang et al., 2024), OpenHands (Wang et al., 2025), and CodeAct (Wang et al., 2024) place an LLM in an executable environment where it can inspect repositories, edit code, run commands, and react to runtime feedback, while Agentless (Xia et al., 2025) and AutoCodeRover (Zhang et al., 2024) prescribe explicit localization and repair stages, and other systems distribute work across designer-specified roles or file-level developers under centralized coordination (Chen et al., 2024; Liu et al., 2024; Tao et al., 2024; Wang et al., 2026b). Evaluation has meanwhile moved toward sustained terminal interaction and multi-stage software workflows (Merrill et al., 2026; Tang et al., 2026), and toward repository-scale consistency during software evolution, repository generation, and whole-repository migration (Ding et al., 2025; Hong et al., 2026), where context growth, semantic drift, and state preservation become bottlenecks for long-running agents (Liu et al., 2026; Wang et al., 2026a).

Reusable and Modular Agent Capabilities. One line of work learns software engineering behavior from repository-level trajectories (Pan et al., 2024; Yang et al., 2026; Ma et al., 2024; 2026). Another represents reusable capabilities explicitly through modular abstractions (Jin et al., 2026a;b; Qiu et al., 2025; Li et al., 2026), encapsulating reasoning procedures, skills, or tool interfaces that an agent can invoke and compose across tasks.

Key Differences. These systems differ from Dev-Primitives in the unit to which reasoning capability is attached. Prior agents assign work to designer-specified roles or centrally coordinated file-level tasks, so the reasoning units and the paths along which they exchange information are fixed in advance (Tao et al., 2024; Wang et al., 2026b); prior primitives encapsulate reusable, task-agnostic behaviors (Jin et al., 2026a;b; Li et al., 2026); and context-management methods keep information inside a single trajectory (Liu et al., 2026; Wang et al., 2026a). A Dev-Primitive is instead defined by the artifact it owns, so the set of primitives is induced by the repository and requirements flow along its dependency edges, and HERMES activates these primitives on demand and revises them from execution evidence. Appendix A gives an extended discussion.

3 METHODOLOGY

3.1 OVERVIEW

HERMES is built on **Dev-Primitives**, modular and executable interfaces that transform repository components into active participants in software engineering. Since only a small subset of components is relevant to a given issue, HERMES instantiates Dev-Primitives on demand through a dynamic activation mechanism that localizes likely components and expands along their dependencies. The activated primitives modify their own artifacts and are validated in an execution environment. Because execution reveals only that the repository still fails, HERMES additionally employs a bug diagnosis mechanism that maps the observed failure back to components implicated by the evidence, so that each revision round reactivates only those primitives. An overview of HERMES is shown in Fig. 1.

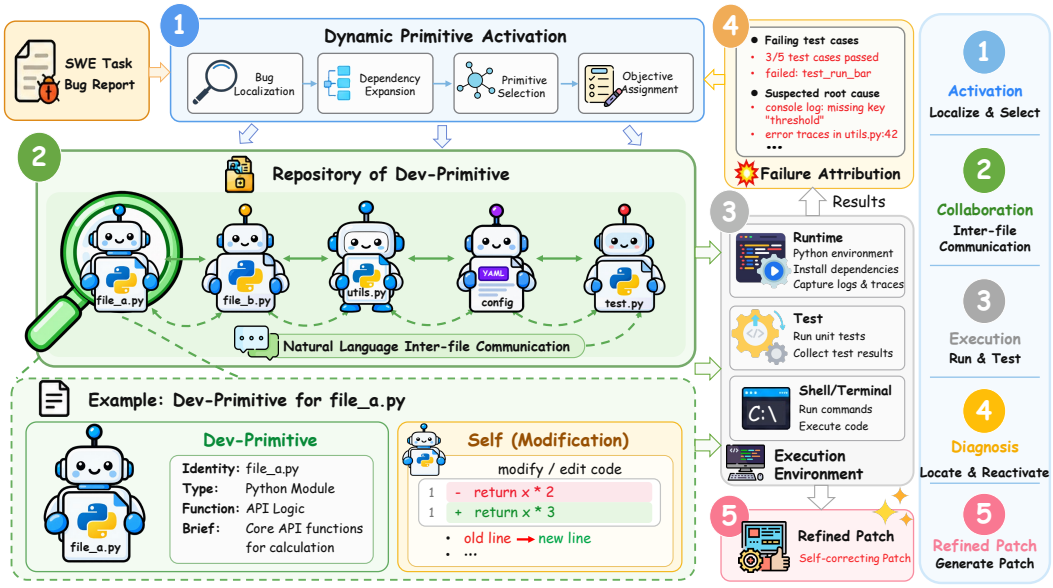


Figure 1: Overview of HERMES. Dynamic activation selects the repository components relevant to the task and instantiates Dev-Primitives, which modify their own artifacts and exchange requirements in natural language. The modified repository is executed, and bug diagnosis maps the observed failure back to the components that must be revised, reactivating only those for the next round.

3.2 DEV-PRIMITIVES

A Dev-Primitive wraps an individual repository component, such as a source file, configuration file, or test file, with a resident LLM. Each primitive has direct access to the artifact it represents and is responsible for reasoning about and modifying that artifact during software engineering. Unlike

conventional agents that repeatedly retrieve repository files as passive context, Dev-Primitives turn these components into active participants that can interpret natural-language requests, communicate implementation requirements with other components, and directly edit their own artifacts.

Formally, let a repository consist of N software components $\mathcal{R} = \{a_i\}_{i=1}^{N}$. Each component a_i is associated with a Dev-Primitive P_i instantiated with an LLM $\mathcal{M}$:

$$(a_i', m_i) = P_i(a_i, x_i, \mathcal{C}_i) = \mathcal{M}([a_i; x_i; \mathcal{C}_i]), \tag{1}$$

where x_i denotes the task assigned to the primitive, $\mathcal{C}_i$ denotes information received from other Dev-Primitives, a_i' is the optionally modified artifact, and m_i is a natural-language message containing task-relevant information or requirements to be communicated to other components. When no modification is required, $a_i' = a_i$.

A Dev-Primitive therefore performs local modification, reasoning over and editing the artifact it represents, and inter-primitive communication, exchanging task-relevant information with other Dev-Primitives through natural language.

Inter-primitive communication. The message m_i carries what the primitive discovered while reasoning over its own implementation and what it therefore requires of other components: interface changes, implementation requirements, dependency updates, configuration constraints, or testing requirements. Messages are directed rather than broadcast. A primitive sends only to the components its local reasoning implicates, which are in practice its callers, callees, configurations, and tests, so the communication topology follows the repository's dependency structure rather than a predefined organization. Messages produced in a round enter the communication context $\mathcal{C}_j$ of their recipients before that round terminates, so a primitive may revise its own modification after receiving a requirement from another component. Participation does not imply modification: a primitive may contribute information about its implementation while leaving its own artifact unchanged.

3.3 Harness Engineering via Modular Executable Dev-Primitives

Only a small subset of a repository is relevant to a given issue, so instantiating every Dev-Primitive would be wasteful. **HERMES**, a **H**arness **E**ngineering framework for software enginee**R**ing via **M**odular **E**xecutable Dev-Primitive**S**, therefore supplies two mechanisms that the abstraction itself does not provide (Fig. 1); complete specifications of each mechanism are in Appendix B.

Dynamic Primitive Activation. Given an issue q and repository $\mathcal{R}$, HERMES produces a plan $\Pi = \{(i, x_i)\}_{i \in \mathcal{A}} = \mathrm{ACTIVATE}(q, \mathcal{R})$, where $\mathcal{A}$ indexes the selected components and x_i is the local objective assigned to P_i. It localizes the components likely to implement the reported behavior and expands along their dependencies to callers, callees, configurations, and tests, inspecting repository structure and retrieving files as needed rather than loading the repository into one context. Only $\{P_i \mid i \in \mathcal{A}\}$ are instantiated, so cost follows the size of $\mathcal{A}$ rather than that of the repository.

Dev-Primitive Collaboration. The activated primitives then operate as a round. Each receives its local objective x_i from the plan together with the messages $\mathcal{C}_i$ addressed to it by other activated primitives, reasons over its own artifact in light of both, and produces $(a_i', m_i) = P_i(a_i, x_i, \mathcal{C}_i)$: an optionally modified artifact and an outgoing message carrying the requirements that its modification imposes elsewhere. Messages circulate only among the primitives in $\mathcal{P}_\mathcal{A}$ and reach their recipients before the round terminates, so a requirement discovered while editing one component can still be satisfied by the components that depend on it within the same round. Cross-file changes are coordinated this way without any single context holding every modified component at once.

Execution Environment. Rather than relying on model-side reasoning about correctness, HERMES evaluates the modified repository $\mathcal{R}'$ in an executable environment, producing $o = \mathrm{EXECUTE}(\mathcal{R}')$: command outputs and exit codes, test outcomes, runtime errors, and available logs and stack traces. These signals expose inconsistencies that appear only once independently modified components are exercised together. Note that held-out evaluation tests are never used during solving (Appendix F.1).

Bug Diagnosis. A single round of modification often leaves the issue unresolved, and execution then reports only whether the repository still fails, not which component should be revised. Bug diagnosis supplies this attribution, returning $v = \mathrm{DIAGNOSE}(q, \Pi, \mathcal{R}', o) \in \{\textsc{pass}, \textsc{fail}\}$ and, when $v = \textsc{fail}$, structured feedback $\phi = (e, c, u)$: the observed failure, the suspected root cause together with the components implicated by the evidence, and revision guidance. The component set named in c is

what keeps revision localized. HERMES revises the plan as $\Pi' = \text{ACTIVATE}(q, \mathcal{R}', \Pi, \phi)$, which may retune objectives, drop components, or activate ones the new evidence implicates, and the loop continues until the repository state is accepted or a budget of B rounds is exhausted.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate HERMES on four benchmarks spanning complementary software engineering settings. SWE-bench Verified (Jimenez et al., 2024) covers real-world GitHub issue resolution with human-validated tasks; SWE Refactor Bench (Hong et al., 2026) targets whole-repository migration and coordinated cross-file modification; Terminal-Bench 4.0 (Merrill et al., 2026) measures performance on interactive terminal-based tasks; and DevOps-Gym (Tang et al., 2026) spans build and configuration, monitoring, issue resolution, test generation, and end-to-end multi-stage workflows.

Baselines and Implementation Details. We follow the official leaderboard settings and evaluation protocols of each benchmark, and run HERMES with the backbones used by the published configurations under each baseline's matched setting. Our evaluations use seven backbones: GPT-5.6 Sol, Terra, and Luna (OpenAI, 2026), Claude Sonnet 5 (Anthropic, 2026b), Claude Opus 5 (Anthropic, 2026a), DeepSeek-V4 (Xu et al., 2026), and Qwen3-8B (Yang et al., 2025). Unless otherwise specified, we use medium reasoning effort where configurable and a revision budget of $B = 3$. In every result table, each group compares a baseline harness with HERMES on the identical backbone and reasoning effort; only the harness differs, and Δ is the percentage-point gain over that row's baseline; HERMES is run once per backbone, Qwen3-8B is evaluated with HERMES only, and blue and orange mark the best and second-best value per column. Backbone assignments, execution isolation, primitive runtime, ablation settings, and baseline provenance are given in Appendix E and Appendix F.

Table 1: Performance and efficiency on SWE-bench Verified. Resolved rates are mean ± standard error over 500 instances.

Backbone	**Baseline Results**			**HERMES Results**			**Δ Res. (p.p.) ↑**
	Resolved (%) ↑	**Cost/Test ($) ↓**	**Latency (s) ↓**	**Resolved (%) ↑**	**Cost/Test ($) ↓**	**Latency (s) ↓**	
Harness	**mini-SWE-agent**			**HERMES**			
GPT-5.6 Sol	**96.20 ± 0.86**	1.15	182.37	**97.00 ± 0.76**	1.80	**205.34**	**+0.80**
GPT-5.6 Terra	95.40 ± 0.94	0.40	180.00	**96.20 ± 0.86**	0.98	**196.71**	**+0.80**
GPT-5.6 Luna	93.00 ± 1.14	**0.04**	201.07	95.60 ± 0.92	**0.10**	218.46	**+2.60**
Claude Fable 5	95.00 ± 0.98	2.05	356.18	96.00 ± 0.88	4.50	381.27	**+1.00**
Claude Opus 5	**97.00 ± 0.76**	1.29	576.99	**97.00 ± 0.76**	2.25	472.18	0.00
Claude Sonnet 5	79.60 ± 1.80	1.49	962.37	85.80 ± 1.56	2.10	824.63	**+6.20**
DeepSeek-V4	77.40 ± 1.87	0.44	634.61	82.80 ± 1.69	0.83	418.52	**+5.40**
GPT-5.5	82.60 ± 1.70	1.36	426.43	85.60 ± 1.57	2.45	398.74	**+3.00**
GPT-5.4 Mini	73.00 ± 1.99	0.51	326.46	82.40 ± 1.70	0.83	347.82	**+9.40**
GPT-5 Mini	60.80 ± 2.19	**0.05**	187.13	80.20 ± 1.78	0.21	213.55	**+19.40**
Claude Opus 4.8	88.60 ± 1.42	1.92	566.95	91.00 ± 1.28	2.65	482.73	**+2.40**
Claude Opus 4.7	82.00 ± 1.72	2.42	441.99	86.00 ± 1.55	3.10	396.52	**+4.00**
Gemini 3.5 Flash	78.80 ± 1.83	0.95	254.13	84.00 ± 1.64	1.42	276.84	**+5.20**
Gemini 3.1 Pro Preview	78.80 ± 1.83	0.78	312.26	83.40 ± 1.67	1.26	328.75	**+4.60**
GPT-5.4 (xhigh)	78.20 ± 1.85	0.80	307.12	83.80 ± 1.65	1.23	322.68	**+5.60**
Claude Opus 4.6 (Thinking)	78.20 ± 1.85	1.22	350.76	84.40 ± 1.62	1.86	341.28	**+6.20**
GPT-5.3-Codex	78.00 ± 1.85	0.46	246.53	83.00 ± 1.68	0.79	267.39	**+5.00**
Claude Sonnet 4.6	77.40 ± 1.87	1.30	511.79	82.80 ± 1.69	1.92	438.64	**+5.40**
Claude Opus 4.5 (Thinking)	76.40 ± 1.90	1.05	320.54	81.80 ± 1.73	1.64	336.28	**+5.40**
Gemini 3 Pro	76.40 ± 1.90	0.77	394.49	82.20 ± 1.71	1.31	381.73	**+5.80**
Harness	**Claude Code**			**HERMES**			
Claude Opus 4.8	85.80 ± 1.56	0.67	**135.40**	91.00 ± 1.28	2.65	482.73	**+5.20**
Harness	**Codex**			**HERMES**			
GPT-5.5	76.40 ± 1.90	0.65	**108.54**	85.60 ± 1.57	2.45	398.74	**+9.20**
Harness	**–**			**HERMES**			
Qwen3-8B	–	–	–	80.60 ± 1.77	**0.13**	229.81	–

4.2 MAIN RESULTS

On SWE-bench Verified. We compare HERMES against leading software engineering agents reported on the SWE-bench Verified leaderboard, including mini-SWE-agent (Yang et al., 2024), Claude Code (Anthropic, 2025), and Codex (OpenAI, 2025). We report the resolved rate together with per-task inference cost and wall-clock latency.

As shown in Table 1, HERMES improves the resolved rate for 19 of the 20 backbones, ties on the remaining one (Claude Opus 5), and outperforms both baselines for the two backbones evaluated against two harnesses. Gains are largest where the baseline harness leaves the most room for coordination, up to $+19.4$ points for GPT-5 Mini, and compress on frontier backbones that already resolve most instances, where HERMES reaches 97.0% with GPT-5.6 Sol against 96.2% under mini-SWE-agent. The additional coordination does not always cost latency: with Claude Sonnet 5, HERMES improves resolution by 6.2 points while reducing average latency from 962.37 s to 824.63 s, and DeepSeek-V4 behaves similarly. HERMES reaches 80.6% with Qwen3-8B, so the framework remains effective with a lightweight backbone; Appendix H provides a complete Qwen3-8B trajectory.

On SWE Refactor Bench. We compare HERMES with representative leaderboard configurations, including Claude Code and Codex, on all 20 whole-repository migration tasks. We report the composite score and per-task inference cost, and include both medium- and high-effort HERMES variants for models that support configurable reasoning effort.

Table 2: Performance and cost on SWE Refactor Bench.

Model	Effort	Baseline Results		HERMES Results		Δ Comp. (p.p.) ↑
		Comp. (%) ↑	Cost ($) ↓	Comp. (%) ↑	Cost ($) ↓	
Harness		**Claude Code**		**HERMES**		
Claude Opus 5	medium	**28.5**	38.4	**36.5**	58.6	**+8.0**
	high	**34.5**	55.7	**41.5**	82.4	**+7.0**
Claude Sonnet 5	medium	15.0	11.9	22.0	22.7	**+7.0**
	high	6.0	24.6	25.5	34.8	**+19.5**
Kimi-K3	max	19.5	28.9	27.0	43.8	**+7.5**
Qwen3.8-Max	max	10.0	14.5	18.5	22.6	**+8.5**
DeepSeek-V4	max	7.0	4.3	15.0	11.6	**+8.0**
GLM-5.2	max	6.5	17.5	14.5	27.2	**+8.0**
Harness		**Codex**		**HERMES**		
GPT-5.6 Sol	medium	6.5	6.0	31.0	24.8	**+24.5**
	high	19.0	7.7	**36.5**	39.6	**+17.5**
GPT-5.6 Terra	medium	4.5	3.4	26.0	14.6	**+21.5**
	high	11.5	4.8	30.0	23.9	**+18.5**
GPT-5.6 Luna	medium	0.0	**1.7**	16.0	**4.9**	**+16.0**
	high	4.0	**1.8**	20.0	**7.2**	**+16.0**
Harness		**–**		**HERMES**		
Qwen3-8B	–	–	–	10.5	8.3	–

As shown in Table 2, whole-repository migration is where HERMES helps most. With GPT-5.6 Sol, the composite score rises from 6.5% to 31.0% under medium effort ($+24.5$) and from 19.0% to 36.5% under high effort ($+17.5$), with similar gains for GPT-5.6 Terra. The pattern holds across models and effort settings: Claude Opus 5 still gains 8.0 and 7.0 points, while GPT-5.6 Luna rises from 0.0% to 16.0% under medium effort. These gains generally require higher inference cost due to iterative execution and revision. Repeating the default GPT-5.6 Sol configuration three times yields 31.0 ± 1.3 (Appendix G), showing that the gains are not due to run-to-run variation.

Table 3: Performance and efficiency on Terminal-Bench 4.0. Resolution rates are mean ± standard deviation over 5 runs; tokens and costs are totals over the full evaluation.

Model	Effort	Baseline Results			HERMES Results			Δ Res. (p.p.) ↑
		Resolution (%) ↑	Tokens ↓	Cost ($) ↓	Resolution (%) ↑	Tokens ↓	Cost ($) ↓	
Harness		**Codex**			**HERMES**			
GPT-5.6 Sol	medium	33.0 ± 3.4	**3.6B**	1.9k	51.8 ± 2.2	6.8B	3.4k	**+18.8**
	max	37.3 ± 3.8	4.4B	2.5k	$\mathbf{55.5 \pm 3.1}$	8.9B	4.7k	**+18.2**
GPT-5.6 Terra	medium	18.2 ± 3.0	4.4B	1.2k	44.5 ± 3.3	7.2B	2.2k	**+26.3**
	max	21.5 ± 3.3	5.7B	1.7k	48.2 ± 2.5	9.1B	2.9k	**+26.7**
GPT-5.6 Luna	medium	14.5 ± 2.6	9.1B	**0.2k**	29.1 ± 2.9	13.2B	**0.4k**	**+14.6**
	max	17.3 ± 2.8	11.6B	**0.3k**	33.3 ± 3.2	15.8B	**0.5k**	**+16.0**
Harness		**Claude Code**			**HERMES**			
Claude Opus 5	medium	$\mathbf{47.0 \pm 3.2}$	5.2B	4.8k	48.5 ± 3.6	7.3B	6.7k	**+1.5**
	max	$\mathbf{51.8 \pm 3.4}$	6.5B	6.0k	$\mathbf{53.0 \pm 2.1}$	9.4B	8.6k	**+1.2**
Claude Fable 5	max	44.5 ± 3.8	3.8B	7.3k	49.7 ± 3.3	**6.2B**	10.1k	**+5.2**
GLM-5.3	max	41.8 ± 3.2	8.7B	2.7k	46.4 ± 2.8	11.5B	3.9k	**+4.6**
Claude Opus 4.8	max	23.6 ± 3.6	6.4B	6.5k	34.2 ± 3.8	9.0B	8.4k	**+10.6**
Claude Sonnet 5	medium	10.6 ± 2.7	15.9B	7.1k	35.8 ± 2.3	13.4B	5.8k	**+25.2**
	max	12.4 ± 3.1	21.6B	9.6k	40.0 ± 3.5	16.8B	7.4k	**+27.6**
Harness		**mini-SWE-agent**			**HERMES**			
DeepSeek-V4	max	28.8 ± 3.4	7.8B	0.7k	39.4 ± 3.1	10.1B	1.1k	**+10.6**
Gemini 3.8 Flash	high	19.1 ± 3.4	17.2B	1.8k	30.3 ± 3.7	20.4B	2.6k	**+11.2**
Gemini 3.7 Flash	high	11.2 ± 2.4	11.1B	1.3k	24.2 ± 3.4	14.8B	1.9k	**+13.0**
Harness		**Grok Build**			**HERMES**			
Grok 4.6	high	20.3 ± 3.1	4.0B	3.6k	31.5 ± 2.5	**6.2B**	5.2k	**+11.2**
Grok 4.5	high	12.4 ± 2.6	**3.4B**	2.1k	25.5 ± 2.2	**5.0B**	3.0k	**+13.1**
Harness		**–**			**HERMES**			
Qwen3-8B	–	–	–	–	27.6 ± 2.7	14.3B	0.7k	–

On Terminal-Bench 4.0. We compare HERMES against leading agent configurations reported on the Terminal-Bench 4.0 leaderboard, including Claude Code, Codex, Grok Build, and mini-SWE-agent.

Table 4: Performance on DevOps-Gym across four software engineering stages. DeepSeek-V4 uses max reasoning effort.

Model	Baseline Results					HERMES Results					Δ Avg. (p.p.) ↑
	Build ↑	Monitor. ↑	Issue ↑	Test ↑	Avg. ↑	Build ↑	Monitor. ↑	Issue ↑	Test ↑	Avg. ↑	
Harness	**Codex**					**HERMES**					
GPT-5.6 Sol	77.78	**38.24**	**40.97**	**42.58**	**49.89**	**83.33**	**44.12**	**46.13**	**48.06**	**55.41**	**+5.52**
GPT-5.6 Terra	74.07	35.29	38.71	40.32	47.10	79.63	41.18	43.55	44.84	52.30	**+5.20**
Harness	**Claude Code**					**HERMES**					
Claude Opus 5	**79.63**	**41.18**	**42.90**	**44.52**	**52.06**	**85.19**	**47.06**	**47.10**	**49.03**	**57.10**	**+5.04**
Claude Sonnet 5	68.52	29.41	34.84	36.45	42.31	75.93	38.24	40.65	41.94	49.19	**+6.88**
Claude Sonnet 4	51.85	20.56	23.87	13.87	27.54	62.96	26.47	29.35	25.48	36.07	**+8.53**
Harness	**OpenHands**					**HERMES**					
DeepSeek-V4	75.93	35.29	39.35	41.29	47.97	81.48	41.18	44.19	45.48	53.08	**+5.11**
Claude Sonnet 4	42.59	14.70	23.87	11.61	23.19	62.96	26.47	29.35	25.48	36.07	**+12.88**
o4-mini	24.07	8.82	10.32	8.70	12.98	35.19	14.71	18.06	16.45	21.10	**+8.12**
Qwen3-Coder-30B	20.37	5.89	13.22	6.13	11.40	31.48	11.76	20.00	14.84	19.52	**+8.12**
Gemini 2.5 Pro	16.66	11.76	10.96	2.90	10.57	29.63	17.65	17.42	13.23	19.48	**+8.91**
DeepSeek-V3.1	11.11	0.00	14.20	3.22	7.13	27.78	8.82	21.61	12.90	17.78	**+10.65**
Harness	**mini-SWE-agent**					**HERMES**					
GPT-5.6 Sol	75.93	35.29	39.35	40.97	47.89	**83.33**	**44.12**	**46.13**	**48.06**	**55.41**	**+7.52**
GPT-5.6 Luna	66.67	29.41	32.90	34.84	40.96	72.22	35.29	38.71	40.32	46.64	**+5.68**
Claude Sonnet 4	29.62	2.91	5.16	0.98	9.67	62.96	26.47	29.35	25.48	36.07	**+26.40**
Harness	**SageAgent**					**HERMES**					
GPT-5.3-Codex	**81.82**	35.29	32.14	37.99	46.81	**85.19**	41.18	37.42	42.58	51.59	**+4.78**
Harness	**Aider**					**HERMES**					
Claude Sonnet 4	5.55	0.00	9.67	2.25	4.37	62.96	26.47	29.35	25.48	36.07	**+31.70**
Harness	**–**					**HERMES**					
Qwen3-8B	–	–	–	–	–	59.26	26.47	31.29	32.90	37.48	–

Terminal-Bench 4.0 contains 66 executable tasks, and each configuration is evaluated over five trials. We report resolution rate, total token consumption, and inference cost.

As shown in Table 3, HERMES consistently improves resolution under matched model–effort configurations, by 18.8 and 18.2 points for GPT-5.6 Sol at medium and max effort and by 26.3 and 26.7 points for GPT-5.6 Terra. The exception is Claude Opus 5, whose baseline configurations are already strong and gain 1.5 and 1.2 points. Most configurations consume additional tokens and cost, but Claude Sonnet 5 is a notable exception: HERMES improves its resolution while reducing both token consumption and inference cost at either effort level.

On DevOps-Gym. We evaluate HERMES on the four DevOps-Gym task categories: build and configuration, monitoring, issue resolving, and test generation. We report the success rate for each category together with the unweighted average across the four categories. In addition to the published DevOps-Gym baselines, we also include same-backbone comparisons with Codex, mini-SWE-agent, and Claude Code to isolate the effect of the harness.

As shown in Table 4, HERMES improves performance across all four DevOps task categories under same-backbone comparison. With GPT-5.6 Sol the average score rises from 49.89% with Codex to 55.41% (+5.52 points), and the gain is distributed across build and configuration, monitoring, issue resolving, and test generation rather than concentrated in a single stage. Claude Opus 5 and GPT-5.6 Luna gain 5.04 and 5.68 points over their respective baseline harnesses, and the improvement extends to earlier and smaller backbones, reaching 37.48% with Qwen3-8B.

Comparison with Related Multi-Agent Frameworks. We additionally compare HERMES with two closely related repository-level multi-agent frameworks in their original evaluation settings.

Table 5: Reference comparison with related repository-level agent frameworks.

Benchmark	Method	Score (%) ↑
SWE-bench	MAGIS	13.9
	HERMES	**36.6**
NL2Repo-Bench	CodeTeam (PE)	34.6
	CodeTeam (SFT)	42.3
	HERMES	**44.9**

As shown in Table 5, MAGIS (Tao et al., 2024) reports 13.9resolution on SWE-bench, while HERMES with GPT-5.6 Luna reaches 36.6%. On NL2Repo-Bench, CodeTeam (Wang et al., 2026b) reports average test pass rates of 34.6% and 42.3% for its prompting and supervised fine-tuning variants, respectively, while HERMES reaches

44.9%. These results further show that HERMES remains effective when compared with repository-level frameworks that explicitly coordinate multiple agents.

4.3 Scaling LLMs Across HERMES Components

We further study how model scale affects different components of HERMES. In addition to homogeneous configurations in which the same backbone is used throughout HERMES, we construct heterogeneous configurations by independently scaling activation and diagnosis while keeping all Dev-Primitives fixed to Qwen3-8B. We conduct the analysis across 4 benchmarks.

Table 6: Effect of backbone assignment across HERMES components. The lower blocks keep Dev-Primitives at Qwen3-8B and scale activation and/or diagnosis; parentheses denote improvement over the homogeneous Qwen3-8B configuration.

Activation	Dev-Primitives	Diagnosis	SWE-bench	SWE Refactor	Terminal-Bench	DevOps-Gym
			Resolved (%) ↑	Composite (%) ↑	Resolved (%) ↑	Avg. (%) ↑
Homogeneous Backbones						
Qwen3-8B	Qwen3-8B	Qwen3-8B	80.6	10.5	27.6	37.48
GPT-5.6 Luna	GPT-5.6 Luna	GPT-5.6 Luna	95.6	16.0	29.1	46.64
DeepSeek-V4	DeepSeek-V4	DeepSeek-V4	82.8	15.0	39.4	53.08
GPT-5.6 Sol	GPT-5.6 Sol	GPT-5.6 Sol	97.0	31.0	51.8	55.41
Diagnosis Scaling						
Qwen3-8B	Qwen3-8B	DeepSeek-V4	84.8 (↑4.2)	14.5 (↑4.0)	44.5 (↑16.9)	41.62 (↑4.14)
Qwen3-8B	Qwen3-8B	GPT-5.5	86.4 (↑5.8)	16.5 (↑6.0)	46.1 (↑18.5)	43.25 (↑5.77)
Qwen3-8B	Qwen3-8B	Claude Sonnet 5	86.0 (↑5.4)	16.0 (↑5.5)	45.5 (↑17.9)	43.01 (↑5.53)
Qwen3-8B	Qwen3-8B	GPT-5.6 Sol	87.2 (↑6.6)	17.5 (↑7.0)	47.9 (↑20.3)	44.36 (↑6.88)
Activation Scaling						
GPT-5.6 Luna	Qwen3-8B	Qwen3-8B	82.4 (↑1.8)	12.0 (↑1.5)	32.7 (↑5.1)	39.82 (↑2.34)
DeepSeek-V4	Qwen3-8B	Qwen3-8B	83.6 (↑3.0)	13.0 (↑2.5)	34.8 (↑7.2)	40.77 (↑3.29)
GPT-5.5	Qwen3-8B	Qwen3-8B	84.4 (↑3.8)	13.5 (↑3.0)	36.7 (↑9.1)	41.58 (↑4.10)
Claude Sonnet 5	Qwen3-8B	Qwen3-8B	84.0 (↑3.4)	13.0 (↑2.5)	36.1 (↑8.5)	41.20 (↑3.72)
GPT-5.6 Sol	Qwen3-8B	Qwen3-8B	85.2 (↑4.6)	14.0 (↑3.5)	38.2 (↑10.6)	42.31 (↑4.83)
Activation + Diagnosis Scaling						
GPT-5.6 Luna	Qwen3-8B	GPT-5.5	89.4 (↑8.8)	20.0 (↑9.5)	47.6 (↑20.0)	46.02 (↑8.54)
GPT-5.5	Qwen3-8B	GPT-5.5	91.2 (↑10.6)	22.5 (↑12.0)	49.1 (↑21.5)	48.73 (↑11.25)
Claude Sonnet 5	Qwen3-8B	Claude Sonnet 5	92.4 (↑11.8)	23.5 (↑13.0)	48.8 (↑21.2)	49.66 (↑12.18)
GPT-5.6 Sol	Qwen3-8B	GPT-5.5	93.6 (↑13.0)	25.0 (↑14.5)	49.7 (↑22.1)	51.44 (↑13.96)
GPT-5.6 Sol	Qwen3-8B	GPT-5.6 Sol	94.2 (↑13.6)	26.5 (↑16.0)	50.6 (↑23.0)	52.30 (↑14.82)

Backbone Scaling. Table 6 shows that HERMES benefits from allocating larger models to dynamic activation and bug diagnosis while keeping the Dev-Primitives lightweight. Strengthening diagnosis consistently helps more than strengthening activation: on Terminal-Bench, using GPT-5.5 only for diagnosis improves resolution from 27.6% to 46.1% (+18.5) against 36.7% (+9.1) for activation alone, and the same ordering holds on SWE-bench Verified (+5.8 versus +3.8). Scaling both closes most of the gap to the homogeneous frontier configuration. What remains is the benefit of scaling the Dev-Primitives themselves, which adds between 1.2 and 4.5 points across the four benchmarks, so the primitives can run on a small model without forfeiting most of the gain.

Table 7: Cost–performance trade-off on Terminal-Bench 4.0. Configurations are ordered as Activation / Dev-Primitives / Diagnosis.

Configuration	Resolved (%)	Cost ($)
Qwen3-8B / Qwen3-8B / Qwen3-8B	27.6	0.70k
Qwen3-8B / Qwen3-8B / GPT-5.5	46.1	1.56k
GPT-5.6 Sol / Qwen3-8B / GPT-5.5	49.7	2.13k
GPT-5.6 Sol / Qwen3-8B / GPT-5.6 Sol	50.6	2.51k
GPT-5.6 Sol / GPT-5.6 Sol / GPT-5.6 Sol	51.8	3.40k

Cost Efficiency. Table 7 gives the cost–performance trade-off on Terminal-Bench 4.0. Using GPT-5.6 Sol for activation, Qwen3-8B for the Dev-Primitives, and GPT-5.5 for diagnosis reaches 49.7% resolution, 2.1 points below the homogeneous GPT-5.6 Sol configuration, at 37.4% lower total inference cost ($2.13k versus $3.40k). Recovering those 2.1 points costs a further $1.27k, split between upgrading diagnosis (+0.9 points for $0.38k) and upgrading the Dev-Primitives (+1.2 points for $0.89k). Against the homogeneous Qwen3-8B configuration, the same setting buys 22.1 points for an additional $1.43k.

4.4 Dev-Primitive Selection Quality

We evaluate whether dynamic activation selects the components a task requires, comparing the activated Dev-Primitives against a target component set derived from the reference patch or, where

no unique reference patch exists, from benchmark-specific supervision. We report selection recall and precision together with the average number of activated primitives, separating solved from failed tasks; target construction and metric definitions are given in Appendix F.4.

Selection Accuracy. Table 8 shows that dynamic activation recovers most target components while activating only a small subset of repository components. Recall is 95.4% on SWE-bench Verified, 83.3% on SWE Refactor Bench, 84.5% on Terminal-Bench, and 85.7% on DevOps-Gym, while the average number of activated Dev-Primitives ranges from 4.7 to 12.6 per task.

Table 8: Dev-Primitive selection quality with GPT-5.6 Sol. DevOps-Gym excludes monitoring tasks.

Benchmark	Recall	Precision	Avg. Act.	Solved R.	Failed R.
SWE-bench Verified	95.4	72.6	4.7	96.1	71.4
SWE Refactor Bench	83.3	68.4	12.6	94.7	78.2
Terminal-Bench 4.0	84.5	64.9	6.8	93.5	74.8
DevOps-Gym	85.7	70.3	5.4	95.0	76.6

Selection Quality and Task Success. Solved tasks consistently exhibit higher selection recall than failed tasks, by 16.5 to 24.7 points across the four benchmarks, linking more complete component selection with downstream task completion. Reference-patch overlap is nonetheless a conservative measure of localization: a task can be resolved along a different file-level path from the developer patch, as in the trajectory of Appendix H.

4.5 Ablation Study

We remove one mechanism at a time while keeping the rest of the framework unchanged, and evaluate GPT-5.6 Sol, Claude Sonnet 5, and Qwen3-8B to test whether the effects persist across backbone families and scales.

Table 9: Component ablation with GPT-5.6 Sol. Metrics are resolved rate for SWE-bench and Terminal-Bench, composite score for SWE Refactor, and average score for DevOps-Gym. Other backbones follow the same ordering (Appendix F.2); setting definitions are in Appendix F.6.

Setting	SWE-bench	Refactor	Terminal	DevOps
HERMES	**97.0**	**31.0**	**51.8**	**55.41**
w/o Inter-Primitive Comm.	94.2 (↓2.8)	23.5 (↓7.5)	46.1 (↓5.7)	50.34 (↓5.07)
w/o On-Demand Activation	95.8 (↓1.2)	28.0 (↓3.0)	49.4 (↓2.4)	53.26 (↓2.15)
w/o Execution Feedback	93.8 (↓3.2)	21.5 (↓9.5)	43.6 (↓8.2)	48.17 (↓7.24)
w/o Diagnosis Feedback	92.6 (↓4.4)	20.0 (↓11.0)	41.2 (↓10.6)	46.73 (↓8.68)

Component Contribution. Table 9 shows a consistent ordering of component importance. Replacing Dev-Primitives with a centralized editor over the same selected components is the most damaging, 14.2 points on SWE Refactor Bench and 13.0 on Terminal-Bench, and a compute-matched editor still trails HERMES by 6.3 points (Appendix G). Among the remaining mechanisms, diagnosis feedback matters most, followed by execution feedback. This is a consequence of the abstraction rather than an argument against it: once edits are artifact-local, a failure observed at the repository level must be routed back to the component that owns the responsible implementation, and a monolithic agent has no such routing problem because it has no owners. Inter-primitive communication matters more on SWE Refactor Bench and Terminal-Bench, 7.5 and 5.7 points, than on SWE-bench Verified, 2.8 points, consistent with its role in coordinating changes that span files. Removing on-demand activation costs at most 3.0 points but raises inference cost by 1.61–1.88×. The same ordering holds for the other two backbones (Appendix F.2).

Revision Budget. Most of the gain is obtained within the first three revision rounds, and increasing B from 3 to 5 adds at most 0.9 points on any benchmark. We therefore use $B = 3$ as the default setting; the full sweep is reported in Appendix F.3.

5 Conclusion

We introduced **Dev-Primitives**, modular executable interfaces that turn repository components from passive artifacts into active participants in software engineering, and **HERMES**, a harness engineering framework that instantiates them through dynamic activation, localized collaboration, environment execution, and diagnosis-driven revision. Across four benchmarks, HERMES improves over matched baseline harnesses by 12.4 percentage points on average, and with strong activation and diagnosis models it stays within 4.5 points of the homogeneous GPT-5.6 Sol configuration while running Qwen3-8B Dev-Primitives, at 26.2% lower inference cost on Terminal-Bench 4.0. These results highlight harness design as a key factor in translating model capability into effective software engineering behavior. Limitations are discussed in Appendix I.

AI USE STATEMENT

We used generative AI tools only for language editing and improving the clarity and consistency of the manuscript. Generative AI tools were not used to formulate the research idea, design the methodology or experiments, or interpret the results. The LLMs evaluated as backbones in Section 4 are objects of study rather than research assistants. All AI-assisted text was reviewed by the authors, who take full responsibility for the final content of the paper.

ETHICS STATEMENT

HERMES is intended for software engineering research in controlled repository environments. As with other code-generating systems, its modifications may be incorrect or insecure and should be validated before deployment. All command execution in our experiments takes place within the isolated environments provided by each benchmark. Our experiments use public research benchmarks and do not involve human subjects or private user data.

REPRODUCIBILITY STATEMENT

Section 3 describes the HERMES framework, and Appendix B provides complete role specifications. Appendix C lists all prompts, and Appendix D gives the end-to-end algorithm. Appendix E lists all backbone models and the deployment configuration of Qwen3-8B. Code and results are available at the anonymous repository in Appendix J.

REFERENCES

Anthropic. Claude code. `https://www.anthropic.com/claude-code`, 2025.

Anthropic. Introducing claude opus 5. `https://www.anthropic.com/news/claude-opus-5`, 2026a.

Anthropic. Introducing claude sonnet 5. `https://www.anthropic.com/news/claude-sonnet-5`, 2026b.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. Coder: Issue resolving with multi-agent and task graphs. arXiv preprint arXiv:2406.01304, 2024.

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.

Deyao Hong, Yizhe Chi, Wenyi Li, Xiaoqiu Wang, Mingju Gao, Kaisen Yang, Bingxiang He, Youjie Zheng, Calvin Xiao, and Qinhuai Na. Swe refactor bench: Can coding agents complete a long-horizon, whole-repository stack migration? arXiv preprint arXiv:2608.23564, 2026.

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, volume 2024, pp. 54107–54157, 2024.

Haibo Jin, Peng Kuang, Ye Yu, Xiaopeng Yuan, and Haohan Wang. Agent primitives: Reusable latent building blocks for multi-agent systems. arXiv preprint arXiv:2602.03695, 2026a.

Haibo Jin, Suijin Wang, Xucheng Yu, Haojing Luo, and Haohan Wang. Harness engineering in llm tool use via agent-native reusable tool primitives. arXiv preprint arXiv:2609.01736, 2026b.

Tue Le, Minh VT Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi DQ Bui. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios. arXiv preprint arXiv:2512.18470, 2025.

Yanzhou Li, Yiran Zhang, Xiaoyu Zhang, Xiaoxia Liu, and Yang Liu. Codeskill: Learning self-evolving skills for coding agents. arXiv preprint arXiv:2605.25430, 2026.

Shukai Liu, Bo Jiang, Jian Yang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon swe-agents. In Findings of the Association for Computational Linguistics: ACL 2026, pp. 20604–20617, 2026.

Yizhou Liu, Pengfei Gao, Xinchen Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. Marscode agent: Ai-native automated bug fixing. arXiv preprint arXiv:2409.00899, 2024.

Murong Ma, Tianyu Chen, Yun Lin, Shuai Lu, Qinglin Zhu, Yeyun Gong, Zhiyong Huang, Peng Cheng, Yan Lu, and Jin Song Dong. From patches to trajectories: Privileged process supervision for software-engineering agents. arXiv preprint arXiv:2605.21996, 2026.

Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. Lingma swe-gpt: An open development-process-centric language model for automated software improvement. arXiv preprint arXiv:2411.00622, 2024.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In International Conference on Learning Representations, volume 2026, pp. 40903–40986, 2026.

OpenAI. Introducing codex. https://openai.com/index/introducing-codex/, May 2025.

OpenAI. Gpt-5.6: Frontier intelligence that scales with your ambition. https://openai.com/index/gpt-5-6/, 2026.

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.

Jiahao Qiu, Xinzhe Juan, Yimin Wang, Ling Yang, Xuan Qi, Tongcheng Zhang, Jiacheng Guo, Yifu Lu, Zixin Yao, Hongru Wang, et al. Agentdistill: Training-free agent distillation with generalizable mcp boxes. arXiv preprint arXiv:2506.14728, 2025.

Yuheng Tang, Kaijie Zhu, Bonan Ruan, Chuqi Zhang, Michael Yang, Hongwei Li, Suyue Guo, Tianneng Shi, Zekun Li, Christopher Kruegel, et al. Devops-gym: Benchmarking ai agents in software devops cycle. In International Conference on Learning Representations, volume 2026, pp. 13021–13045, 2026.

Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. Magis: Llm-based multi-agent framework for github issue resolution. Advances in Neural Information Processing Systems, 37:51963–51993, 2024.

Boshi Wang, Weijian Xu, Yunsheng Li, Xuemei Gao, Yujia Xie, Huan Sun, and Dongdong Chen. Improving code localization with repository memory. In International Conference on Learning Representations, volume 2026, pp. 111266–111285, 2026a.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. arXiv preprint arXiv:2402.01030, 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, 2025.

Yifei Wang, Ruiyin Li, Peng Liang, Qiong Feng, Zengyang Li, Mojtaba Shahin, and Arif Ali Khan. Codeteam: An llm-powered multi-agent framework for repository-level code generation. arXiv preprint arXiv:2606.22082, 2026b.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. Proceedings of the ACM on Software Engineering, 2(FSE):801–824, 2025.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. arXiv preprint arXiv:2606.19348, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. arXiv preprint arXiv:2505.09388, 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 Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. Advances in Neural Information Processing Systems, 38, 2026.

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

A Extended Related Work

Software Engineering Agent Architectures. Recent software engineering agents increasingly combine LLM reasoning with executable environments. Systems such as SWE-agent (Yang et al., 2024), OpenHands (Wang et al., 2025), and CodeAct (Wang et al., 2024) support repository inspection, code editing, command execution, and runtime feedback, while Agentless (Xia et al., 2025) and AutoCodeRover (Zhang et al., 2024) introduce more explicit localization and repair stages. Other systems distribute work across specialized roles or file-level developers under centralized coordination (Chen et al., 2024; Liu et al., 2024; Tao et al., 2024; Wang et al., 2026b).

Long-Horizon Software Engineering. Recent evaluation has increasingly moved beyond isolated issue resolution toward tasks that require sustained interaction with executable environments. Terminal-Bench (Merrill et al., 2026) and DevOps-Gym (Tang et al., 2026) evaluate sustained terminal interaction and multi-stage software workflows, while repository-scale benchmarks expose difficulties in maintaining consistency across components during software evolution, repository generation, and whole-repository migration (Le et al., 2025; Ding et al., 2025; Hong et al., 2026). Related work on context and repository memory further identifies context growth, semantic drift, and state preservation as bottlenecks in long-running software agents (Liu et al., 2026; Wang et al., 2026a). These findings motivate architectures that can maintain task-relevant state near the repository components that generate and consume it, rather than repeatedly reconstructing this state inside a single growing reasoning trajectory.

Reusable and Modular Agent Capabilities. A related line of work studies how agent capabilities can be learned, distilled, reused, or composed across tasks. SWE-Gym (Pan et al., 2024), SWE-smith (Yang et al., 2026), and Lingma SWE-GPT (Ma et al., 2024) learn software engineering behavior from repository-level trajectories, while P2T (Ma et al., 2026) emphasizes high-quality process supervision. Other work represents reusable capabilities explicitly through modular abstractions, including Agent Primitives (Jin et al., 2026a), Tool Primitives (Jin et al., 2026b), AgentDistill (Qiu et al., 2025), and CODESKILL (Li et al., 2026). These methods modularize reusable reasoning procedures, skills, or tool-use capabilities so that they can be invoked and composed by an agent. In particular, Tool Primitives expose tools through agent-native natural-language interfaces, hiding schema resolution and execution details from the calling model. Dev-Primitives adopt a related modular perspective, but attach the capability to mutable repository artifacts rather than reusable task-agnostic behaviors or external tools.

Key Differences. Dev-Primitives differ from prior software-engineering agents and reusable primitives in the unit to which reasoning capability is attached. First, prior systems decompose work across software-engineering roles or centrally assigned file-level tasks (Tao et al., 2024; Wang et al., 2026b). Even when individual developers are responsible for specific files, coordination is organized through a manager, shared plan, or design contract, so both the set of reasoning units and the paths along which they exchange information are fixed by the designer rather than by the repository. Dev-Primitives instead pair each repository component with a resident LLM that reasons over the component's current implementation, functionality, and dependencies. A component can therefore discover requirements during modification and communicate them directly to the components that must satisfy them, without requiring a central agent to reconstruct and relay all component-specific state. Second, existing primitive- and skill-based methods encapsulate reusable, task-agnostic behaviors, reasoning procedures, or tools (Jin et al., 2026a;b; Li et al., 2026), whereas a Dev-Primitive is bound to a specific mutable artifact and evolves together with that artifact throughout execution. Third, context-management approaches primarily compress, retrieve, or reorganize information within a centralized agent trajectory (Liu et al., 2026; Wang et al., 2026a). Dev-Primitives instead keep implementation state associated with the repository components that own it and exchange only task-relevant constraints when coordination is required. Building on this artifact-bound representation, HERMES dynamically activates task-relevant Dev-Primitives and coordinates their modification through environment execution, diagnosis feedback, and revision.

B Detailed Component Specifications

This section provides detailed specifications of the components used in HERMES, including how the dynamic activation and bug diagnosis mechanisms described in Section 3 are implemented with an

LLM. HERMES consists of an activation module, a dynamically activated set of Dev-Primitives, an executable environment, and a diagnosis module. The activation module operates at the repository level to identify task-relevant components and assign localized objectives. Each Dev-Primitive reasons over and modifies the repository artifact it represents, while communicating task-relevant information to other activated primitives. The resulting repository is evaluated in an executable environment, and the diagnosis module converts execution evidence into structured feedback for targeted revision.

B.1 Dynamic Primitive Activation

The activation module converts a software issue into a task-specific plan over repository components. Given issue q and repository $\mathcal{R}$, it performs issue analysis, component localization, dependency analysis, task decomposition, and edit planning:

$$\Pi = \text{ACTIVATE}(q, \mathcal{R}) = \{(i, x_i)\}_{i \in \mathcal{A}}, \tag{2}$$

where $\mathcal{A} \subseteq \{1, \ldots, N\}$ denotes the set of repository components selected for the current task and x_i denotes the local objective assigned to Dev-Primitive P_i.

The activation module performs the following operations:

- **Issue analysis.** Identify the requested behavior, observed failure, and task constraints.
- **Component localization.** Identify source files, tests, configuration files, build files, or other repository components likely involved in the task.
- **Dependency analysis.** Identify relationships among selected components that may require coordinated modification.
- **Task decomposition.** Convert the repository-level issue into localized objectives for individual Dev-Primitives.
- **Edit planning.** Specify the expected modification and coordination requirements for each activated component.

The activation module activates only task-relevant Dev-Primitives:

$$\mathcal{P}_{\mathcal{A}} = \{P_i \mid i \in \mathcal{A}\}. \tag{3}$$

Components outside $\mathcal{A}$ remain inactive unless subsequent execution evidence indicates that they should be considered.

Importantly, $\mathcal{R}$ denotes access to the repository and its structure rather than concatenation of the complete repository into the activation module context. The activation module inspects task-relevant repository information as needed when constructing or revising Π.

During revision, the activation module additionally receives the updated repository state, previous plan, and diagnosis feedback:

$$\Pi' = \text{ACTIVATE}(q, \mathcal{R}', \Pi, \phi). \tag{4}$$

The revised plan may retain successful modifications, update local objectives, remove no-longer-relevant components, or activate additional Dev-Primitives implicated by the execution evidence.

B.2 Dev-Primitives

A Dev-Primitive is associated with an individual repository component a_i, such as a source file, configuration file, build file, or test file. Each primitive directly reasons over the artifact it represents and is responsible for modifying that artifact when required by the assigned objective.

Given a local objective x_i and communication context $\mathcal{C}_i$, Dev-Primitive P_i produces

$$(a_i', m_i) = P_i(a_i, x_i, \mathcal{C}_i), \tag{5}$$

where a_i' denotes the optionally modified artifact and m_i denotes task-relevant information communicated to other activated primitives. If no modification is necessary, $a_i' = a_i$.

Each Dev-Primitive supports two primary capabilities:

- **Local modification.** The primitive inspects the implementation, reasons about the assigned objective, and edits the artifact it represents.

- **Inter-primitive communication.** The primitive communicates interface changes, implementation requirements, dependency updates, configuration constraints, or testing requirements to related Dev-Primitives.

Communication is restricted to task-relevant components rather than broadcast throughout the repository. Messages generated by one primitive are incorporated into the communication context of their target primitives before the collaboration round terminates. A Dev-Primitive may therefore revise its local modification after receiving information from another activated component.

B.3 Execution Environment

After the activated Dev-Primitives complete a modification round, HERMES constructs the updated repository $\mathcal{R}'$ and evaluates it in the benchmark-provided executable environment:

$$o = \mathrm{EXECUTE}(\mathcal{R}') = (o_{\mathrm{shell}}, o_{\mathrm{test}}, o_{\mathrm{runtime}}, o_{\mathrm{trace}})\,. \tag{6}$$

The execution observation contains:

- o_{shell}: shell commands, outputs, exit codes, and build status;
- o_{test}: test results and failing test cases;
- o_{runtime}: runtime behavior, exceptions, and program failures;
- o_{trace}: available logs, stack traces, and execution traces.

The execution environment does not rely on an LLM to predict whether a modification is correct. HERMES evaluates the modified repository through actual execution using the repository snapshots, dependencies, and runtime environments provided by each benchmark. Held-out evaluation tests are not exposed during solving (Appendix F.1).

B.4 Bug Diagnosis

The diagnosis module analyzes execution evidence and determines whether the current repository state satisfies the original issue. Given issue q, current plan Π, modified repository $\mathcal{R}'$, and execution observation o, it produces

$$v = \mathrm{DIAGNOSE}(q, \Pi, \mathcal{R}', o) \in \{\textsc{pass}, \textsc{fail}\}. \tag{7}$$

When the current solution fails, the diagnosis module produces structured feedback ϕ containing:

- concrete execution evidence associated with the failure;
- the repository component or interaction suspected of causing the failure;
- inconsistencies between the implementation and the original objective;
- components that may have been missed during the current planning round;
- specific guidance for the next revision.

The diagnosis module does not modify repository artifacts directly. Its output is returned to the activation module, which performs targeted revision and selectively reactivates the Dev-Primitives required for the next iteration.

The diagnosis module cannot override deterministic execution failures. A trajectory with failing task-visible tests, unsuccessful compilation, or other observed execution failures cannot be marked as PASS solely based on model judgment.

B.5 Revision and Termination

HERMES alternates between activation, Dev-Primitive collaboration, environment execution, and bug diagnosis. When $v = \textsc{fail}$ and the revision budget has not been exhausted, diagnosis feedback ϕ is used to construct a revised plan:

$$\Pi' = \mathrm{ACTIVATE}(q, \mathcal{R}', \Pi, \phi). \tag{8}$$

Previously successful modifications are retained unless contradicted by new execution evidence. The loop terminates when the diagnosis module returns PASS or when the maximum revision budget B is reached.

Here, B denotes the maximum number of revision rounds allowed after the initial plan–modify–execute trajectory. Thus, $B = 0$ still performs an initial planning, collaboration, execution, and bug diagnosis round, but disables subsequent revision.

C Prompt Templates

This section provides the system prompts used by the LLM-based components in HERMES. Dataset-specific issue descriptions, repository contents, execution evidence, and runtime information are inserted into the corresponding placeholders during inference.

C.1 Activation Prompt

Activation Module System Prompt

Role. You are the activation module in HERMES, a software engineering framework based on modular executable Dev-Primitives. Your responsibility is to transform a repository-level software issue into a concrete plan over task-relevant repository components.
You do **not** directly modify repository files. Instead, you identify which Dev-Primitives should be activated, assign each activated primitive a localized objective, identify dependencies among them, and determine how the resulting changes should be validated.
Inputs. You may receive:
- **Issue:** the original software engineering task or bug report;
- **Repository Structure:** available files, directories, modules, tests, configuration files, and build files;
- **Repository Context:** task-relevant implementation and dependency information;
- **Previous Plan:** the plan produced in the previous iteration, if revision is required;
- **Diagnosis Feedback:** execution-grounded failure analysis from the previous iteration, if available.

Issue Analysis. Identify:
- the expected behavior or requested functionality;
- the observed or implied failure;
- explicit constraints specified by the issue;
- likely implementation areas involved;
- whether the task requires coordinated modifications across multiple components.

Component Localization. Identify the repository components most likely relevant to the task. Consider:
- where the reported behavior is implemented;
- callers and callees of relevant functions or classes;
- configuration or build files controlling the behavior;
- tests exercising the affected functionality;
- interfaces shared across repository components;
- dependencies that may require coordinated modification.

Prefer a compact task-relevant component set rather than activating unrelated files.
Task Decomposition. For every activated component, construct a localized objective specifying:
- what behavior should be inspected or modified;
- what constraints should be preserved;
- which other activated components may affect the modification;
- what information may need to be communicated to another Dev-Primitive.

Dependency Analysis. Explicitly identify dependencies among activated components, including interface changes, shared data structures, configuration propagation, caller–callee relationships, and corresponding tests.
Activation Principle. Activate only Dev-Primitives required for the current iteration. Additional components may be activated during revision if execution evidence exposes a previously missed dependency.
Initial Planning Output.

```
{
```

```
  "status": "PLAN",
  "issue_summary": "<concise interpretation of the task>",
  "activated_components": [
    {
      "component": "<repository path>",
      "objective": "<localized objective>",
      "dependencies": [
        "<related component>"
      ],
      "communication_targets": [
        "<related component>"
      ],
      "expected_action": "<inspect | modify | validate>"
    }
  ],
  "execution_plan": [
    "<command, test, or validation step>"
  ],
  "rationale": "<brief explanation of component selection>"
}
```

Revision. When diagnosis feedback is available:
- identify the failure diagnosed by the diagnosis module;
- preserve modifications not contradicted by new evidence;
- revise objectives for components implicated by the failure;
- activate additional components only when required;
- update the execution plan according to the new evidence.

Return the revised plan using the same structure:

```
{
  "status": "REPLAN",
  "replanned": true,
  ...
}
```

Constraints.
- Do not directly edit repository files.
- Do not invent components that do not exist.
- Do not activate unrelated components without a task-specific reason.
- Preserve successful previous modifications unless contradicted by execution evidence.
- Prefer targeted revision over restarting the entire trajectory.

C.2 Dev-Primitive Prompt

Dev-Primitive System Prompt

Role. You are a Dev-Primitive in HERMES. You are responsible for one concrete repository component, such as a source file, configuration file, build file, or test file.
You have direct access to the artifact associated with this component. Your responsibility is to reason about the assigned objective, modify your artifact when necessary, and communicate task-relevant information to other activated Dev-Primitives.
Inputs. You receive:
- **Component:** the repository path or component identifier assigned to you;
- **Artifact:** the current implementation of the component;
- **Local Objective:** the objective assigned by the activation module;
- **Issue Context:** a concise description of the original issue;
- **Incoming Messages:** task-relevant information from other activated Dev-Primitives;

- **Dependency Context:** known relationships between your component and related components.

Local Reasoning. Determine:
- whether the current implementation contributes to the issue;
- whether the assigned objective requires modification;
- which functions, classes, configuration entries, or tests are affected;
- whether a local change modifies an interface or assumption used elsewhere;
- whether incoming messages impose additional implementation constraints.

Local Modification. When modification is required:
- modify only the portions of the artifact necessary for the objective;
- preserve unrelated behavior;
- maintain existing repository conventions;
- avoid unrelated refactoring;
- ensure that the resulting artifact remains syntactically and semantically coherent.

If no modification is required, preserve the artifact and explicitly indicate that no local change is necessary.

Inter-Primitive Communication. Communicate with another activated Dev-Primitive when your local reasoning identifies information affecting its component. Messages may describe:
- changed interfaces;
- assumptions about input or output formats;
- configuration requirements;
- newly introduced or removed dependencies;
- behavioral constraints that tests should verify;
- requests for another component to inspect or modify related behavior.

Output Format.

```
{
  "component": "<repository path>",
  "action": "<modify | no_change>",
  "summary": "<concise explanation of local reasoning>",
  "changes": [
    {
      "location": "<function, class, or code region>",
      "description": "<what was changed and why>"
    }
  ],
  "updated_artifact": "<updated artifact or patch>",
  "messages": [
    {
      "target": "<related component>",
      "message": "<task-relevant requirement>"
    }
  ],
  "validation_notes": [
    "<behavior that should be checked during execution>"
  ]
}
```

Constraints.
- Modify only the artifact assigned to you.
- Do not directly edit another Dev-Primitive's component.
- Communicate cross-component requirements through messages.
- Do not overwrite unrelated modifications already present.
- Do not fabricate details about components you cannot inspect.
- If required information is unavailable, communicate the dependency rather than guessing.

C.3 Inter-Primitive Communication Format

Dev-Primitive Communication Message

When a local modification affects another activated repository component, a Dev-Primitive communicates the requirement using the following structure:

```
{
  "source": "<source component>",
  "target": "<target component>",
  "type": "<interface | dependency | behavior |
           test | configuration>",
  "message": "<concise task-relevant information>",
  "required_action": "<expected action from target>"
}
```

Messages contain only information needed by the receiving component. The receiving Dev-Primitive incorporates the message into its communication context before the current collaboration round terminates and may revise its local modification accordingly.

C.4 Diagnosis Prompt

Diagnosis Module System Prompt

Role. You are the diagnosis module in HERMES. Your responsibility is to evaluate the current repository state using execution evidence and determine whether the current repair satisfies the original software engineering task.
You do not directly modify repository components. Instead, you diagnose failures and provide structured feedback to the activation module.
Inputs. You receive:
- **Original Issue:** the repository-level task;
- **Current Plan:** activated Dev-Primitives and their local objectives;
- **Modified Components:** modifications performed in the current iteration;
- **Shell Output:** command output, exit codes, build logs, and execution status;
- **Test Results:** passing and failing tests with associated failure messages;
- **Runtime Evidence:** exceptions, stack traces, logs, and execution traces;
- **Previous Feedback:** prior diagnosis feedback when evaluating a later revision round.

Evaluation. Check:
- whether the requested behavior has been implemented;
- whether required tests pass;
- whether new regressions have been introduced;
- whether execution failures remain;
- whether cross-component modifications are mutually consistent;
- whether the implementation addresses the original issue rather than only an observed symptom.

Failure Diagnosis. If the solution fails, identify:
- concrete evidence demonstrating failure;
- the most likely evidence-supported cause;
- modified or unmodified components implicated by the failure;
- whether the failure results from incorrect implementation, missing modification, dependency inconsistency, or incomplete localization;
- the revision required in the next iteration.

Pass Output.

```
{
  "status": "PASS",
  "evidence": [
    "<execution or test evidence supporting success>"
  ],
```

```
  "summary": "<why the repository satisfies the issue>"
}
```

Fail Output.

```
{
  "status": "FAIL",
  "failure_evidence": [
    {
      "source": "<test | shell | runtime | trace>",
      "evidence": "<observed failure>"
    }
  ],
  "suspected_causes": [
    {
      "component": "<repository component>",
      "reason": "<evidence-supported diagnosis>"
    }
  ],
  "missing_components": [
    "<component that should be considered>"
  ],
  "revision_guidance": [
    {
      "component": "<component>",
      "action": "<specific revision guidance>"
    }
  ],
  "summary": "<concise diagnosis>"
}
```

Constraints.

- Base the evaluation on execution evidence.
- Do not directly modify repository files.
- Do not mark a trajectory as successful when required tests or benchmark-defined execution checks fail.
- Do not request unrelated modifications without evidence.
- Preserve successful parts of the current trajectory when diagnosing a localized failure.

D END-TO-END EXECUTION FLOW

Algorithm 1 summarizes the full HERMES pipeline. Starting from software issue q, the activation module identifies task-relevant repository components and assigns localized objectives to their Dev-Primitives. The activated primitives collaborate through natural-language communication and modify their local artifacts. HERMES then executes the resulting repository, and the diagnosis module analyzes the execution evidence. Failed execution produces structured feedback for targeted revision. This loop continues until the task passes evaluation or the maximum revision budget is exhausted. Here, B denotes the maximum number of revision rounds after the initial execution.

Collaboration. COLLABORATE executes the activated Dev-Primitives over their local artifacts and routes natural-language messages among task-relevant components. Messages generated by one primitive are incorporated into the communication context of their target primitives before the collaboration round terminates. A primitive may therefore revise its local modification after receiving information from another activated component.

Repository Access. $\mathcal{R}$ denotes repository access rather than concatenation of the complete repository into the activation module context. The activation module can inspect repository structure and task-relevant components as needed when constructing or revising Π.

Algorithm 1 Pseudo-code of HERMES

Require: Software issue q, repository $\mathcal{R}$, Dev-Primitives $\mathcal{P} = \{P_i\}_{i=1}^{N}$, max revision budget B
Ensure: Modified repository or structured failure report

$\Pi \leftarrow$ ACTIVATE$(q, \mathcal{R})$
$b \leftarrow 0$
while $b \leq B$ **do**
  $\mathcal{A}, \{x_i\}_{i\in\mathcal{A}} \leftarrow \Pi$
  $\{a'_i\}_{i\in\mathcal{A}} \leftarrow$ COLLABORATE$(\{P_i\}_{i\in\mathcal{A}}, \{x_i\}_{i\in\mathcal{A}})$
  $\mathcal{R}' \leftarrow$ UPDATEREPOSITORY$(\mathcal{R}, \{a'_i\}_{i\in\mathcal{A}})$
  $o \leftarrow$ EXECUTE$(\mathcal{R}')$
  $v \leftarrow$ DIAGNOSE$(q, \Pi, \mathcal{R}', o)$
  if $v =$ `pass` **then**
    return $\mathcal{R}'$
  end if
  if $b = B$ **then**
    return FAILUREREPORT$(q, \mathcal{R}', o)$
  end if
  $\phi \leftarrow$ DIAGNOSE.FEEDBACK$(q, \Pi, \mathcal{R}', o)$
  $\Pi \leftarrow$ ACTIVATE$(q, \mathcal{R}', \Pi, \phi)$
  $\mathcal{R} \leftarrow \mathcal{R}'$
  $b \leftarrow b + 1$
end while

Execution. After collaboration, HERMES applies the resulting component modifications to form $\mathcal{R}'$ and evaluates the repository in the executable benchmark environment. Shell output, test outcomes, runtime behavior, and available traces are passed to the diagnosis module as execution evidence.

Revision. When execution fails, the diagnosis module produces feedback ϕ identifying the observed failure and the components implicated by the evidence. The activation module uses ϕ together with the updated repository and previous plan to revise only the affected portion of the trajectory. Successful modifications are retained unless contradicted by new execution evidence.

Termination. $B = 0$ permits one initial plan–collaborate–execute–diagnose trajectory but disables subsequent revision. For $B > 0$, HERMES performs at most B additional revision rounds after the initial execution. The diagnosis module cannot override deterministic execution failures: failing task-visible tests, unsuccessful compilation, or other observed execution failures prevent a PASS decision.

E BACKBONE MODELS

Qwen3-8B is served locally with Ollama on NVIDIA A40 GPUs, using the model's default sampling parameters (temperature 0.6, top-p 0.95, top-k 20) and Ollama's 32K context window. All other models are accessed through their official APIs.

F IMPLEMENTATION AND EVALUATION DETAILS

F.1 EXECUTION ISOLATION AND SETTINGS

LLM Instantiation of Each Mechanism. Both mechanisms described in Section 3 are implemented by prompting an LLM. Dynamic activation is realized by a single LLM call chain that inspects the repository structure, retrieves candidate files, and returns the selected component set $\mathcal{A}$ together with a local objective x_i for each selected component; it is not a static retriever, since the dependency expansion is carried out by the model reading the candidate files rather than by a precomputed call graph. Bug diagnosis is likewise realized by a single LLM call that receives the issue, the current plan, the modified repository state, and the execution observation, and returns the verdict v together with structured feedback $\phi = (e, c, u)$. Each Dev-Primitive is instantiated with its own LLM, which may differ from the models used by the two mechanisms; the heterogeneous configurations

Table 10: Backbone models used in our experiments. **Bold** denotes primary backbones. Scaling covers Tables 6–7, and Ablation covers Table 9. GPT-5.6 Sol is additionally used in Tables 8 and 12. Reasoning effort follows the setting reported in each table.

Model	SWE-bench	Refactor	T-Bench	DevOps	Scaling	Ablation
OpenAI						
GPT-5.6 Sol	✓	✓	✓	✓	✓	✓
GPT-5.6 Terra	✓	✓	✓	✓		
GPT-5.6 Luna	✓	✓	✓	✓	✓	
GPT-5.5	✓				✓	
GPT-5.4, GPT-5.4 Mini, GPT-5 Mini	✓					
GPT-5.3-Codex	✓			✓		
o4-mini				✓		
Anthropic						
Claude Opus 5	✓	✓	✓	✓		
Claude Sonnet 5	✓	✓	✓	✓	✓	✓
Claude Fable 5, Claude Opus 4.8	✓		✓			
Claude Opus 4.7 / 4.6 / 4.5, Claude Sonnet 4.6	✓					
Claude Sonnet 4				✓		
Google						
Gemini 3.5 Flash, Gemini 3.1 Pro Preview, Gemini 3 Pro	✓					
Gemini 3.8 Flash, Gemini 3.7 Flash			✓			
Gemini 2.5 Pro				✓		
Open-weight and other providers						
DeepSeek-V4	✓	✓	✓	✓	✓	
Qwen3-8B	✓	✓	✓	✓	✓	✓
DeepSeek-V3.1, Qwen3-Coder-30B				✓		
Qwen3.8-Max, Kimi-K3, GLM-5.2		✓				
GLM-5.3, Grok 4.6, Grok 4.5			✓			

in Section 4.3 exploit this separation. Neither mechanism can override deterministic execution outcomes: a trajectory with failing task-visible tests, unsuccessful compilation, or other observed execution failures cannot be accepted on the basis of model judgment alone. The prompts used for both mechanisms are listed in Appendix C.

Execution and Evaluation Isolation. HERMES separates the execution environment available during task solving from the benchmark evaluator used to compute the final score. During a trajectory, dynamic activation, the Dev-Primitives, and bug diagnosis may use only information available from the task repository and its execution environment, including repository-provided tests, build commands, linters, type checkers, runtime outputs, stack traces, logs, and reproduction scripts constructed during solving. Benchmark-held-out evaluation tests and final grading outcomes are not exposed to HERMES during task execution.

For SWE-bench Verified, HERMES may execute tests already available in the repository or construct task-specific reproduction tests from the issue description and repository context. The tests used by the benchmark evaluator, including those associated with `FAIL_TO_PASS` and `PASS_TO_PASS`, are used only after HERMES terminates. We follow the same separation for SWE Refactor Bench, Terminal-Bench 4.0, and DevOps-Gym: benchmark-specific graders and held-out evaluation checks are reserved for final scoring rather than being returned as feedback during solving.

Reasoning Effort. We use medium reasoning effort by default for models that support configurable reasoning effort. High- or maximum-effort configurations are used only when explicitly indicated in the corresponding experiment. Unless otherwise stated, the component ablation, revision, and backbone-scaling studies therefore use the default medium-effort setting.

F.2 Complete Component Ablation

Table 11 reports the component ablation for all three backbones. Section 4 shows the GPT-5.6 Sol block; the ordering of component importance is consistent across Claude Sonnet 5 and Qwen3-8B.

Table 11: Complete component ablation of HERMES across three backbone models. Parentheses denote absolute changes relative to complete HERMES with the same backbone.

Backbone	Setting	SWE-bench	SWE Refactor	Terminal-Bench	DevOps-Gym
		Resolved (%) ↑	Composite (%) ↑	Resolved (%) ↑	Avg. (%) ↑
		GPT-5.6 Sol			
GPT-5.6 Sol	**HERMES**	**97.0**	**31.0**	**51.8**	**55.41**
GPT-5.6 Sol	w/o Inter-Primitive Communication	94.2 (↓2.8)	23.5 (↓7.5)	46.1 (↓5.7)	50.34 (↓5.07)
GPT-5.6 Sol	w/o On-Demand Activation	95.8 (↓1.2)	28.0 (↓3.0)	49.4 (↓2.4)	53.26 (↓2.15)
GPT-5.6 Sol	w/o Execution Feedback	93.8 (↓3.2)	21.5 (↓9.5)	43.6 (↓8.2)	48.17 (↓7.24)
GPT-5.6 Sol	w/o Diagnosis Feedback	92.6 (↓4.4)	20.0 (↓11.0)	41.2 (↓10.6)	46.73 (↓8.68)
		Claude Sonnet 5			
Claude Sonnet 5	**HERMES**	**85.8**	**22.0**	**35.8**	**49.19**
Claude Sonnet 5	w/o Inter-Primitive Communication	82.8 (↓3.0)	16.5 (↓5.5)	31.2 (↓4.6)	44.82 (↓4.37)
Claude Sonnet 5	w/o On-Demand Activation	84.4 (↓1.4)	20.0 (↓2.0)	34.2 (↓1.6)	47.63 (↓1.56)
Claude Sonnet 5	w/o Execution Feedback	81.8 (↓4.0)	14.5 (↓7.5)	28.8 (↓7.0)	42.96 (↓6.23)
Claude Sonnet 5	w/o Diagnosis Feedback	80.6 (↓5.2)	13.0 (↓9.0)	26.7 (↓9.1)	41.18 (↓8.01)
		Qwen3-8B			
Qwen3-8B	**HERMES**	**80.6**	**10.5**	**27.6**	**37.48**
Qwen3-8B	w/o Inter-Primitive Communication	77.8 (↓2.8)	6.5 (↓4.0)	23.0 (↓4.6)	33.74 (↓3.74)
Qwen3-8B	w/o On-Demand Activation	79.4 (↓1.2)	9.0 (↓1.5)	26.1 (↓1.5)	36.11 (↓1.37)
Qwen3-8B	w/o Execution Feedback	76.8 (↓3.8)	5.5 (↓5.0)	20.9 (↓6.7)	31.62 (↓5.86)
Qwen3-8B	w/o Diagnosis Feedback	75.6 (↓5.0)	4.5 (↓6.0)	19.1 (↓8.5)	29.83 (↓7.65)

F.3 Revision Budget

Table 12 reports the effect of the revision budget B, with changes measured relative to the default $B = 3$. Increasing B from 0 to 3 adds 12.5 points on SWE Refactor Bench, 12.1 on Terminal-Bench, 9.50 on DevOps-Gym, and 4.8 on SWE-bench Verified, whereas increasing B from 3 to 5 adds only 0.5, 0.9, 0.47, and 0.2 points respectively.

Table 12: Effect of the revision budget B, with changes measured relative to the default $B = 3$.

B	SWE-bench	SWE Refactor	Terminal-Bench	DevOps-Gym
0	92.2 (↓4.8)	18.5 (↓12.5)	39.7 (↓12.1)	45.91 (↓9.50)
1	94.6 (↓2.4)	24.0 (↓7.0)	45.2 (↓6.6)	50.17 (↓5.24)
2	96.4 (↓0.6)	28.5 (↓2.5)	48.8 (↓3.0)	53.72 (↓1.69)
3	**97.0**	**31.0**	**51.8**	**55.41**
4	97.2 (↑0.2)	31.5 (↑0.5)	52.4 (↑0.6)	55.79 (↑0.38)
5	97.2 (↑0.2)	31.5 (↑0.5)	52.7 (↑0.9)	55.88 (↑0.47)

F.4 Dev-Primitive Selection Protocol

We evaluate whether dynamic activation selects the repository components required for each task. For benchmarks with reference patches, we use the files modified by the developer patch as the target component set and compare them with the Dev-Primitives selected by HERMES. For benchmarks without a unique reference patch, we construct the target set from benchmark-specific supervision: for SWE Refactor Bench, we use the repository components implicated by the task specification and grading artifacts; for Terminal-Bench, we use the files and artifacts accessed or modified by the oracle solution; and for DevOps-Gym, we use the reference artifacts associated with build, issue-resolving, and test-generation tasks, excluding monitoring tasks from file-level selection evaluation. Let $\mathcal{G} \subseteq \{1, \ldots, N\}$ denote the indices of the target components and $\mathcal{A}$ the indices of Dev-Primitives activated across all planning and revision rounds. We report selection recall, $|\mathcal{A} \cap \mathcal{G}|/|\mathcal{G}|$, selection precision, $|\mathcal{A} \cap \mathcal{G}|/|\mathcal{A}|$, and the average number of activated Dev-Primitives per task. We additionally separate solved and failed tasks to examine how selection quality relates to downstream task completion. Recall and precision, including the solved and failed breakdowns, are micro-averaged over components

within each benchmark. For SWE Refactor Bench, we regard a task as solved when it receives a non-zero composite score, indicating that it passes the migration audit and behavioral checks and reaches the verifier stage.

F.5 DEV-PRIMITIVE RUNTIME

Primitive State Across Rounds. Each Dev-Primitive is associated with a concrete repository component and persists across revision rounds within the same task. Its state contains the current version of the underlying artifact together with task-relevant communication accumulated during the trajectory. Model hidden states are not preserved across calls. When a primitive is activated again, it receives the current artifact, its assigned objective, and the communication context relevant to that component.

Inter-Primitive Communication. Dev-Primitives communicate through explicit natural-language messages. A primitive can communicate implementation requirements, interface changes, dependency information, expected behavior, or other constraints to another active primitive. Each message is addressed by the sending primitive to a specific target and placed verbatim into the target's communication context $\mathcal{C}_j$. No central agent interprets, filters, or rewrites messages, and they are not merged into a single global reasoning trajectory.

Communication is interleaved with local modification rather than performed as a fixed all-to-all stage. A primitive sends a message when its local task depends on another component or when its modification introduces a requirement that another component must satisfy. This allows communication to follow repository dependencies encountered during execution rather than requiring every activated primitive to communicate with every other primitive.

On-Demand Activation. The activation module activates only the Dev-Primitives selected as relevant to the current plan. Inactive components remain available in the repository but do not receive model calls or participate in communication during that round. The active set may change across revision rounds: execution and diagnosis feedback may cause the activation module to activate additional components, remove previously activated components, or revise their assigned objectives.

Creation of New Components. When solving a task requires a new persistent source, test, configuration, build, or other repository file, an active Dev-Primitive may create the artifact. HERMES subsequently registers the new artifact as a Dev-Primitive so that it can participate in later communication and revision rounds. Temporary outputs, caches, logs, and other transient execution artifacts are treated as environment observations rather than Dev-Primitives.

Component Granularity. For repository-level software engineering tasks, we use files as the default Dev-Primitive granularity. A component may therefore correspond to a source file, test file, configuration file, build specification, script, or another persistent repository artifact. This provides a stable mapping between executable repository artifacts and Dev-Primitives without introducing the overhead of assigning separate agents to individual functions or symbols.

Terminal-Bench tasks do not always follow the structure of a conventional software repository. We therefore treat persistent filesystem artifacts that can be independently inspected or modified as components, including source files, shell scripts, configuration files, service definitions, build files, and other task-relevant artifacts. Runtime processes, terminal outputs, and transient system state are treated as environment observations rather than Dev-Primitives.

F.6 ABLATION SETTINGS

For the component ablation study, we use the complete HERMES configuration as the reference and disable one mechanism at a time while keeping the backbone model, reasoning effort, revision budget, and benchmark environment unchanged.

w/o Inter-Primitive Communication. Dev-Primitives retain their local reasoning and editing capabilities but cannot exchange messages with one another. Each primitive receives its objective from the activation module and independently modifies its own component. Information across repository components can therefore only be recovered indirectly through subsequent planning and execution.

w/o On-Demand Activation. We retain the same initial task analysis and component identification procedure, but remove on-demand invocation of Dev-Primitives. In the complete HERMES configuration, the activation module activates a primitive only when its associated component is required by the current plan, and the active set may change after execution feedback and revision. In this ablation, all Dev-Primitives identified during the initial task analysis are activated at the beginning of the trajectory and remain active throughout subsequent planning and revision rounds.

w/o Execution Feedback. HERMES retains activation, the Dev-Primitives, inter-primitive communication, and diagnosis, but execution observations generated after repository modifications are not provided for subsequent revision. The diagnosis module therefore evaluates the modified repository against the task and current plan through static inspection alone, and cannot use test failures, runtime errors, command outputs, or execution traces to guide later rounds.

w/o Diagnosis Feedback. Environment execution remains enabled and its raw outputs remain available for subsequent planning, but the dedicated diagnosis module does not analyze the execution results or produce structured failure diagnosis and revision guidance. Revision therefore relies on the activation module's direct interpretation of the available execution evidence. This setting differs from $B = 0$: removing diagnosis feedback still permits subsequent planning and execution rounds, whereas $B = 0$ disables revision after the initial round.

Revision Budget. For the revision study, we vary B while keeping all other components unchanged. $B = 0$ performs only the initial planning, modification, and execution cycle, whereas $B > 0$ permits up to B additional revisions based on execution and diagnosis feedback. We use $B = 3$ as the default setting in the main experiments.

F.7 Benchmark-Specific Evaluation Protocol

SWE-bench Verified. We evaluate HERMES on the 500 human-validated tasks in SWE-bench Verified using the official repository snapshots and evaluation procedure. During task solving, HERMES operates on the issue description, repository contents, and task-visible execution environment. Repository-provided tests and reproduction scripts constructed during solving may be executed, whereas benchmark evaluation tests are reserved for final scoring. We report the percentage of successfully resolved tasks.

SWE Refactor Bench. We evaluate all 20 whole-repository migration tasks and report the benchmark composite score. HERMES operates on the provided repository, migration specification, and task-visible execution environment, while the benchmark grader is used only for final evaluation. For analyses that divide tasks into solved and failed groups, we regard a task as solved when it obtains a non-zero composite score, indicating that the migration passes the required audit and behavioral checks and reaches the verifier stage.

Terminal-Bench 4.0. We evaluate the 66 Terminal-Bench 4.0 tasks using five runs per task. HERMES interacts with the benchmark-provided terminal environment and may inspect or modify task-visible filesystem artifacts. The benchmark grader is used only to determine the final task outcome and is not exposed as execution feedback. We report resolution as the mean across the five runs together with its standard deviation. Token counts and inference costs are accumulated over the complete evaluation trajectories.

DevOps-Gym. We evaluate four DevOps-Gym task categories: build and configuration, monitoring, issue resolving, and test generation. HERMES interacts with the task-visible repository and runtime environment during solving, while benchmark-specific grading artifacts are used only for final evaluation. We report the success rate for each category and the unweighted average across the four categories.

F.8 Baseline Provenance

Our baseline tables contain both publicly reported results and controlled baseline runs. When the required model, harness, reasoning-effort setting, and benchmark version are available from the corresponding benchmark leaderboard or published evaluation, we directly use the reported result. When an exact matched model–effort configuration is unavailable, we run the corresponding baseline harness under the same benchmark snapshot and evaluation protocol used for HERMES.

For SWE-bench Verified, we use reported results for available mini-SWE-agent, Claude Code, and Codex configurations and use controlled runs where an exact matched configuration is required. For SWE Refactor Bench, published Claude Code and Codex results are supplemented with matched configurations evaluated under the same 20-task protocol. For Terminal-Bench 4.0, publicly reported high- or maximum-effort results are retained when available, while additional medium-effort configurations are evaluated separately to provide matched comparisons with the default HERMES setting. For DevOps-Gym, we include the baseline systems reported by the benchmark together with same-backbone controlled runs used for direct harness comparison.

All HERMES results are obtained with our implementation. Unless otherwise specified, HERMES uses medium reasoning effort and a maximum revision budget of $B = 3$. Token counts and inference costs include all activation, Dev-Primitive, and diagnosis calls over the complete trajectory.

G Additional Analysis of Dev-Primitives

The main experiments evaluate HERMES as a complete harness. We further conduct controlled analyses to isolate the contribution of the Dev-Primitive abstraction itself and to distinguish it from several alternative explanations, including component localization, inference budget, and centralized context capacity. We additionally analyze the computational effect of on-demand activation and run-to-run variation on SWE Refactor Bench.

Dev-Primitives vs. a Centralized Editor. We construct a controlled variant, denoted w/o Dev-Primitives (single editor), that removes the Dev-Primitive abstraction while leaving the remaining HERMES pipeline unchanged. We use the same activation module, execution environment, diagnosis module, revision budget $B = 3$, GPT-5.6 Sol backbone with medium reasoning effort, and, critically, the same component set $\mathcal{A}$ selected by the activation module. Instead of assigning each selected component to its corresponding Dev-Primitive, a single Editor receives the complete plan $\Pi = \{(i, x_i)\}_{i \in \mathcal{A}}$ together with the contents of all selected components and modifies them jointly. This control preserves component localization while removing component-local reasoning and direct communication between components. It therefore tests whether Dev-Primitives provide benefits beyond exposing the same selected components to a centralized editing agent.

Table 13: Controlled comparison between HERMES and a centralized single-editor variant. Both use the same activation-selected component set, execution environment, diagnosis module, GPT-5.6 Sol backbone with medium reasoning effort, and $B = 3$. Tok. and Cost are normalized to HERMES on each benchmark.

Setting	SWE-bench Verified			SWE Refactor Bench			Terminal-Bench 4.0			DevOps-Gym		
	Res. ↑	Tok.	Cost	Comp. ↑	Tok.	Cost	Res. ↑	Tok.	Cost	Avg. ↑	Tok.	Cost
HERMES	**97.0**	1.00×	1.00×	**31.0**	1.00×	1.00×	**51.8**	1.00×	1.00×	**55.41**	1.00×	1.00×
Single editor	93.4	0.86×	0.85×	16.8	0.90×	0.89×	38.8	0.86×	0.84×	44.21	0.87×	0.86×
Δ	**−3.6**			**−14.2**			**−13.0**			**−11.20**		

As shown in Table 13, replacing Dev-Primitives with a centralized Editor reduces performance on all four benchmarks. The reduction is smaller on SWE-bench Verified, from 97.0% to 93.4%, but becomes substantially larger on benchmarks requiring coordination across multiple components: from 31.0% to 16.8% on SWE Refactor Bench, from 51.8% to 38.8% on Terminal-Bench, and from 55.41% to 44.21% on DevOps-Gym. Because both configurations operate on the same activation-selected component set, the difference cannot be attributed to component localization. Instead, the results are consistent with the benefit of component-local reasoning and direct propagation of cross-component requirements. The centralized Editor makes fewer model calls because it processes all selected components jointly, using 10–14% fewer tokens and 11–16% lower cost than HERMES, but this reduction is accompanied by considerably lower task performance.

Controls for Compute and Context Capacity. Since the single Editor uses less inference than HERMES, we conduct two further controls on Terminal-Bench 4.0 (Table 14) to examine whether the difference can instead be explained by inference budget or centralized context capacity.

Compute-matched centralized Editor. We allocate approximately the same inference budget as HERMES to the GPT-5.6 Sol centralized Editor. The Editor generates multiple candidate revisions

within each planning round, evaluated with the same execution environment and diagnosis mechanism, and candidate generation is increased until total token usage and inference cost approximately match HERMES.

Table 14: Compute and context controls on Terminal-Bench 4.0. All settings use the same activation-selected component set, execution environment, diagnosis module, and $B = 3$; GPT-5.6 Sol uses medium reasoning effort.

Setting	Resolution (%) ↑	Tokens	Cost ($)
GPT-5.6 Sol			
Single editor	38.8	5.9B	2.90k
Single editor + compute matching	45.5	6.7B	3.35k
HERMES	**51.8**	6.8B	3.40k

Matching the inference budget improves the GPT-5.6 Sol Editor from 38.8% to 45.5%, recovering 6.7 points, but it remains 6.3 points below HERMES at essentially the same token and monetary budget. Additional inference therefore accounts for part, but not all, of the gap.

Efficiency of On-Demand Activation. HERMES first identifies task-relevant components during planning, but does not require every identified Dev-Primitive to remain active throughout the trajectory. Instead, primitives are invoked when their components are needed by the current plan, and the active set may change after execution feedback and revision. In the w/o On-Demand Activation variant, the same initial task analysis and component identification are retained, but all Dev-Primitives identified during the initial analysis are activated at the beginning of the trajectory and remain active throughout subsequent planning and revision rounds. This ablation therefore does not activate every file in the repository; it removes the ability to selectively invoke and deactivate identified Dev-Primitives as the task evolves.

We distinguish activated components from primitive calls. Table 8 reports the number of distinct Dev-Primitives activated for a task, whereas Table 15 counts every model invocation over the complete trajectory. Since the same primitive may be called multiple times for local revision or communication across revision rounds, primitive calls substantially exceed distinct activated components.

Table 15: Effect of on-demand activation. Primitive calls include repeated invocations during communication and revision; cost is relative to HERMES.

Benchmark	Avg. Primitive Calls		Relative Cost w/o Activation
	HERMES	w/o Activation	
SWE-bench Verified	9.6	28.7	1.61×
SWE Refactor Bench	25.1	64.3	1.88×
Terminal-Bench 4.0	14.2	39.6	1.73×
DevOps-Gym	11.7	33.2	1.67×

Across the four benchmarks, on-demand activation reduces the average number of primitive calls from 28.7–64.3 to 9.6–25.1 per task, with the largest reduction on SWE Refactor Bench, where whole-repository migrations involve more potentially relevant components. Disabling on-demand activation also increases inference cost by 1.61–1.88× while reducing task performance in Table 9. Dynamic activation thus limits both unnecessary model calls and interference from components that are no longer required at the current stage of the task.

Run-to-Run Variation on SWE Refactor Bench. SWE Refactor Bench contains only 20 whole-repository migration tasks, making its aggregate score more sensitive to stochastic variation than the larger benchmarks. Repeating the default GPT-5.6 Sol configuration (medium reasoning effort, $B = 3$) three times yields composite scores of 29.5%, 32.0%, and 31.5%, i.e., 31.0 ± 1.3. We report this variation explicitly because the small number of tasks makes individual trajectories more influential on the aggregate score.

H CASE STUDY: A COMPLETE HERMES TRAJECTORY WITH QWEN3-8B

We illustrate HERMES through a successful trajectory on `django__django-13512` from SWE-bench Verified, using Qwen3-8B as the backbone under the default revision budget $B = 3$. The issue reports that non-ASCII characters stored in a `JSONField` are displayed in the Django admin as escaped Unicode sequences rather than as the original characters. Figure 1 gives the general workflow; here we follow the concrete decisions made during this trajectory.

Planning and activation. The activation module first identifies six repository components whose behavior may contribute to the issue and assigns each an `expected_action` according to the schema in Appendix C. As shown in Table 16, four components are assigned `modify`, while two are assigned `inspect`. The latter remain active because their local behavior is relevant to understanding the rendering and encoding paths, even though the activation module does not expect them to edit their artifacts.

Table 16: Activated Dev-Primitives and their initial component-local responsibilities. Activation does not require a component to be modified.

Component	Action	Initial local objective
`db/models/fields/json.py`	modify	Preserve non-ASCII characters when preparing `JSONField` values
`forms/fields.py`	modify	Update `prepare_value` to preserve non-ASCII characters
`forms/utils.py`	modify	Avoid ASCII escaping in JSON-formatted form output
`core/serializers/json.py`	modify	Preserve non-ASCII characters in core JSON serialization
`forms/widgets.py`	inspect	Inspect the rendering path and communicate relevant constraints
`utils/encoding.py`	inspect	Inspect shared encoding behavior and communicate Unicode-related constraints

The resulting plan already identifies the central constraint: JSON values that flow through the relevant form and serialization paths should preserve non-ASCII characters. The activation module also records dependencies between the selected components, including the relationship between model-field JSON preparation and core serialization.

Collaboration. The six activated Dev-Primitives then exchange 15 directed natural-language messages. These messages are not broadcast globally; each primitive sends requirements only to components implicated by its local reasoning. Table 17 shows three representative exchanges.

Table 17: Representative directed messages from the collaboration round. Messages are abridged for space.

From	To	Message (abridged)	Local consequence
`forms/widgets.py`	`forms/fields.py`	Use `ensure_ascii=False` when preparing JSON values so Unicode is preserved during rendering.	Refine `prepare_value`
`db/models/fields/json.py`	`core/serializers/json.py`	Keep JSON encoding behavior consistent across model-field and serializer paths.	Refine serializer behavior
`utils/encoding.py`	`core/serializers/json.py`	Preserve Unicode rather than emitting ASCII escape sequences during serialization.	Reinforce serializer objective

In this trajectory, communication is mostly confirmatory: the recipient primitives already favor Unicode-preserving behavior, while incoming messages make the required cross-component consis-

tency explicit. The two `inspect` primitives illustrate that participation does not imply modification: both contribute information to other components while leaving their own artifacts unchanged.

Local modification. After communication, each `modify` primitive edits only its own artifact,

$$\mathcal{P}_i(a_i, x_i, \mathcal{C}_i) \to (a_i', m_i).$$

The four modifying primitives independently materialize compatible changes that introduce `ensure_ascii=False` across the relevant JSON-processing paths:

```
db/models/fields/json.py  get_prep_value:  json.dumps(value,
cls=self.encoder, ensure_ascii=False)
forms/fields.py  prepare_value:  json.dumps(value,
ensure_ascii=False, cls=self.encoder)
forms/utils.py  as_json (×2):  json.dumps(...,
ensure_ascii=False)
core/serializers/json.py  serializer configuration:
ensure_ascii=False
```

The final diff therefore modifies four of the six activated components. The other two remain unchanged after inspection and communication.

Execution and diagnosis decision. HERMES next executes the modified repository under the isolation protocol of Appendix F.1. The selected repository-visible regression suite runs 407 tests and returns the same successful exit status as before modification, providing evidence that the patch does not introduce an observed regression in this subsystem.

A task-specific reproducer is less informative in this trajectory. Three automatically constructed reproduction attempts terminate during Django configuration before reaching the reported display behavior and are therefore discarded. Consequently, the diagnosis module receives a successful regression run together with the candidate diff and original issue, but no reliable failing-to-passing task-specific execution signal.

The diagnosis module returns PASS after the first round, so no revision is triggered. This decision should be interpreted using the evidence available during solving: it indicates that the candidate patch is consistent with the issue and has no observed regression, rather than independently certifying the hidden benchmark behavior.

Held-out outcome. After HERMES terminates, the SWE-bench evaluator executes the held-out evaluation suite. All 35 tests pass, including the task-specific `test_json_display_for_field`, and the instance is marked `resolved=True`.

Interestingly, the accepted HERMES patch differs from the developer repair path. The developer patch modifies `django/contrib/admin/utils.py` and `django/forms/fields.py`, whereas HERMES does not activate `django/contrib/admin/utils.py`. Instead, it propagates the same Unicode-preserving constraint through lower-level JSONField and serialization paths. The resulting patch nevertheless satisfies the held-out behavioral evaluation.

This case also illustrates a limitation of the reference-overlap selection metrics in Section 4.4. Because those metrics use the developer-patch file set as ground truth, an alternative but behaviorally valid repair path can receive low reference overlap. Selection recall and precision should therefore be interpreted as agreement with the developer repair path rather than as exhaustive measures of all components capable of supporting a correct repair.

I LIMITATIONS

HERMES introduces additional inference cost and latency because it invokes multiple Dev-Primitives, executes the repository, and performs iterative diagnosis. Its gains are also smaller in already saturated settings, such as SWE-bench Verified with the strongest backbones. In addition, performance depends on correctly activating task-relevant components and on informative execution feedback for revision; missed dependencies or sparse runtime signals can therefore limit recovery. We use files as the default Dev-Primitive granularity and evaluate on four software engineering benchmarks, leaving finer-grained primitives and broader software engineering settings for future work. Finally, Qwen3-8B

is served locally with Ollama on NVIDIA A40 GPUs using the model's default sampling parameters (temperature 0.6, top-p 0.95, top-k 20) and Ollama's 32K context window, while all other models are accessed through their official APIs; results may therefore vary with deployment settings and model updates.

J CODE AND RESULT

We publish the comprehensive results of HERMERS on the web. For detailed information, please visit the following link: `https://anonymous.4open.science/r/D206/`.