

000 GISTIFY! CODEBASE-LEVEL UNDERSTANDING 001 002 VIA RUNTIME EXECUTION 003 004

005 **Anonymous authors**

006 Paper under double-blind review

007 008 ABSTRACT 009 010

011 As coding agents are increasingly deployed in large codebases, the need to au-
012 tomatically design challenging, codebase-level evaluation is central. We pro-
013 pose **GISTIFY**, a task where a coding LLM must create a single, minimal, self-
014 contained file that can reproduce a specific functionality of a codebase. The cod-
015 ing LLM is given full access to a codebase along with a specific entrypoint (e.g.,
016 a python command), and the generated file must replicate the output of the same
017 command ran under the full codebase, while containing only the essential compo-
018 nents necessary to execute the provided command. Success on **GISTIFY** requires
019 both structural understanding of the codebase, accurate modeling of its execution
020 flow as well as the ability to produce potentially large code patches. Our findings
021 show that current state-of-the-art models struggle to reliably solve **GISTIFY** tasks,
022 especially ones with long executions traces.

023 1 INTRODUCTION

024 Large language models (LLMs) are increasingly being used in code-related tasks, powering appli-
025 cations in debugging (Yuan et al., 2025) and agentic code generation (Yang et al., 2024; Liang et al.,
026 2025). Thus, the ability to handle isolated snippets and reasoning across entire codebases, includ-
027 ing complex file and module relationships, is becoming increasingly essential. Yet, the evaluation
028 toolkit for assessing such capabilities has lagged behind. Recent evidence shows that widely-adopted
029 repository-level benchmarks such as SWE-bench (Jimenez et al., 2024) and RepoBench (Liu et al.,
030 2023b) still do not *require full* reasoning over the whole execution and could be solved through
031 heuristic shortcuts or retrieval of localized patches (Aleithan et al., 2024; Liang et al., 2025). More-
032 over, because many of these datasets rely on GitHub issues or pull requests for construction, they
033 are not easily generalizable to arbitrary repositories. At the same time, coding agents are increas-
034 ingly deployed in large, real-world codebases, highlighting the need for *automatically constructed*,
035 broadly applicable, and more challenging repository-level evaluation.

036 To fill this gap, we introduce the **GISTIFY** task, which is deliberately inspired by a common prac-
037 tice of how developers navigate and understand unfamiliar repositories. Rather than reading files
038 in isolation, they start from a concrete execution point such as test command or entry script often
039 mentioned in READMEs. Then, they iteratively reason over the runtime behavior such as identi-
040 fying dependencies, following control paths to uncover the codebase’s structure and functionality.
041 **GISTIFY** formalizes this practice by requiring an (agentic) coding model to extract the *gist* of a given
042 command, i.e. to generate a single, self-contained, minimal, and executable *gistified* file that faith-
043 fully reproduces the runtime behavior of a given command as when using the original full codebase
044 (Figure 1). In addition to serving as a challenging coding task, such *gistified* repositories might give
045 human coders a better understanding of a specific functionality of a given codebase, or even a way
046 to *export* the single functionality of interest without inheriting heavy dependencies.

047 To perform well in **GISTIFY**, an agent should generate a single *gistified* file that satisfies *four* key
048 requirements: it should be **self-contained**, including all necessary components from the codebase
049 so that it can be executed independently; it should ensure **execution fidelity**, producing the same
050 outputs as the original codebase under the given command; it should satisfy **minimality**, retaining
051 only the essential code required for execution without redundant or extraneous lines; and it should
052 guarantee **faithful preservation**, avoiding hallucinated or fabricated code and relying solely on con-
053 tent from the original codebase. To assess model performance, we introduce evaluation metrics that

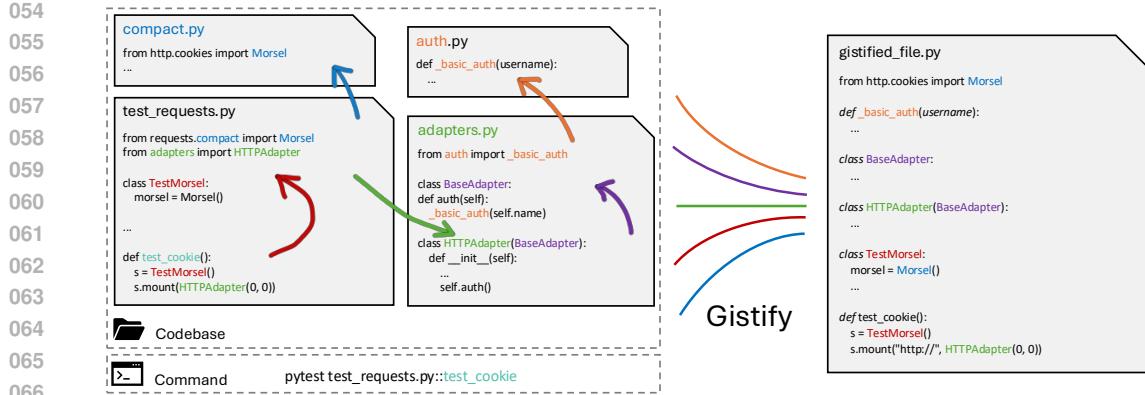


Figure 1: The GISTIFY task: given a codebase and a command of entrypoint, the goal is to generate a minimal, self-contained gistified code file that faithfully reproduces the original runtime behavior using code from the given codebase.

align with these requirements, providing a systematic way to measure codebase-level understanding. GISTIFY requires agents to follow the execution path through the codebase without bypassing modules, i.e., understanding how relevant objects are modified along the way, and identifying which classes or functions can be simplified or removed. Since even moderately sized codebases exceed the context window of current LLMs, success also requires effective search capabilities.

The advantages that GISTIFY brings are multiple: first, it provides direct insight into the ability of models to reason at the codebase level with an understanding of runtime execution, rather than on isolated code snippets. Second, it is lightweight and broadly applicable: it requires only the repository and a test suite (or any other collection of entrypoints with a well-defined expected output) and does not require issue logs or pull requests. This allows automatic construction of challenging tasks for arbitrary repositories, including private ones. Finally, gistified files themselves are valuable outputs: by compressing a specific feature of a large codebase into a minimal file, they can be applied to various downstream tasks, including automated debugging or error localization.

We conduct experiments across a variety of frameworks (mini-SWE-agent, SWE-agent, and Copilot) and models (GPT-5-mini, GPT-5, Claude-3.7-Sonnet, and Claude-Sonnet-4) and uncover several interesting findings. First, even widely used, high-performing frameworks and models struggle to create a successful gistified file, especially when execution traces are long and have high coverage on the repositories. Second, faithfully reproducing the test function in the generated file is a strong indicator of gistified performance, as it serves as the starting step for reasoning about execution traces. Third, enabling execution tools yields small but consistent performance gains, and additionally providing global code context and runtime information further boosts performance. Finally, agentic models benefit from dynamically deciding what to read and refine their reasoning through multi-step trajectories, outperforming static approaches.

2 RELATED WORKS

2.1 CODEBASE-LEVEL UNDERSTANDING BENCHMARK

Previous work has introduced a variety of benchmarks to evaluate LLMs on codebase-level code understanding¹. These generally fall into three categories: question answering, code synthesis, and mapping natural language specifications to the entire codebase. Several benchmarks introduce codebase-level question-answering (Strich et al., 2024; Li et al., 2024b; Sahu et al., 2024; Chen et al., 2025; Hu et al., 2024; Fu et al., 2025). In these settings, the model must correctly answer questions that require an understanding of the codebase. The questions are drawn from various sources, including real-world GitHub issues and queries resembling those asked of tools like Copilot. Another line of work evaluates whether models can synthesize code by leveraging information distributed

¹See Appendix A.1 for related works regarding “Methods for Codebase-level Understanding”

108 across multiple files in the codebase (Zhang et al., 2023; Liu et al., 2023b; Ding et al., 2023; Li
 109 et al., 2024a; Yu et al., 2024). These benchmarks include tasks such as retrieval-augmented comple-
 110 tion, cross-file refactoring, and more specialized settings such as sketch-based coding or codebase
 111 evolution. Moreover, there is a line of benchmark that maps natural language specifications to entire
 112 code repositories, leveraging hierarchical or multi-stage representations to capture inter-file relation-
 113 ships and maintain consistency across a codebase (Tang et al., 2023; Zan et al., 2024; Ni et al., 2025).
 114 Our work tackles a more complex setting, where models must reason over *full* execution traces and
 115 examine multiple files, making the task challenging, and even widely used agentic models struggle
 116 alongside static ones.

117 2.2 RUNTIME EXECUTION

118 Various works have introduced benchmarks to evaluate LLMs’ ability to reason over code execution
 119 at runtime (Gu et al., 2024; Chen et al., 2024; Xie et al., 2025; Beger & Dutta, 2025; Hu et al., 2025).
 120 These benchmarks typically test whether models can predict execution traces or intermediate states
 121 such as variable values, control flow, or data dependencies—given code and inputs, or alternatively,
 122 infer inputs from code and outputs. Some benchmarks further extend this paradigm by leveraging
 123 execution traces to construct new problems through program composition, thereby varying complex-
 124 ity in a principled way. Beyond evaluation, execution traces have also been incorporated into training
 125 pipelines to strengthen models’ runtime reasoning abilities (Liu et al., 2023a; Ding et al., 2024). By
 126 augmenting pre-training and fine-tuning with execution states, paths, and coverage signals, these
 127 methods help models capture program dynamics and generalize to execution-aware tasks. At infer-
 128 ence time, several frameworks leverage runtime feedback to iteratively guide models in debugging
 129 or completing partial programs, thereby improving performance on execution-driven tasks (Zhong
 130 et al., 2024; Xue et al., 2024). In this work, we extend prior approaches by going beyond reasoning
 131 over execution traces to also reformulate programs; the model not only tracks execution but also
 132 identifies how to compress and organize code into a concise, coherent file. We further show that
 133 this capability serves as a useful tool at inference time, helping models better structure and complete
 134 execution-driven tasks.

135 3 GISTIFY

136 3.1 TASK DEFINITION

137 As shown in Figure 1, when given a codebase and a command as input, the coding agent must
 138 generate a single gistified file that reproduces the runtime behavior of the original codebase under
 139 the given command. Specifically, the gistified file must satisfy the following requirements.

140 **Self-Contained:** All necessary components from the given codebase must be included so that the
 141 gistified file can be executed standalone, i.e. without relying on the codebase. The model must iden-
 142 tify all relevant modules and dependencies, demonstrating understanding of inter-file relationships.

143 **Execution Fidelity:** Executing the gistified file must replicate the original codebase’s runtime be-
 144 havior, ensuring the model captures the dynamic execution, not just static code patterns.

145 **Minimalism:** Only the code essential to reproducing the runtime behavior should be preserved,
 146 with unused functions and objects pruned. This requires fine-grained understanding of the code to
 147 identify which lines are actually executed and essential for the task.

148 **Grounded Preservation:** No hallucinated code may be introduced. All content must be derived
 149 directly from the original codebase. This ensures the task evaluates the model’s understanding of
 150 the codebase, rather than its ability to generate arbitrary code that happens to satisfy the command.

151 3.2 EVALUATION PROTOCOL

152 There are two inputs to a GISTIFY task: i) a docker image containing the target codebase, for
 153 consistent evaluation; ii) an entrypoint, such as a pytest command on one of the tests in the codebase.
 154 Test cases are existing entrypoints one can easily leverage, but broadly, any command that the user
 155 would want to use to run a functionality of the existing codebase is allowed.

162 All models are prompted to generate a gistified file for the entrypoint. We can programmatically
 163 verify whether the expected behavior is preserved when the ground-truth test is run within this setup.
 164 Here, we focus on comparing outputs of test commands. Once the model generates the gistified file,
 165 to ensure that execution for evaluation is based on the original test, we integrate the test code from
 166 the original codebase to the gistified file and execute it. This ensures that the model does not cheat
 167 by modifying the test.

168 3.3 METRICS

169 Once a gistified file is generated, we evaluate it using the given execution command. The evaluation
 170 considers three dimensions, aligned with the task requirements, to provide a comprehensive measure
 171 of a model’s ability to reason over an entire codebase and understand its execution behavior. See
 172 Appendix B.1 for more details.

173 **Execution Fidelity** is a binary metric where 1 means the gistified file runs successfully and produces
 174 the same output as the original codebase when executed under the given command; otherwise, it is
 175 0. Failures include cases where the file is not runnable or yields different outputs. The comparison
 176 checks for tests pass/fail consistency and stdout/stderr matching.

177 Formally, let c denote the given command, \mathcal{C} a given codebase, and \mathcal{G} a gistified file. Define
 178 $\text{runs}(c, \mathcal{C})$ as an indicator of whether c executes without crashing when running over \mathcal{C} , and $\text{out}(c, \mathcal{C})$
 179 returns the set of outputs and error traces from running c with \mathcal{C} . Then, execution fidelity is defined
 180 as

$$181 \mathbb{1}[\text{runs}(c, \mathcal{G}) \wedge \text{out}(c, \mathcal{G}) = \text{out}(c, \mathcal{C})], \quad (1)$$

182 where $\mathbb{1}[\cdot]$ is the indicator function.

183 **Line Execution Rate** measures minimality by calculating the fraction of lines in the gistified file that
 184 are actually executed under the given command. A 100% execution rate means all lines are essential,
 185 indicating a focused and concise file. This metric is only computed for files that run successfully,
 186 since the execution trace is required to determine which lines are run.

187 Formally, let $\mathcal{L}_{\text{exec}}(\mathcal{G})$ be a list of executable lines (i.e., no comments) in \mathcal{G} . Then, the Line Execution
 188 rate is defined as

$$189 \frac{1}{|\mathcal{L}_{\text{exec}}(\mathcal{G})|} \sum_{\ell \in \mathcal{L}_{\text{exec}}(\mathcal{G})} \mathbb{1}[\ell \text{ is executed}]. \quad (2)$$

190 **Line Existence Rate** measures the proportion of code in the gistified file that is directly preserved
 191 from the original codebase. Specifically, lines of code are grouped into blocks (classes, functions,
 192 or top-level units), and matches are computed block by block while respecting the code hierarchy.
 193 This helps avoiding false matches from common lines appearing in unrelated parts of the codebase.
 194 To ensure robustness, we normalize across common variations such as indentation, multi-line state-
 195 ments, and imports. A 100% existence rate indicates full fidelity to the original codebase without
 196 hallucination.

197 Formally, let $\mathcal{B}_{\mathcal{G}}$ and $\mathcal{B}_{\mathcal{C}}$ be the sets of blocks in the gistified file and the original codebase, respec-
 198 tively. For a block b , let $\mathcal{L}(b)$ represent its set of lines. Then, the existence rate is defined as

$$199 \frac{1}{\sum_{b \in \mathcal{B}_{\mathcal{G}}} |\mathcal{L}(b)|} \sum_{b \in \mathcal{B}_{\mathcal{G}}} \sum_{\ell \in \mathcal{L}(b)} \mathbb{1}\{\ell \in \mathcal{L}_{\mathcal{C}}(b)\}, \quad (3)$$

200 where $\mathbb{1}\{\ell \in \mathcal{L}_{\mathcal{C}}(b)\} = 0$, if no matching block exists in $\mathcal{B}_{\mathcal{C}}$.

210 4 EXPERIMENTS

211 4.1 SETTING

212 We conduct experiments using three widely adopted open-sourced frameworks. SWE-Agent (Yang
 213 et al., 2024) and GitHub Copilot (Microsoft, 2025) provide a rich scaffolding to LLM-based agents,
 214 enabling them autonomously perform software engineering tasks. This includes a set of tools for

216
 217 Table 1: Average Performance over three agentic frameworks with four models. We evaluated over
 218 25 tests over 5 repositories. Execution Fidelity is shown as *w/o exec*, and *w execution tools*. Line
 219 Existence and Execution are averaged across the two settings for clarity.

220	Framework	Model	Execution Fidelity (wo exec / w. exec)	Line Existence	Line Execution
221	mini-SWE-agent	GPT-5-mini	17.1 / 24.0	44.9	61.2
222		GPT-5	51.0 / 54.0	56.8	83.1
223		Claude-3.7	38.7 / 43.3	66.0	69.2
224		Claude-4	54.0 / 55.3	67.0	75.7
225	SWE-agent	GPT-5-mini	30.9 / 45.3	47.9	74.8
226		GPT-5	30.7 / 46.0	48.3	81.7
227		Claude-3.7	40.7 / 46.0	66.8	69.9
228		Claude-4	56.7 / 57.3	66.3	72.9
229	Copilot	GPT-5-mini	58.0 / 55.3	62.4	77.8
230		GPT-5	58.7 / 60.7	66.9	81.4
231		Claude-3.7	43.3 / 56.0	63.0	74.4
232		Claude-4	58.7 / 61.3	69.6	80.3

233
 234
 235 creating and editing code files, navigating repositories, and executing tests. These frameworks also
 236 offer the LLM controllable cache management, and LLMs follow the standard tool-calling format.
 237 We also experiment with Mini-SWE-Agent (Yang et al., 2024), a lightweight framework where
 238 LLMs only have access to a bash terminal to solve the task. Commands are parsed from the agent
 239 output and executed directly. As the task objective is for the model to use *reasoning* over the ex-
 240 ecution flow rather than ability of tool usage, for the agentic models, we exclude the execution tools
 241 (“python”, “pytest”) in the default setting where execution is disabled. For all three frameworks,
 242 unless specified otherwise, hyperparameters and configurations (e.g. system prompts, cache man-
 243 agement, tools) are kept to the default values. Please see Appendix

244 Our evaluation spans four leading LLM variants: GPT-5 (OpenAI, 2025a), GPT-5-mini (OpenAI,
 245 2025b), Claude-3.7-Sonnet (Anthropic, 2025a), and Claude-Sonnet-4 (Anthropic, 2025b), offering
 246 different cost / performance tradeoffs. For ease or reading, we will refer to the last two models as
 247 Claude-3.7 and Claude-4. We use a 128K token limit for all models. All experiments ran are capped
 248 at 50 steps, after which whatever is generated at this moment in the gistified file is submitted for
 249 evaluation.

250 On the data side, we experiment over with widely used GitHub repositories which are present in
 251 SWE-Bench (requests, pylint, flask, scikit-learn, seaborn). We also explore an
 252 additional repository, debug-gym (Yuan et al., 2025)². This library is relatively new and impor-
 253 tantly does not overlap with SWE-Bench. We extract and filter test sets for each repository. Namely,
 254 we remove tests whose execution is dependent on the test’s file location. For the main experiment,
 255 we evaluate over 25 tests for each of the 5 repositories. More details regarding the evaluation setup
 256 and prompt can be found in the Appendix C.

257 258 4.2 RESULTS

259 We begin by giving an overview of the main results presented in Table 1. We report results for our
 260 main evaluation protocol, where the model does not have access to execution tools (e.g. “python”
 261 and “pytest” commands), as well as the alternative. Examples of gisted files are in Appendix D.1.

262
 263 **Claude-4 shows the most robust performance.** Across all frameworks and configurations. Claude-
 264 4 consistently provides the best performance, reaching a 54-60% average solve rate. Moreover, the
 265 model shows the highest values of Line Existence, meaning that it was the most successful model
 266 at faithfully extracting code from the original codebase. We note however that GPT-5 produces the
 267 most concise outputs, with Line Execution rate markedly higher than other models.

268
 269 ²We provide link to all the GitHub repositories used in this work in Table 4.

270 Table 2: Average error rates (%) of different failure reasons when running SWE-agent across models.
 271 Error cases are categorized into four groups. The numbers in parentheses indicate the number of
 272 errors for each category.

Models	Import Error	File Creation Failure	Missing Test Function	Pytest Runtime Error
GPT-5-mini	2.1 (2)	11.3 (11)	76.3 (72)	10.3 (10)
GPT-5	5.2 (4)	10.4 (8)	77.9 (60)	6.5 (5)
Claude-Sonnet-3.7	20.0 (10)	20.0 (10)	2.0 (1)	58.0 (29)
Claude-Sonnet-4	32.5 (13)	10.0 (4)	7.5 (3)	50.0 (20)

273
 274
 275
 276
 277
 278
 279
 280
 281 **Frontier models (GPT-5 / Claude-4) are strong bash users.** When looking at performance on
 282 mini-swe-agent, where the models only have access to a bash terminal to solve the task, both models
 283 perform relatively well, solving over half of the tasks. Importantly, this is not the case for smaller
 284 and previous-generation models.
 285

286 **Execution tools are not a silver bullet.** Overall, when comparing performance with and without
 287 execution in Table 1, we note that in most cases we observe only a small performance gain. We
 288 expected that current coding LLMs could better leverage execution tools: indeed, using tools specif-
 289 ically for runtime execution analysis, such as a debugger, could significantly help solving a gistify
 290 task. However, we are not seeing this behavior emerge, even from frontier models. We observed a
 291 sharp decrease in performance for the GPT-5 model when evaluated on SWE-Agent without execu-
 292 tion tools. We performed a visual inspection and noticed formatting issues when rewriting the input
 293 test function. A detailed discussion can be found in Appendix D.2.
 294

295 **Small(er) models perform well with scaffolding.** We note that GPT-5-mini’s performance varies
 296 significantly across different evaluation settings, from 17% in a bash-only setup to 58% when pro-
 297 vided with a large inventory of tools from the Copilot framework (see Appendix D.3 for a full list).
 298 We note that this performance increase is also reflected in the quality of the generated gist, where
 299 we see a notable increase in line existence and line execution.
 300

301 4.3 ERROR ANALYSIS OVER EXECUTION FAILURE

302 We proceed with an analysis of the underlying failure causes, in order to understand which aspect
 303 of the GISTIFY task different models struggle with. Table 2 shows that each model tends to fail for
 304 different reasons. See Appendix D.4 for detailed examples of each error case.
 305

306 **Import Error** occurs when the model incorrectly imports the original codebase (e.g., `import`
 307 `requests`) instead of inlining the required modules into the gistified file. We note that this error
 308 occurs even as coding LLMs are explicitly prompted not to import the specific packages in question.
 309 Perhaps surprisingly, the best performing model, Claude-4, commits this seemingly innocuous error
 310 the most out of all four models.
 311

312 **File Creation Failure** errors arise when the model fails to generate the gistified file. This can happen
 313 in two ways: the model exceeds the maximum step limit, or the model terminates the task without
 314 any file being generated.
 315

316 **Missing Test Function** errors occur when the generated gistified file does not contain the function
 317 implementation for the test specified in the given command, or implements the test in a different
 318 structure. This can happen when the model strips out the content of the test and executes it outside
 319 of the pytest wrapper, under e.g. `if __name__ == __main__:`. Claude models tend to avoid this
 320 mistake, while this is the main source of error for GPT-5 models, specifically under the SWE-agent
 321 framework. Importantly, we observe that this error does not happen at random, but rather alongside
 322 other execution errors; we attempted to add the missing test function, and it in most cases the test
 323 fails to run, i.e. it results in a runtime error. This aligns with the analysis in the next section,
 324 showing a strong correlation between the task’s success and the fidelity between the original and the
 325 generated tests.
 326

324
 325 Table 3: Analysis of the effect of different strategies and tools (global information, execution)
 326 on the GISTIFY task. We evaluate SWE-Agent with Claude 4 using 50 test instances from the pylint
 327 codebase. Max Steps Reached (%) indicates the percentage of runs that terminated because the
 328 maximum step limit was reached.

Ablation	Type	Execution Fidelity	Line Existence	Line Execution	Max Steps Reached (%)
Base GISTIFY		42.0	65.0	58.3	14.6
Prompted Strategies	Tracing	48.0	75.4	62.8	0.0
	Reading	50.0	77.6	62.6	3.9
Global Info (Tool)	RepoGraph	52.0	76.1	60.1	6.0
	Tracing	56.0	75.1	65.1	0.0
Execution (Tool)	Bash	52.0	73.1	64.2	16.0
	Edit And Execute	56.0	74.3	64.2	10.0

339
 340 **Pytest Runtime Error** occurs when the execution of the generated file fails, either due to a runtime
 341 error or because the gistified output does not match the output from the original codebase. The
 342 results indicate this is the most common cause of error for the best performing model, Claude-4.

344 4.4 IMPORTANCE OF FAITHFULLY PRESERVING THE TEST FUNCTION

345
 346 We observe that models frequently modify the test function, despite being provided with explicit
 347 instructions to copy without modification, except for unavoidable adjustments (e.g., removing im-
 348 ports). Again, to ensure consistent evaluation, we replace the test function in the gistified file with
 349 the original version before evaluation.

350 To measure such modifications, we define the *Test F_1 Score* as the line-level overlap between the
 351 *test code* of the original file and the gistified version. High Test F_1 Score indicates that the model
 352 has successfully identified and copied the correct test function to the gistified file. We observe
 353 a strong correlation between Test F_1 Score and execution fidelity (correlation=0.76, p=0.01); test
 354 instances with higher F_1 scores are substantially more likely to produce a successful gistified file.
 355 We hypothesize that this arises because in the GISTIFY task, models often reason backwards from
 356 the test file, thereby if the model fails from identifying or copying the test function, the subsequent
 357 reasoning process is highly likely to fail.

358 To better understand the impact of the first step—searching, viewing, and copying the test func-
 359 tion—we conduct an ablation study where we remove potential failure at this stage. Specifically, we
 360 explicitly provide the correct test function body and signature in the prompt, so the model no longer
 361 needs to locate or copy it. This isolates the effect of errors in identifying the test function. In this set-
 362 ting, we observe that Test F_1 Score improves highly from the base GISTIFY 68.4 to 85.3, along with
 363 execution fidelity (from 42.0% to 60.0%). This suggests that accurately handling the test function is
 364 a critical first step to do the GISTIFY task successfully. Detailed results are in Appendix D.5.

365 5 ANALYSIS

366
 367 In this section, we analyze how different strategies and tools affect performance on the GISTIFY
 368 task, identify factors that contribute to its difficulty, and experiment with the use of a static coding
 369 LLM to gain a deeper understanding of the task. For all experiments, we evaluate 50 test instances
 370 drawn from the pylint codebase, a setting where the model generally exhibited modest performance.
 371 We use SWE-Agent paired with Claude-Sonnet-4.

373 5.1 EFFECT OF VARIOUS STRATEGIES AND TOOLS

374
 375 In this section, we analyze how different strategies and sources of information affect model perfor-
 376 mance. We begin with the simplest approach, modifying the prompt to guide the model (*Prompt-
 377 Based Guidance*), and then move to more explicit approaches that rely on additional tools: providing
 378 global context (*Global Information via Tools*) or feedback from code execution (*Execution-Based*

378 *Tools*). Detailed descriptions of prompts and tools, along with examples, are provided in the Appendix E.1.
 379
 380

381 **Prompt-Based Guidance** We first begin with the simplest approach: modifying the prompt to
 382 provide explicit task guidance. We experiment over two settings. In the former, we prompt the
 383 model to perform step-by-step reasoning, by first predicting the execution traces and then going
 384 over them, adding relevant code snippets along the way (*tracing*). In the latter, a similar approach is
 385 used, with explicit instructions on *how* to recursively determine the execution traces: starting from
 386 the test, identify the relevant components and read the files where they are defined, and repeat until
 387 the end (*reading*). As shown in Table 3, we observe that adding such strategies tends to enhance
 388 overall metrics, giving both better execution fidelity and more faithful code extractions, as measured
 389 by line existence.
 390

391 **Global Information via Tools** Building on the above observation, we next assess the effect of
 392 explicitly providing global context through external tools, rather than predicting it. We examine
 393 two tools: (1) *RepoGraph* (Ouyang et al., 2024), which constructs a graph of the codebase where
 394 each node represents a line of code and edges capture connections between lines, enabling graph-
 395 based search over the entire codebase; and (2) a *Tracing* tool that exposes gold execution traces
 396 obtained from running the given test command. Results in Table 3 show that both tools improve
 397 performance, with the *Tracing* tool yielding the largest gains. This finding suggests that access to
 398 the global context, especially the gold tracing information, substantially strengthens the model’s
 399 ability to perform runtime reasoning, as it can easily identify which file to look at.
 400

401 **Execution-Based Tools** In Section 4.2, we saw that enabling execution tools resulted in small but
 402 consistent gains overall. In this section, we examine whether having unrestricted access to a bash
 403 terminal is really necessary to observe these gains, or whether simply having access to execution logs
 404 of the generated file is enough. For this experiment we compare *Bash* access with a simple method
 405 that executes and prints the output of the gistified file whenever it is edited (*Edit And Execute*). No
 406 other execution tools are available to the agent, including runtime information about the ground truth
 407 test. The results are surprising: having access to *fewer* tools actually increases performance. Indeed,
 408 we note that when give access to a full set of bash commands, the coding LLM tends to explore more
 409 tools, increasing the overall trajectory length, and potentially reaching the maximum step limit.
 410

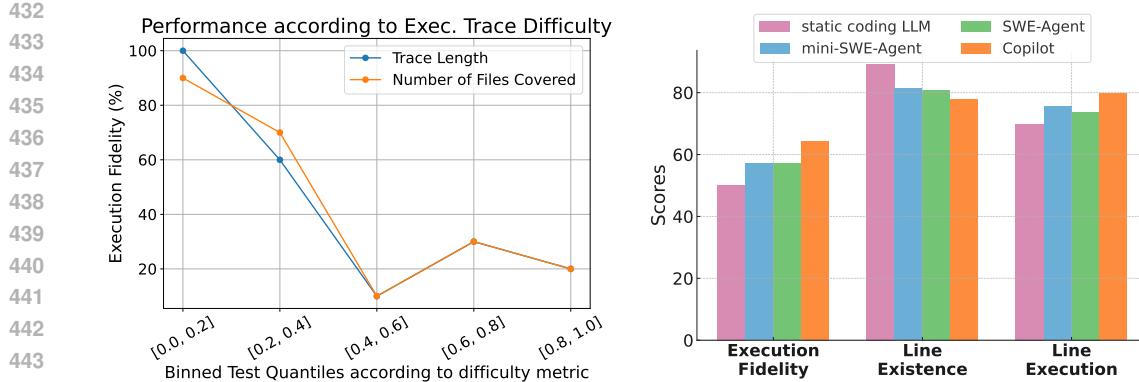
411 5.2 TESTS WITH HIGH COVERAGE ARE HARDER TO GISTIFY

412 In this section, we investigate what properties makes a given test hard to GISTIFY. We hypothesize
 413 that tests generating a longer and more complex execution trace would entail a harder task for the
 414 coding LLM. To this end, we investigate how two axes to measure a runtime execution’s difficulty
 415 affect performance: the **length of trace**, as measured by the number of function calls executed, and
 416 the **number of unique files** touched by the tracing procedure. While these metrics correlate with
 417 one another, they will differ when, for example, a function is looped over many times or when the
 418 location of the relevant functions is in a single file versus across multiple files.
 419

420 For this experiment, we use again the same configuration as prior analysis, namely Claude-4 with 50
 421 tests sampled from the pylint codebase. In Figure 2a, we see a clear correlation between the difficulty
 422 of a given GISTIFY task, and how complex the execution traces are, according to both metrics
 423 considered. We leverage this insight to create a **GISTIFY-hard** subset, where we select the 30 most
 424 difficult examples according to each. We end up with 57 unique datapoints (30 from pylint, 28 from
 425 sklearn, 6 from seaborn). On this subset, performance drops to **21%**, as compared to **43%**, the
 426 baseline weighted performance average following the same distribution over repositories. Overall,
 427 this selection criteria offers a promising direction for designing challenging evaluation scenarios
 428 with GISTIFY.
 429

430 5.3 STATIC CODING LLM

431 In this section, we experiment over how models perform in a static setup, where they have no access
 432 to tools and cannot iterate on the generated solution. As such static coding LLMs do not have
 433 tools, they cannot search or view files dynamically. Thereby, to measure a possible upper bound for
 434 non-agentic approaches, we provide as input all files that were accessed during the original program
 435



(a) Difficulty of the Gistify task is measured as a function of the execution trace difficulty of the underlying test.

(b) Performance of a static coding LLM and various agentic coding LLMs (mini-SWE-Agent, SWE-Agent, Copilot).

execution (gold files). Also, as they cannot iterate over multiple steps, they have to output everything at once and are therefore restricted by the context window of the LLM. Since solving the GISTIFY task involves touching multiple files, we observe in many cases that the inputs exceed the model’s maximum sequence length. Thus, we sample a subset of test examples where the combined content fits within the 128K token limit of the LLM. As shown in Figure 2b, agentic models outperform static ones even when the latter receive all relevant files. This suggests that selecting files dynamically over multiple iterations is more effective than providing everything at once, which can overwhelm the model³. However, interestingly, the static coding LLM setup achieves the highest Line Existence score. This is likely because the model can copy lines directly from input, yet it performs worse on Line Execution and Execution Fidelity, suggesting that models do not have a good understanding of the codebase, often copying lines that are incomplete or incorrect.

6 DISCUSSION AND CONCLUSION

In this paper, we introduced the GISTIFY task in which a coding LLM extracts a specific functionality of a codebase into a single, self-contained file. Beyond serving as a standalone evaluation task that is easily applicable to arbitrary repositories with a test suite, the gistified file itself also opens several promising directions for research and practical applications. Large codebases often overwhelm automated agents due to their complex dependencies, and they especially struggle when tasked with fixing bugs that span multiple files (Ganhotra, 2025). In such scenarios, a gistified file would greatly reduce this challenge, and enable a more efficient reasoning about the codebase without navigating through unrelated code. In other words, this file could be leveraged in other downstream tasks such as code refactoring or debugging, or even as a way to extract and share a minimal implementation of a specific codebase functionality. Lastly, a current limitation of the results presented is the reliance on an existing test suite for a given repository. We believe that the GISTIFY task can be extended to arbitrary entrypoints, although issues stemming from non-deterministic execution will need to be carefully addressed. We defer a proper exploration of this to future work.

In summary, with coding LLMs increasingly being deployed in real-world software development, the need for automatically constructing evaluation setups that require codebase-level understanding of arbitrary repositories is growing. Through extensive experiments across a range of models and frameworks, we found that state-of-the-art LLMs still face challenges on the GISTIFY task, especially when faced with long, complex execution traces. Our analysis shows that incorporating global code context or execution-aware tools improves performance, and agentic coding LLM tend to handle the task more effectively by reasoning about which files to inspect using various tools. Beyond serving as a benchmark, the gistified files themselves are valuable artifacts. They distill the essential functionality of complex systems into a compact, executable form, making them easier to inspect and understand. Such files could support a range of practical applications, including debugging, refactoring, and code review, which we leave this for future work.

³See Appendix E.3 for detailed statistics on the usage of various tools.

486 REFERENCES
487

488 aider. Ai pair programming in your terminal. 2025. URL <https://github.com/Aider-AI/aider?tab=readme-ov-file>.

489

490 Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song
491 Wang. Swe-bench+: Enhanced coding benchmark for llms. *arXiv preprint arXiv:2410.06992*,
492 2024.

493

494 Anthropic. Claude sonnet 3.7. <https://www.anthropic.com/news/claude-3-7-sonnet>, 2025a. Hybrid reasoning model; accessed: 2025-09-25.

495

496 Anthropic. Claude sonnet 4. <https://www.anthropic.com/clause/sonnet>, 2025b.
497 Improved version over Sonnet 3.7; accessed: 2025-09-25.

498

499 Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy,
500 Sriram Rajamani, Balasubramanyan Ashok, and Shashank Shet. Codeplan: Repository-level
501 coding using llms and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE):
502 675–698, 2024.

503

504 Claas Beger and Saikat Dutta. Coconut: Structural code understanding does not fall out of a tree. In
505 *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pp.
506 128–136. IEEE, 2025.

507

508 Jialiang Chen, Kaifa Zhao, Jie Liu, Chao Peng, Jierui Liu, Hang Zhu, Pengfei Gao, Ping Yang, and
509 Shiguang Deng. Coreqa: uncovering potentials of language models in code repository question
510 answering. *arXiv preprint arXiv:2501.03447*, 2025.

511

512 Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. Reasoning runtime behavior
513 of a program with llm: How far are we? *arXiv preprint arXiv:2403.16437*, 2024.

514

515 cursor. cursor. 2025. URL <https://cursor.com/>.

516

517 Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna
518 Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. Crosscodeeval: A diverse
519 and multilingual benchmark for cross-file code completion. *Advances in Neural Information
520 Processing Systems*, 36:46701–46723, 2023.

521

522 Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. Traced:
523 Execution-aware pre-training for source code. In *Proceedings of the 46th IEEE/ACM Interna-
524 tional Conference on Software Engineering*, pp. 1–12, 2024.

525

526 Lingyue Fu, Hao Guan, Bolun Zhang, Haowei Yuan, Yaoming Zhu, Jun Xu, Zongyu Wang, Lin Qiu,
527 Xunliang Cai, Xuezhi Cao, et al. Corecodebench: A configurable multi-scenario repository-level
528 benchmark. *arXiv preprint arXiv:2507.05281*, 2025.

529

530 Jatin Ganhatra. Do swe-agents solve multi-file issues like humans? a deep dive into swe-bench ver-
531 ified, January 2025. URL <https://jatinganhatra.dev/blog/swe-agents/2025/01/05/swe-bench-mutliple-files/>. Blog post.

532

533 Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian Moghaddam.
534 Refactorbench: Evaluating stateful reasoning in language agents through code. *arXiv preprint
535 arXiv:2503.07832*, 2025.

536

537 Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and
538 Sida Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. In
539 Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scar-
540 lett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine
541 Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 16568–16621. PMLR,
542 21–27 Jul 2024.

543

544 Ruida Hu, Chao Peng, Jingyi Ren, Bo Jiang, Xiangxin Meng, Qinyun Wu, Pengfei Gao, Xinchen
545 Wang, and Cuiyun Gao. Coderepoqa: A large-scale benchmark for software engineering question
546 answering. *arXiv preprint arXiv:2412.14764*, 2024.

540 Wenhao Hu, Jinhao Duan, Chunchen Wei, Li Zhang, Yue Zhang, and Kaidi Xu. Dynacode: A
 541 dynamic complexity-aware code benchmark for evaluating large language models in code gener-
 542 ation. *arXiv preprint arXiv:2503.10452*, 2025.

543

544 Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik
 545 Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint*
 546 *arXiv:2310.06770*, 2023.

547

548 Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R
 549 Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth*
 550 *International Conference on Learning Representations*, 2024.

551

552 Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. Evocodebench: An evolving code gen-
 553 eration benchmark aligned with real-world code repositories. *arXiv preprint arXiv:2404.00599*,
 554 2024a.

555

556 Linyi Li, Shijie Geng, Zhenwen Li, Yibo He, Hao Yu, Ziyue Hua, Guanghan Ning, Siwei Wang,
 557 Tao Xie, and Hongxia Yang. Infibench: Evaluating the question-answering capabilities of code
 558 large language models. *Advances in Neural Information Processing Systems*, 37:128668–128698,
 559 2024b.

560

561 Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. The swe-bench illusion:
 562 When state-of-the-art llms remember instead of reason. *arXiv preprint arXiv:2506.12286*, 2025.

563

564 Chenxiao Liu, Shuai Lu, Weizhu Chen, Dixin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sun-
 565 daresan, and Nan Duan. Code execution with pre-trained language models. In Anna Rogers,
 566 Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Findings of the Association for Computational*
 567 *Linguistics: ACL 2023*, pp. 4984–4999, Toronto, Canada, July 2023a. Association for Compu-
 568 *tational Linguistics*. doi: 10.18653/v1/2023.findings-acl.308. URL <https://aclanthology.org/2023.findings-acl.308/>.

569

570 Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code
 571 auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023b.

572

573 Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Shieh, and
 574 Wenmeng Zhou. Codexgraph: Bridging large language models and code repositories via code
 575 graph databases. *arXiv preprint arXiv:2408.03910*, 2024.

576

577 Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong,
 578 Yankai Lin, Yingli Zhang, et al. Repoagent: An llm-powered open-source framework for
 579 repository-level code documentation generation. *arXiv preprint arXiv:2402.16667*, 2024.

580

581 Microsoft. Github copilot in vs code. 2025. URL <https://code.visualstudio.com/docs/copilot/overview>.

582

583 Ziyi Ni, Huacan Wang, Shuo Zhang, Shuo Lu, Ziyang He, Wang You, Zhenheng Tang, Yuntao Du,
 584 Bill Sun, Hongzhang Liu, et al. Gittaskbench: A benchmark for code agents solving real-world
 585 tasks through code repository leveraging. *arXiv preprint arXiv:2508.18993*, 2025.

586

587 OpenAI. Gpt-5 technical overview. <https://platform.openai.com/docs>, 2025a. Ac-
 588 cessed: 2025-09-25.

589

590 OpenAI. Gpt-5 mini. <https://platform.openai.com/docs/models/gpt-5-mini>,
 591 2025b. Compact variant of GPT-5; accessed: 2025-09-25.

592

593 Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hong-
 594 ming Zhang, and Dong Yu. Repograph: Enhancing ai software engineering with repository-level
 595 code graph. *arXiv preprint arXiv:2410.14684*, 2024.

596

597 Surya Prakash Sahu, Madhurima Mandal, Shikhar Bharadwaj, Aditya Kanade, Petros Maniatis, and
 598 Shirish Shevade. Codequeries: A dataset of semantic queries over code. In *Proceedings of the*
 599 *17th Innovations in Software Engineering Conference*, pp. 1–11, 2024.

594 Disha Shrivastava, Denis Kocetkov, Harm De Vries, Dzmitry Bahdanau, and Torsten Scholak. Re-
 595 posfusion: Training code models to understand your repository. *arXiv preprint arXiv:2306.10998*,
 596 2023.

597 Jan Strich, Florian Schneider, Irina Nikishina, and Chris Biemann. On improving repository-level
 598 code QA for large language models. In Xiyan Fu and Eve Fleisig (eds.), *Proceedings of the 62nd*
 599 *Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research*
 600 *Workshop)*, pp. 209–244, Bangkok, Thailand, August 2024. Association for Computational Lin-
 601 *guistics*. ISBN 979-8-89176-097-4. doi: 10.18653/v1/2024.acl-srw.28.

602

603 Xiangru Tang, Yuliang Liu, Zefan Cai, Yanjun Shao, Junjie Lu, Yichi Zhang, Zexuan Deng, Helan
 604 Hu, Kaikai An, Ruijun Huang, et al. MI-bench: Evaluating large language models and agents for
 605 machine learning tasks on repository-level code. *arXiv preprint arXiv:2311.09835*, 2023.

606

607 Huacan Wang, Ziyi Ni, Shuo Zhang, Shuo Lu, Sen Hu, Ziyang He, Chen Hu, Jiaye Lin, Yifu Guo,
 608 Yuntao Du, et al. Repomaster: Autonomous exploration and understanding of github repositories
 609 for complex task solving. *arXiv preprint arXiv:2505.21577*, 2025.

610

611 Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan,
 612 Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software
 613 developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

614

615 Danning Xie, Mingwei Zheng, Xuwei Liu, Jiannan Wang, Chengpeng Wang, Lin Tan, and Xiangyu
 616 Zhang. Core: Benchmarking llms code reasoning capabilities through static analysis tasks. *arXiv*
 617 *preprint arXiv:2507.05269*, 2025.

618

619 Zhipeng Xue, Zhipeng Gao, Shaohua Wang, Xing Hu, Xin Xia, and Shaping Li. Selfpico: Self-
 620 guided partial code execution with llms. In *Proceedings of the 33rd ACM SIGSOFT International*
 621 *Symposium on Software Testing and Analysis*, pp. 1389–1401, 2024.

622

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

626

627 Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianx-
 628 iang Wang, and Tao Xie. Codereval: A benchmark of pragmatic code generation with generative
 629 pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software*
 630 *Engineering*, pp. 1–12, 2024.

631

632 Xingdi Yuan, Morgane M Moss, Charbel El Feghali, Chinmay Singh, Darya Moldavskaya, Drew
 633 MacPhee, Lucas Caccia, Matheus Pereira, Minseon Kim, Alessandro Sordoni, et al. debug-gym:
 634 A text-based environment for interactive debugging. *arXiv preprint arXiv:2503.21557*, 2025.

635

636 Daoguang Zan, Ailun Yu, Wei Liu, Dong Chen, Bo Shen, Wei Li, Yafen Yao, Yongshun Gong,
 637 Xiaolin Chen, Bei Guan, et al. Codes: Natural language to code repository via multi-layer sketch.
 638 *arXiv preprint arXiv:2403.16443*, 2024.

639

640 Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang
 641 Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval
 642 and generation. *arXiv preprint arXiv:2303.12570*, 2023.

643

644 Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger
 645 via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*, 2024.

646

647

648 A RELATED WORKS
649650 A.1 METHODS FOR CODEBASE-LEVEL UNDERSTANDING
651652 Recent work on autonomous agents for codebase-level code understanding has focused on improving
653 code navigation, reasoning, and generation through structured representations and planning.
654 Approaches leverage structural information of code for function-call graphs, module-dependency
655 graphs, and hierarchical code structures to provide models with core components of repositories
656 (Wang et al., 2025; Liu et al., 2024). Another line of work integrate multi-step reasoning and
657 state update policies to enable more effective planning over complex tasks (Bairi et al., 2024; Gau-
658 tam et al., 2025). Additional methods combine various agents with multiple tools to streamline
659 codebase-level exploration and task solving (Luo et al., 2024; Zhang et al., 2023; Shrivastava et al.,
660 2023; Wang et al., 2024; Yang et al., 2024; Tang et al., 2023; aider, 2025; Microsoft, 2025; cursor,
661 2025).662
663 B GISTIFY
664665 B.1 METRICS
666667 **Execution Fidelity** Execution fidelity measures whether the generated gistified file reproduces the
668 same functional behavior as the original codebase under the given command. This includes produc-
669 ing the same number of test passes or failures, as well as consistent outputs and error handling. If
670 the file’s behavior matches the original codebase, it is assigned 100%; otherwise it receives 0%.
671672 **Line Execution Rate** The line execution rate measures the proportion of lines in the gistified file
673 that are actually executed when running it under the given command. We first analyze the gistified
674 file to identify which lines are executable (e.g., imports, function or class definitions) versus not-
675 executable (e.g., comments). Using a tracing function, we then determine which of the executable
676 lines are touched during execution. The line execution rate is computed as the fraction of executable
677 lines that are executed. A rate of 100% indicates that the gistified file is concise and contains
678 primarily necessary lines that are executed, while 0% indicates that none of the executable lines were
679 touched. When calculating line execution rate, we exclude the tests where the self-containment is
680 0% as the goal of line execution rate is to evaluate the model’s ability to construct concise, executable
681 file, not to penalize failures in generating runnable code.682 We classify each line of code into three categories: executable, potentially executable, and non-
683 executable. Executable lines include imports and functional code that can be directly run. Potentially
684 executable lines are those that may or may not be executed during a run, such as the except block of a
685 try-except statement or placeholders for classes and function definitions. Non-executable lines, such
686 as comments, are those that have no effect on execution. To calculate the line execution rate, we first
687 classify each line in the gistified file and then consider only the *executable* lines. Non-executable
688 lines are ignored since their presence or absence does not affect execution outcomes, and potentially
689 executable lines are excluded because they are often ambiguous (e.g., placeholders) and cannot be
690 reliably judged as necessary or removable.691 **Line Existence Rate** The line existence rate measures the proportion of lines in the gistified file
692 that are directly preserved from the original codebase. We first parse both the gistified file and the
693 original codebase into blocks, where each block corresponds to a class or function. Within classes,
694 functions are nested under their parent class, forming a hierarchy. Lines outside of any block (e.g.,
695 top-level statements) are treated as standalone units.696 For each block in the gistified file, we locate the corresponding block in the original codebase using
697 its name and hierarchical position. If a matching block exists, we compare the two line by line to
698 determine which lines are preserved; whether the lines in the gistified block appear in the corre-
699 sponding original block. If no match is found, all lines in that block are treated as non-existent.
700 For lines outside any block, existence is determined by direct comparison with top-level lines in the
701 original codebase.

```

702 Preprocess
703
704 Require: Gistified file  $G$ , original repository  $R$ , module name  $M$ , execution information  $E$ 
705 for lines in  $G$ 
706
707 1: Preprocessing: Line Classification
708 2: for all lines  $\ell$  in  $G$  do
709 3:   assign  $\text{type}[\ell] \in \{\text{comment}, \text{control\_flow}, \text{definition}, \text{executable}, \text{import}, \text{blank}\}$ 
710 4: end for
711
712
713 Figure 3: Details of Line Execution and Line Existence (Part 1): Preprocessing
714
715 Details of Line Execution
716
717 Require: Preprocessed gistified file  $G$ , original repository  $R$ , module name  $M$ , execution
718 information  $E$  for lines in  $G$ 
719 Ensure: Line execution rate  $r_{\text{exec}}$ 
720
721 1: Line Execution Rate
722 2:  $S \leftarrow \{\ell \in G \mid \text{type}[\ell] \in \{\text{executable}, \text{import}\}\}$ 
723 3:  $S_{\text{exec}} \leftarrow \{\ell \in S \mid \ell \text{ is marked as executed in } E\}$ 
724 4:  $r_{\text{exec}} \leftarrow \frac{|S_{\text{exec}}|}{|S|}$ 
725
726
727 Figure 4: Details of Line Execution and Line Existence (Part 2): Line Execution
728
729
730 An existence rate of 100% indicates perfect preservation of the original code without hallucinated
731 content.
732
733 Normalization for Line-wise Code Matching Figure 3, 4, and 5 show the detailed procedure
734 to compute the line existence rate and line execution rate for a gistified file. The process begins
735 by classifying every line of the gistified file into one of the categories: comment, control flow,
736 definition, executable, import, or blank (Figure 3). This classification forms the basis for later
737 filtering and comparison steps.
738
739 When calculating the line execution rate, we consider only lines classified as executable or import,
740 because these are the lines whose execution status can be directly observed in the execution infor-
741 mation (Figure 4). We deliberately exclude control-flow and definition lines from this rate: although
742 they are crucial for the gistified file to behave like the original repository, whether such lines “exe-
743 cutes” or not is often input- or parameter-dependent and therefore not reliably captured by a simple
744 per-line execution count. The validity of control-flow and definition structures will be instead be
745 indirectly assessed through our execution fidelity metric, which measures whether the gistified file
746 and the original repository exhibit consistent overall execution behavior. In this way, the line ex-
747 ecution rate focuses on directly measurable execution coverage, while execution fidelity provides a
748 higher-level signal about behavioral correctness.
749
750 When calculating the line existence rate, both the gistified file and the original repository are parsed
751 into nested structural blocks such as functions or classes. Any lines that are not contained within a
752 structural block are treated separately as top-level lines (line 1-4 in Figure 5). For all cases, when
753 calculating the line existence rate, we do not consider comments or blanks. The algorithm then
754 examines each structural block in the gistified file and attempts to find the matching block from the
755 original repository. If no matching block is found, all lines in the block are considered missing (line
14

```

756 Details of Line Existence

757

758 **Require:** Preprocessed gistified file G , original repository R , module name M , execution information E for lines in G

759 **Ensure:** Line existence rate r_{exist}

760 1: **Structural Block Parsing**

761 2: parse G and R into nested structural blocks (e.g., functions and classes)

762 3: $\mathcal{B}_G \leftarrow$ set of blocks in G

763 4: $L_{\text{top}} \leftarrow$ lines in G not contained in any structural block

764 5: **Initialization**

765 6: **for** all lines ℓ in G **do**

766 7: $\text{exists}[\ell] \leftarrow$ undefined

767 8: **end for**

768 9: **Block-Level Existence Analysis**

769 10: **for** all blocks $b \in \mathcal{B}_G$ **do**

770 11: $\text{orig} \leftarrow$ best-matching block for b in R

771 12: **if** orig does not exist **then**

772 13: **for** all lines $\ell \in b$ **do**

773 14: **if** $\text{type}[\ell] \notin \{\text{comment}, \text{blank}\}$ **then**

774 15: $\text{exists}[\ell] \leftarrow$ false

775 16: **end if**

776 17: **end for**

777 18: **continue** to next block

778 19: **end if**

779 20: **for** all lines $\ell \in b$ **do**

780 21: **if** $\text{type}[\ell] \in \{\text{comment}, \text{blank}\}$ **then**

781 22: **continue**

782 23: **end if**

783 24: normalize ℓ : fix spacing, remove trailing comments, remove module prefixes, and split compound (“;”) statements

784 25: **if** $\text{type}[\ell] = \text{control_flow}$ **then**

785 26: **if** ℓ is an **if** or **elif** statement **then**

786 27: set $\text{exists}[\ell]$ by comparing the conditional expression with corresponding **if**/**elif** statements in orig

787 28: **else if** ℓ is an **else** statement **then**

788 29: $\text{exists}[\ell] \leftarrow$ true for else-body matches

789 30: **end if**

790 31: **else if** $\text{type}[\ell] = \text{definition}$ **then**

791 32: verify existence of corresponding decorators (if any) in orig

792 33: verify existence of each argument in the definition separately in orig

793 34: assign $\text{exists}[\ell]$ based on these matches

794 35: **else if** $\text{type}[\ell] = \text{import}$ **then**

795 36: decompose into individual imports and set $\text{exists}[\ell]$ by per-import matching

796 37: **else**

797 38: compare against lines in orig and assign $\text{exists}[\ell]$ accordingly

798 39: **end if**

799 40: **end for**

800 41: **end for**

801 42: **Line Existence Rate**

802 43: $L_{\text{valid}} \leftarrow \{\ell \in G \mid \text{type}[\ell] \notin \{\text{comment}, \text{blank}\}\}$

803 44: $L_{\text{exist}} \leftarrow \{\ell \in L_{\text{valid}} \mid \text{exists}[\ell] = \text{true}\}$

804 45: $r_{\text{exist}} \leftarrow \frac{|L_{\text{exist}}|}{|L_{\text{valid}}|}$

Figure 5: Details of Line Execution and Line Existence (Part 3): Line Existence

Table 4: Details of the GitHub repositories used as the test set.

Repository	URL	License
flask	https://github.com/pallets/flask	BSD 3-Clause
requests	https://github.com/psf/requests	Apache-2.0
pylint	https://github.com/pylint-dev/pylint	GPL 2.0
scikit-learn	https://github.com/scikit-learn/scikit-learn	BSD 3-Clause
seaborn	https://github.com/mwaskom/seaborn	BSD 3-Clause
debug-gym	https://github.com/microsoft/debug-gym	MIT

809 how many lines are marked as existing in the original repository over all lines except for comments or blanks (line 43-45 in Figure 5).

810 C EXPERIMENTAL SETTING
811812 C.1 FRAMEWORK
813

814 We evaluate experiments with three agentic frameworks: mini-SWE-Agent (Yang et al., 2024),
815 SWE-Agent (Yang et al., 2024), and Copilot (Microsoft, 2025). Unless otherwise noted, all ex-
816 periments are run in the default GISTIFY setup, where the model is restricted from executing any
817 commands (e.g., `python`, `pytest`). SWE-Agent and Copilot Agent enable LLMs to interact with
818 a codebase through a suite of tools, including bash commands. These tools support capabilities such
819 as viewing, searching, editing, and creating files or directories. In addition, Copilot Agent extends
820 this functionality with browser integration, explicit reasoning, and API usage. mini-SWE-agent is
821 a simplified variant of SWE-Agent that only supports bash commands. Despite its minimal design,
822 it achieves strong performance on the SWE-Bench Verified benchmark (Jimenez et al., 2023). For
823 both mini-SWE-Agent and SWE-Agent, we set the maximum number of steps to 50 and run them
824 in the same Docker environment, using the current version of the repositories.

825 C.2 EXPERIMENTAL TEST SET CONSTRUCTION
826

827 Table 4 summarizes the repositories used in our evaluation. For each repository, we begin by ex-
828 tracting all available test cases, including parameterized ones. For experimental test runs, we group
829 tests⁴ that share the same base structure but differ only in parameterization, treating them as a single
830 test. During evaluation, however, we execute all parameterized instances and measure how many
831 are passed, thereby assessing execution fidelity. Finally, we filter out environment-dependent tests,
832 such as those requiring relative file paths or fixed module locations. In the main experiments, we
833 used 25 test instances for each of the six codebases, and the analysis was conducted using 50 test
834 instances from the pylint codebase.

835 C.3 PROMPT FOR GISTIFY
836

837 Figure 6 shows the prompt used in the main experiments.
838

839 C.4 PROVIDING SPECIFIC PARAMETERS TO COMMANDS TENDS TO MAKE MODELS
840 GENERATE PARAMETER-SPECIFIC GISTIFIED FILES
841

842 We observe that when specific command-line parameters are provided, models often adapt the gen-
843 erated gistified file to those parameters rather than producing a fully general solution. Examples of
844 this parameter-specific behavior are shown in Figures 7 and 8. Accordingly, in our experiments, we
845 group test cases based on the parameters provided to the command.

846 C.5 ADDITIONAL HYPERPARAMETERS
847

848 All experiments were executed with the same base set of hyperparameters. For framework specific
849 hyperparameters (e.g. cache management, system prompts, tools), we use the default configuration
850 made available by each framework, except for `vscode-copilot-chat`, where we have listed the tools
851 used in the experiments can be found in 5.

852 All the models used in the were configured with a maximum context window length of 128K. We
853 used a default temperature of 1 and performed a single run for each configuration.
854

855 D RESULTS
856858 D.1 EXAMPLE OF GISTIFIED FILE
859

860 Figure 9 and Figure 10 show two gistified files on the same test case with different models; each
861 model succeed or fail in generating a gistified file with execution fidelity of 100% and 0%, re-

863 ⁴We adopt this grouping design as we observe that models often overfit to specific values when parameters
864 are provided. See Appendix C.4 for more details.

864
865

Prompt for GISTIFY

866
867
868
869
870
871

I've uploaded a python code repository in the directory {working dir}.
 There is an original test invocation (the command that reproduces behavior we want to preserve): problem statement Your job: create a single file named 'concise.py' saved at {working dir} that is **self-contained**, **minimal**, and **executable**, and when used in place of the original test run reproduces the same runtime behavior and outputs. Follow the instructions below when creating the file.

872
873
874
875
876

OUTPUT

- Produce one file only: '{working dir}/concise.py'.
- The assistant must return only the contents of 'concise.py' (no extra files, no analysis, no commentary).

877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909

HIGH-LEVEL RULES for creating 'concise.py'

1. Inline internal dependencies

- * Copy into 'concise.py' every function, class, or top-level code from the files inside {working dir} that is executed when running {problem statement}.
- * Do not use 'import' statements for modules defined in {working dir}.

2. Remove unexecuted lines

- * When copying lines in 'concise.py', keep only the lines that is actually executed when running {problem statement}.
- * Delete unused functions, classes, variables, if-else, imports, and unreachable branches.
- * Ensure the file remains syntactically correct and minimal after removal.

3. Preserve original source lines

- * Do not rewrite or reformat lines unless necessary to keep the files valid.
- * Do not arbitrary generate new lines that do not exist in the original {working dir} files.
- * You may adjust indentation, remove empty 'else' blocks, or adapt 'try-except' structures only when required to preserve correctness.

4. Keep external imports

- * Leave imports to external libraries, frameworks, or standard runtime libraries unchanged.
- * Only remove or inline dependencies that come from {working dir}.

5. No shortcuts or cheating

- * Do not stub, fake, or monkey-patch external modules.
- * Do not reimplement or newly add third-party libraries.
- * Do not hard-code outputs
- * Do not replace test logic with simplified equivalents

6. Preserve test behavior

- * The test function must remain unchanged, except for import adjustments needed to reference inlined code.
- * The output, exceptions, or exit codes must match the original run of {problem statement}.

7. Do not execute the code

- * Do not run or simulate the program (e.g., with 'pytest', 'python', or any other tools)

Figure 6: Base Prompt Template for GISTIFY Task.

910 respectively. In the successful case (Figure 9), the generated file handles both parameters correctly,
 911 achieving a 100% line existence rate, a 65.5% execution rate, and a test F_1 score of 100. In contrast,
 912 the failed case (Figure 10) cannot execute due to a missing `import pytest` statement. More-
 913 over, the hallucinated test function yields a test F_1 score of 0, and the file shows a much lower line
 914 existence rate of 28%.

D.2 ERROR ANALYSIS OVER EXECUTION FAILURE

915
916
917 We categorize errors into four types:

```

918 @pytest.mark.parametrize(
919     "value, expected",
920     (
921         ("application/xml", ("application/xml", {})),
922         (
923             "application/json ; charset=utf-8",
924             ("application/json", {"charset": "utf-8"}),
925         ),
926         ("text/plain", ("text/plain", {})),
927         ...
928     )
929     def test__parse_content_type_header(value, expected):
930         assert _parse_content_type_header(value) == expected
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
20100
20101
20102
20103
20104
20105
20106
20107
20108
20109
20110
20111
20112
20113
20114
20115
20116
20117
20118
20119
20120
20121
20122
20123
20124
20125
20126
20127
20128
20129
20130
20131
20132
20133
20134
20135
20136
20137
20138
20139
20140
20141
20142
20143
20144
20145
20146
20147
20148
20149
20150
20151
20152
20153
20154
20155
20156
20157
20158
20159
201510
201511
201512
201513
201514
201515
201516
201517
201518
201519
201520
201521
201522
201523
201524
201525
201526
201527
201528
201529
201530
201531
201532
201533
201534
201535
201536
201537
201538
201539
201540
201541
201542
201543
201544
201545
201546
201547
201548
201549
201550
201551
201552
201553
201554
201555
201556
201557
201558
201559
201560
201561
201562
201563
201564
201565
201566
201567
201568
201569
201570
201571
201572
201573
201574
201575
201576
201577
201578
201579
201580
201581
201582
201583
201584
201585
201586
201587
201588
201589
201590
201591
201592
201593
201594
201595
201596
201597
201598
201599
2015100
2015101
2015102
2015103
2015104
2015105
2015106
2015107
2015108
2015109
2015110
2015111
2015112
2015113
2015114
2015115
2015116
2015117
2015118
2015119
2015120
2015121
2015122
2015123
2015124
2015125
2015126
2015127
2015128
2015129
2015130
2015131
2015132
2015133
2015134
2015135
2015136
2015137
2015138
2015139
2015140
2015141
2015142
2015143
2015144
2015145
2015146
2015147
2015148
2015149
2015150
2015151
2015152
2015153
2015154
2015155
2015156
2015157
2015158
2015159
20151510
20151511
20151512
20151513
20151514
20151515
20151516
20151517
20151518
20151519
20151520
20151521
20151522
20151523
20151524
20151525
20151526
20151527
20151528
20151529
20151530
20151531
20151532
20151533
20151534
20151535
20151536
20151537
20151538
20151539
20151540
20151541
20151542
20151543
20151544
20151545
20151546
20151547
20151548
20151549
20151550
20151551
20151552
20151553
20151554
20151555
20151556
20151557
20151558
20151559
20151560
20151561
20151562
20151563
20151564
20151565
20151566
20151567
20151568
20151569
20151570
20151571
20151572
20151573
20151574
20151575
20151576
20151577
20151578
20151579
20151580
20151581
20151582
20151583
20151584
20151585
20151586
20151587
20151588
20151589
20151590
20151591
20151592
20151593
20151594
20151595
20151596
20151597
20151598
20151599
201515100
201515101
201515102
201515103
201515104
201515105
201515106
201515107
201515108
201515109
201515110
201515111
201515112
201515113
201515114
201515115
201515116
201515117
201515118
201515119
201515120
201515121
201515122
201515123
201515124
201515125
201515126
201515127
201515128
201515129
201515130
201515131
201515132
201515133
201515134
201515135
201515136
201515137
201515138
201515139
201515140
201515141
201515142
201515143
201515144
201515145
201515146
201515147
201515148
201515149
201515150
201515151
201515152
201515153
201515154
201515155
201515156
201515157
201515158
201515159
201515160
201515161
201515162
201515163
201515164
201515165
201515166
201515167
201515168
201515169
201515170
201515171
201515172
201515173
201515174
201515175
201515176
201515177
201515178
201515179
201515180
201515181
201515182
201515183
201515184
201515185
201515186
201515187
201515188
201515189
201515190
201515191
201515192
201515193
201515194
201515195
201515196
201515197
201515198
201515199
201515200
201515201
201515202
201515203
201515204
201515205
201515206
201515207
201515208
201515209
201515210
201515211
201515212
201515213
201515214
201515215
201515216
201515217
201515218
201515219
201515220
201515221
201515222
201515223
201515224
201515225
201515226
201515227
201515228
201515229
201515230
201515231
201515232
201515233
201515234
201515235
201515236
201515237
201515238
201515239
201515240
201515241
201515242
201515243
201515244
201515245
201515246
201515247
201515248
201515249
201515250
201515251
201515252
201515253
201515254
201515255
201515256
201515257
201515258
201515259
201515260
201515261
201515262
201515263
201515264
201515265
201515266
201515267
201515268
201515269
201515270
201515271
201515272
201515273
201515274
201515275
201515276
201515277
201515278
201515279
201515280
201515281
201515282
201515283
201515284
201515285
201515286
201515287
201515288
201515289
201515290
201515291
201515292
201515293
201515294
201515295
201515296
201515297
201515298
201515299
201515200
201515201
201515202
201515203
201515204
201515205
201515206
201515207
201515208
201515209
201515210
201515211
201515212
201515213
201515214
201515215
201515216
201515217
201515218
201515219
201515220
201515221
201515222
201515223
201515224
201515225
201515226
201515227
201515228
201515229
201515230
201515231
201515232
201515233
201515234
201515235
201515236
201515237
201515238
201515239
201515240
201515241
201515242
201515243
201515244
201515245
201515246
201515247
201515248
201515249
201515250
201515251
201515252
201515253
201515254
201515255
201515256
201515257
201515258
201515259
201515260
201515261
201515262
201515263
201515264
201515265
201515266
201515267
201515268
201515269
201515270
201515271
201515272
201515273
201515274
201515275
201515276
201515277
201515278
201515279
201515280
201515281
201515282
201515283
201515284
201515285
201515286
201515287
201515288
201515289
201515290
201515291
201515292
201515293
201515294
201515295
201515296
201515297
201515298
201515299
201515200
201515201
201515202
201515203
201515204
201515205
201515206
201515207
201515208
201515209
201515210
201515211
201515212
201515
```

```

972
973     @pytest.mark.parametrize(
974         "url, expected",
975         (
976             ("http://192.168.0.1:5000/", True),
977             ...
978             ("http://google.com:5000/v1.0/", False),
979         ),
980     )
981     def test_should_bypass_proxies_no_proxy(url, expected, monkeypatch):
982         """Tests for function should_bypass_proxies to check if proxy
983         can be bypassed or not using the 'no_proxy' argument
984         """
985         no_proxy = "192.168.0.0/24,127.0.0.1,localhost.localdomain,172.16.1.1
986         """
987         # Test 'no_proxy' argument
988         assert should_bypass_proxies(url, no_proxy=no_proxy) == expected
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002 D.3 TOOLS AVAILABLE IN GITHUB COPILOT
1003
1004 Table 5 shows the list of available tools in Github Copilot.
1005
1006
1007 D.4 CHANGE TEST
1008
1009 even high performing models and frameworks (especially GPT-5 and GPT-5-mini) seems to modify
1010 test codes even though explicitly mentioned not to. We observed three common modification: (1)
1011 removing the test function but move the lines in the test function under the "__main__" guard (e.g.,
1012 Figure 20), (2) adding the "__main__" guard even though unnecessary (e.g., Figure 21), and (3)
1013 mocking a minimal in-memory package to bypass missing dependencies and force the test to run
1014 (e.g., Figure 22).
1015
1016
1017 D.5 ADDITIONAL METRICS
1018
1019 Table 6 shows the result of additional evaluation metrics, including the Average Pytest Pass Rate,
1020 which is defined as the average test pass rate over cases with at least one successful run, and the
1021 Test F1 Score, which quantifies the line-wise  $F_1$  existence between the test functions in the original
1022 codebase and those in the gistified fie.
1023
1024 GPT-5 shows a notably higher Average Pytest Pass Rate, indicating that among the ones they suc-
1025 cessfully generate, they tend to pass all pytest. For the Test  $F_1$  Score, Claude-4 shows the highest
1026 performance, aligning with the trend discussed in Section 4.4.

```

```

1026
1027     # Licensed under the GPL: https://www.gnu.org/licenses/old-licenses/gpl-2.0.html
1028     # For details: https://github.com/pylint-dev/pylint/blob/main/LICENSE
1029     # Copyright (c) https://github.com/pylint-dev/pylint/blob/main/CONTRIBUTORS.txt
1030
1031     from __future__ import annotations
1032
1033     import os
1034     from collections.abc import Sequence
1035     from typing import Any
1036
1037     import pytest
1038
1039
1040     def discover_package_path(modulepath: str, source_roots: Sequence[str]) -> str:
1041         """Discover package path from one its modules and source roots."""
1042         dirname = os.path.realpath(os.path.expanduser(modulepath))
1043         if not os.path.isdir(dirname):
1044             dirname = os.path.dirname(dirname)
1045
1046         # Look for a source root that contains the module directory
1047         for source_root in source_roots:
1048             source_root = os.path.realpath(os.path.expanduser(source_root))
1049             if os.path.commonpath([source_root, dirname]) in [dirname, source_root]:
1050                 return source_root
1051
1052         # Fall back to legacy discovery by looking for __init__.py upwards as
1053         # it's the only way given that source root was not found or was not provided
1054         while True:
1055             if not os.path.exists(os.path.join(dirname, "__init__.py")):
1056                 return dirname
1057             olddirname = dirname
1058             dirname = os.path.dirname(dirname)
1059             if olddirname == dirname:
1060                 return os.getcwd()
1061
1062
1063     @pytest.mark.parametrize(
1064         "py_mod_base_name",
1065         ("__init__", "impl"),
1066         ids=("explicit-namespace", "implicit-namespace"),
1067     )
1068     def test_discover_package_path_source_root_as_parent(
1069         py_mod_base_name: str,
1070         tmp_path: Any,
1071     ) -> None:
1072         """Test discover_package_path when source root is a parent of the module."""
1073         # Create this temporary structure:
1074         # /tmp_path/
1075         #   project/
1076         #     my-package/
1077         #       __init__.py
1078         project_dir = tmp_path / "project"
1079         package_dir = project_dir / "mypackage"
1080         package_dir.mkdir(parents=True)
1081         (package_dir / f"{py_mod_base_name}.py").touch()
1082
1083         # Test with project_dir as source root (parent of package)
1084         result = discover_package_path(str(package_dir), [str(project_dir)])
1085         assert result == str(project_dir)
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
20100
20101
20102
20103
20104
20105
20106
20107
20108
20109
20110
20111
20112
20113
20114
20115
20116
20117
20118
20119
20120
20121
20122
20123
20124
20125
20126
20127
20128
20129
20130
20131
20132
20133
20134
20135
20136
20137
20138
20139
20140
20141
20142
20143
20144
20145
20146
20147
20148
20149
20150
20151
20152
20153
20154
20155
20156
20157
20158
20159
201510
201511
201512
201513
201514
201515
201516
201517
201518
201519
201520
201521
201522
201523
201524
201525
201526
201527
201528
201529
201530
201531
201532
201533
201534
201535
201536
201537
201538
201539
201540
201541
201542
201543
201544
201545
201546
201547
201548
201549
201550
201551
201552
201553
201554
201555
201556
201557
201558
201559
201560
201561
201562
201563
201564
201565
201566
201567
201568
201569
201570
201571
201572
201573
201574
201575
201576
201577
201578
201579
201580
201581
201582
201583
201584
201585
201586
201587
201588
201589
201590
201591
201592
201593
201594
201595
201596
201597
201598
201599
2015100
2015101
2015102
2015103
2015104
2015105
2015106
2015107
2015108
2015109
2015110
2015111
2015112
2015113
2015114
2015115
2015116
2015117
2015118
2015119
2015120
2015121
2015122
2015123
2015124
2015125
2015126
2015127
2015128
2015129
2015130
2015131
2015132
2015133
2015134
2015135
2015136
2015137
2015138
2015139
2015140
2015141
2015142
2015143
2015144
2015145
2015146
2015147
2015148
2015149
2015150
2015151
2015152
2015153
2015154
2015155
2015156
2015157
2015158
2015159
20151510
20151511
20151512
20151513
20151514
20151515
20151516
20151517
20151518
20151519
20151520
20151521
20151522
20151523
20151524
20151525
20151526
20151527
20151528
20151529
20151530
20151531
20151532
20151533
20151534
20151535
20151536
20151537
20151538
20151539
20151540
20151541
20151542
20151543
20151544
20151545
20151546
20151547
20151548
20151549
20151550
20151551
20151552
20151553
20151554
20151555
20151556
20151557
20151558
20151559
20151560
20151561
20151562
20151563
20151564
20151565
20151566
20151567
20151568
20151569
20151570
20151571
20151572
20151573
20151574
20151575
20151576
20151577
20151578
20151579
20151580
20151581
20151582
20151583
20151584
20151585
20151586
20151587
20151588
20151589
20151590
20151591
20151592
20151593
20151594
20151595
20151596
20151597
20151598
20151599
201515100
201515101
201515102
201515103
201515104
201515105
201515106
201515107
201515108
201515109
201515110
201515111
201515112
201515113
201515114
201515115
201515116
201515117
201515118
201515119
201515120
201515121
201515122
201515123
201515124
201515125
201515126
201515127
201515128
201515129
201515130
201515131
201515132
201515133
201515134
201515135
201515136
201515137
201515138
201515139
201515140
201515141
201515142
201515143
201515144
201515145
201515146
201515147
201515148
201515149
201515150
201515151
201515152
201515153
201515154
201515155
201515156
201515157
201515158
201515159
201515160
201515161
201515162
201515163
201515164
201515165
201515166
201515167
201515168
201515169
201515170
201515171
201515172
201515173
201515174
201515175
201515176
201515177
201515178
201515179
201515180
201515181
201515182
201515183
201515184
201515185
201515186
201515187
201515188
201515189
201515190
201515191
201515192
201515193
201515194
201515195
201515196
201515197
201515198
201515199
201515200
201515201
201515202
201515203
201515204
201515205
201515206
201515207
201515208
201515209
201515210
201515211
201515212
201515213
201515214
201515215
201515216
201515217
201515218
201515219
201515220
201515221
201515222
201515223
201515224
201515225
201515226
201515227
201515228
201515229
201515230
201515231
201515232
201515233
201515234
201515235
201515236
201515237
201515238
201515239
201515240
201515241
201515242
201515243
201515244
201515245
201515246
201515247
201515248
201515249
201515250
201515251
201515252
201515253
201515254
201515255
201515256
201515257
201515258
201515259
201515260
201515261
201515262
201515263
201515264
201515265
201515266
201515267
201515268
201515269
201515270
201515271
201515272
201515273
201515274
201515275
201515276
201515277
201515278
201515279
201515280
201515281
201515282
201515283
201515284
201515285
201515286
201515287
201515288
201515289
201515290
201515291
201515292
201515293
201515294
201515295
201515296
201515297
201515298
201515299
201515300
201515301
201515302
201515303
201515304
201515305
201515306
201515307
201515308
201515309
201515310
201515311
201515312
201515313
201515314
201515315
201515316
201515317
201515318
201515319
201515320
201515321
201515322
201515323
201515324
201515325
201515326
201515327
201515328
201515329
201515330
201515331
201515332
201515333
201515334
201515335
201515336
201515337
201515338
201515339
201515340
201515341
201515342
201515343
201515344
201515345
201515346
201515347
201515348
201515349
201515350
201515351
201515352
201515353
201515354
201515355
201515356
201515357
201515358
201515359
201515360
201515361
201515362
201515363
201515364
201515365
201515366
201515367
201515368
201515369
201515370
201515371
201515372
201515373
201515374
201515375
201515376
201515377
201515378
201515379
201515380
201515381
201515382
201515383
201515384
201515385
201515386
201515387
201515388
201515389
201515390
201515391
201515392
201515393
201515394
201515395
201515396
201515397
201515398
201515399
201515400
201515401
201515402
201515403
201515404
201515405
201515406
201515407
201515408
201515409
201515410
201515411
201515412
201515413
201515414
201515415
201515416
201515417
201515418
201515419
201515420
20
```

```

1080
1081     import os
1082     import sys
1083     from contextlib import contextmanager
1084     from pathlib import Path
1085
1086     def discover_package_path(path, _search_paths):
1087         """
1088             Minimal version: accept a filesystem path (string or Path) and return a Path
1089             to that directory. This matches the usage in the specific test where a
1090             path to the tests data directory is passed.
1091         """
1092         if isinstance(path, Path):
1093             p = path
1094         else:
1095             p = Path(path)
1096         if p.exists() and p.is_dir():
1097             return p
1098         raise ValueError("Cannot determine package path")
1099
1100     @contextmanager
1101     def augmented_sys_path(paths):
1102         """
1103             Minimal context manager: set sys.path to the parent directories of the
1104             provided paths (in order), restoring the original sys.path on exit.
1105         """
1106         original = list(sys.path)
1107         try:
1108             new = []
1109             for p in paths:
1110                 pth = Path(p)
1111                 # Ensure we use the parent directory (the "source root" in the test)
1112                 parent = pth.parent
1113                 new.append(str(parent))
1114             sys.path[:] = new
1115             yield
1116         finally:
1117             sys.path[:] = original
1118
1119     @pytest.mark.parametrize(
1120         "py_mod_base_name",
1121         ("__init__"),
1122         ids=({"explicit-namespace"},),
1123     )
1124     def test_discover_package_path_source_root_as_parent():
1125         # This test's behavior is preserved: it uses discover_package_path on the
1126         # tests data directory and ensures augmented_sys_path sets sys.path to its
1127         # parent (the project/tests directory).
1128         TEST_DATA_DIR = os.path.abspath(os.path.join(os.path.dirname(__file__), "tests", "data"))
1129         PROJECT_ROOT_DIR = os.path.abspath(os.path.join(TEST_DATA_DIR, ".."))
1130         with augmented_sys_path([discover_package_path(TEST_DATA_DIR, [])]):
1131             assert sys.path == [PROJECT_ROOT_DIR]
1132
1133

```

Figure 10: Example of failed gistified file: the code fails to import `pytest`. The model hallucinates the function `test_discover_package_path_source_root_as_parent()`, resulting in a test F_1 score of 0 and a low line-existence rate of 28.0%

Global Information via Tools We experiment with two tools that provide global information: *RepoGraph* and *Tracing*. Details of the information provided to the model about each tool are shown in Figure 16.

RepoGraph (Ouyang et al., 2024) is a plug-in module designed to help LLMs leverage the codebase-level structure. It parses code at the line level, extracts relationships, and constructs a graph where each node represents a line of code and each edge encodes dependencies between code definitions and their references. Thereby, when given a specific module, it returns the relationship with other modules as represented within the constructed graph.

Tracing is a tool that uses the tracer provided from the `sys` module to execute a command and track which components of the codebase are accessed. When the model uses the tool with a specific command, the tool provides the model with the files and functions touched when running the command, in the order in which they are encountered.

```

1134
1135     @click.option("--all-methods", is_flag=True, help="Show HEAD and OPTIONS
1136         methods.")
1137     @with_appcontext
1138     def routes_command(sort, all_methods):
1139         """Show all registered routes with endpoints and methods."""
1140         from flask import current_app
1141
1142         rules = list(current_app.url_map.iter_rules())
1143
1144         if not rules:
1145             click.echo("No routes were registered.")
1146         return
1147
1148
1149     T = t.TypeVar("T")
1150
1151     class ConfigAttribute(t.Generic[T]):
1152         """Makes an attribute forward to the config"""
1153
1154         def __init__(
1155             self, name: str, get_converter: t.Callable[[t.Any], T] | None =
1156             None
1157             ) -> None:
1158                 self.__name__ = name
1159                 self.get_converter = get_converter
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3139
3140
3141
3142
3143
3143
3144
3145
3146
3146
3147
3147
3148
3148
3149
3149
3150
3150
3151
3151
3152
3152
3153
3153
3154
3154
3155
3155
3156
3156
3157
3157
3158
3158
3159
3159
3160
3160
3161
3161
3162
3162
3163
3163
3164
3164
3165
3165
3166
3166
3167
3167
3168
3168
3169
3169
3170
3170
3171
3171
3172
3172
3173
3173
3174
3174
3175
3175
3176
3176
3177
3177
3178
3178
3179
3179
3180
3180
3181
3181
3182
3182
3183
3183
3184
3184
3185
3185
3186
3186
3187
3187
3188
3188
3189
3189
3190
3190
3191
3191
3192
3192
3193
3193
3194
3194
3195
3195
3196
3196
3197
3197
3198
3198
3199
3199
3200
3200
3201
3201
3202
3202
3203
3203
3204
3204
3205
3205
3206
3206
3207
3207
3208
3208
3209
3209
3210
3210
3211
3211
3212
3212
3213
3213
3214
3214
3215
3215
3216
3216
3217
3217
3218
3218
3219
3219
3220
3220
3221
3221
3222
3222
3223
3223
3224
3224
3225
3225
3226
3226
3227
3227
3228
3228
3229
3229
3230
3230
3231
3231
3232
3232
3233
3233
3234
3234
3235
3235
3236
3236
3237
3237
3238
3238
3239
3239
3240
3240
3241
3241
3242
3242
3243
3243
3244
3244
3245
3245
3246
3246
3247
3247
3248
3248
3249
3249
3250
3250
3251
3251
3252
3252
3253
3253
3254

```

1188	Tool	Description
1189	copilot.getNotebookSummary	Returns the list of Notebook cells with id, types, line ranges, language, execution info, and output mime types. Useful for getting cell IDs, execution order, and outputs.
1190	edit_notebook_file	Edit an existing Notebook file in the workspace. Supports inserting, deleting, or editing cells while preserving whitespace and indentation.
1191	apply_patch	Edit text files using a special diff/patch format. Do not use for Jupyter notebooks.
1192	semantic_search	Run a natural language search for relevant code or documentation comments in the workspace.
1193	create_directory	Create a new directory structure in the workspace (like <code>mkdir -p</code>).
1194	create_file	Create a new file with specified content. Automatically creates directories if they do not exist.
1195	file_search	Search for files in the workspace by glob pattern (e.g., <code>**/*.*</code>). Returns matching paths only.
1196	test_search	For a source file, find the corresponding test file, and vice versa.
1197	grep_search	Fast text or regex search in the workspace. Useful for exact string or regex queries.
1198	run_notebook_cell	Run a code cell in a notebook file and return the output. Avoid running Markdown cells.
1199	read_notebook_cell_output	Retrieve the latest output for a notebook cell, even if not run in the current session.
1200	get_search_view_results	Returns results from the search view.
1201	github_repo	Search a GitHub repository for relevant code snippets. Use only for external repos, not local workspaces.
1202	insert_edit_into_file	Insert or edit code in an existing file using minimal hints, avoiding duplication of unchanged code.
1203	install_extension	Install an extension in VS Code. Used only during workspace creation.
1204	list_dir	List the contents of a directory (folders and files).
1205	create_new_jupyter_notebook	Generate a new Jupyter Notebook (.ipynb) in VS Code.
1206	create_new_workspace	Set up a complete new project (scaffolding, dependencies, config, boilerplate).
1207	get_project_setup_info	Provides project setup information for a VS Code workspace after workspace creation.
1208	read_file	Read the contents of a file. Supports offsets and limits for large files.
1209	open_simple_browser	Preview or open a URL in VS Code's Simple Browser.
1210	test_failure	Include test failure information in the prompt.
1211	think	Think deeply about a request and log structured reasoning (no execution). Useful for planning, debugging, and brainstorming.
1212	get_vscode_api	Retrieve comprehensive VS Code API documentation and references for extension development.
1213	run_vscode_command	Run a VS Code command by ID with arguments. Used mainly in workspace creation.
1214	fetch_webpage	Fetch main content from a webpage for summarization or analysis.

1221 Table 5: Available tools and their descriptions. We note that many tools available to the agent are
 1222 never used.

1224 Table 6: Average Pytest Pass Rate and Test F_1 Score of different models using SWE-Agent on the
 1225 main table (Table 1) test dataset.

1227	Models	Execution Fidelity	Average Pytest Pass Rate	Test F_1 Score
1228	GPT-5-mini	30.9	49.2	47.9
1229	GPT-5	30.7	88.8	45.0
1230	Claude-3.7	40.7	61.9	55.9
1231	Claude-4	56.7	72.2	60.0

1232 Table 7: Analysis of tool usage during the GISTIFY task

1235	Models	Avg. tool usage	view	search	execute	other
1236	GPT-5-mini	10.8	71.9	9.8	1.7	16.6
1237	GPT-5	18.5	72.4	8.3	3.3	16.1
1238	Claude-Sonnet-3.7	17.3	67.5	10.1	4.5	17.9
1239	Claude-Sonnet-4	19.3	74.6	2.1	11.8	11.5

```

1242
1243     @pytest.mark.parametrize(
1244         "value, expected",
1245         (
1246             ('foo="is a fish", bar="as well"', {"foo": "is a fish", "bar": "as well"}),
1247             ("key_without_value", {"key_without_value": None}),
1248         ),
1249     )
1250     def test_parse_dict_header(value, expected):
1251         assert parse_dict_header(value) == expected
1252
1253         (a) Original Test Case
1254
1255         assert parse_dict_header('foo="is a fish", bar="as well"') == {"foo": "is
1256             a fish", "bar": "as well"}
1257         assert parse_dict_header("key_without_value") == {"key_without_value": None}
1258
1259         (b) Gistified File

```

Figure 13: The test function `test_parse_dict_header` is simplified: in the original, it used `@pytest.mark.parametrize` to feed multiple input/expected pairs into one function; in the gistified version, this is replaced with two direct `assert` statements, one per case.

Behavior Reading

How to Operate:

1. Examine the test file and the test function used for {problem statement}
2. Identify which module used by these functions are defined in {working dir}
3. Copy and inline the code from those modules into ‘concise.py’
4. Check these modules for any internal functions or classes and inline them as needed.
5. Repeat this process recursively until all internal dependencies are inlined.
6. Do not forget to copy and paste external imports.

Figure 14: Prompt for *Reading* strategy.

Trace Reasoning

How to Operate:

1. Predict the execution traces.
2. Follow the traces and inline (copy) only the necessary executed lines into ‘concise.py’
3. Repeat until all traces are fully handled.

Figure 15: Prompt for *Tracing* strategy.

Tool Information

* search repo (reograph): searches in the mentioned repository with a specific function or class, and returns the def and ref relations for the search term.
 * trace info (tracing): returns the trace log when running the ‘pytest’ command for the test currently being debugged in the given repo

Figure 16: Details of the information provided to the model about each tool.

```

1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
{
  "role": "assistant",
  "tool_calls": {
    "arguments": "{\"command\": \"cd /app && python -m pytest concise.py::TestDeprecationOptions::test_new_names -v\"}",
    "name": "bash"
  }
},
{
  "role": "tool",
  "content": [
    {
      "type": "text",
      "text": "OBSERVATION: test session starts ...AssertionError\n...pylint: error: unrecognized arguments: /app/regtest_data/empty.py\n\\n FAILED concise.py::TestDeprecationOptions::test_new_names ..."
    }
  ],
  ...
  (fix concise.py based on observation)
  ...
{
  "role": "assistant",
  "tool_calls": {
    "arguments": "{\"command\": \"cd /app && python -m pytest tests/config/test_argparse_config.py::TestDeprecationOptions::test_new_names -v -s\"}",
    "name": "bash"
  }
},
...
{
  "role": "assistant",
  "content": "Excellent! The messages are identical. Let me verify my implementation removes all unused imports and only includes what's needed."
}

```

Figure 17: Trajectory when including execution tool. Models often *iteratively* execute the given codebase or generated gistified file to ensure that it operates expectedly.

E.2 STATIC CODING LLM

Figure 18 presents the performance of static and dynamic LLM, including a static model augmented with a *single-turn, multi-step chain-of-thought* prompt. To test whether explicit reasoning and planning can mitigate the limitations of static coding LLMs, which must output the entire program in a single turn, we compare this CoT-augmented static model against both the baseline static model and the dynamic models.

Overall, we observe a trend similar to the baseline static model without a chain-of-thought: adding chain-of-thought yields only a modest improvement over the static model without it, and it still performs worse than the dynamic models, underscoring the importance of multi-turn generation and tool use.

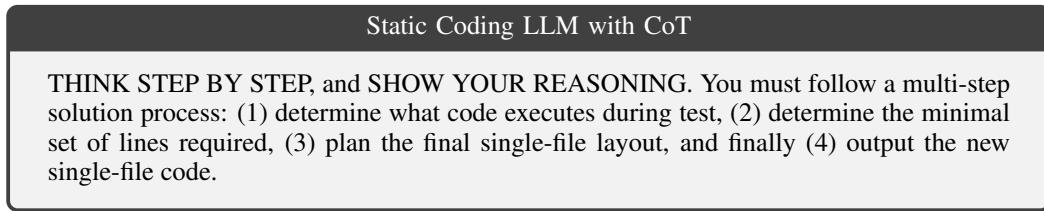
Relative to the baseline static model, the CoT-augmented version slightly improves line-execution rates but slightly



Figure 18: Performance over Static and Dynamic Coding LLM

1350 decreases line-existence. We hypothesize that this is because the CoT procedure encourages the
 1351 model to focus on minimal set of necessary lines.
 1352

1353 Details of the prompt used in this experiment are provided in Figure 19.



1361
 1362 Figure 19: Prompt added to the static coding LLM to enable chain-of-thought reasoning. A similar
 1363 instruction is also included in the system prompt.
 1364

1365 E.3 TOOL USAGE RATES

1366
 1367 Table 7 shows the statistics on tool usage across models using SWE-bench. We group various tools
 1368 into four categories: view, search, execute, and other, which includes all remaining tools. For all
 1369 models, we compute usage rates both with and without execution enabled, and then average across
 1370 the two settings.

1371 Among all models, Claude-4 exhibits the highest average tool usage for each test cases, followed by
 1372 GPT-5, Claude-3.7, and GPT-5-mini. In terms of specific functionality, Claude-4 shows the highest
 1373 rate of both view and execute tool usage, while Claude-3.7 shows the highest usage of the search
 1374 tool. To generate a high-quality gisted file, a model must effectively view relevant files and copy
 1375 only the necessary content. The strong performance of Claude-4 on line existence may be related to
 1376 its high usage of the *view* tool. Also, the *execution* tool tends to support correctness verification of
 1377 the generated file, which would lead to high execution fidelity.
 1378

1379 F USER STUDY ON THE DEFINITION

1380 We conduct a user study to assess whether our proposed metrics, line existence and line execution,
 1381 successfully capture the task’s intended notions of *faithfulness* and *minimality*, respectively.
 1382

1383 We recruit three software/AI engineers as annotators and provide them with 15 test cases. Each test
 1384 case contains a pair of gisted files generated by different frameworks or different models. For each
 1385 pair, annotators were asked to choose the file that better satisfied the task’s criteria of minimality and
 1386 faithfulness. Annotators were given: (1) a description of how the gisted files were constructed,
 1387 (2) the definitions of minimality and faithfulness used in our task, (3) the two gisted files for
 1388 comparison, and (4) execution-tracing information to help them understand the flow of each test
 1389 run. Figure 23 shows the instructions provided to annotators.
 1390

1391 To measure alignment between human judgment and our metrics, we computed Cohen’s kappa
 1392 correlation between the annotators’ selections and the rankings produced by line existence and line
 1393 execution. We observed an average Cohen’s kappa of 0.61 (0.52, 0.72, 0.58) for minimality with
 1394 line execution and 0.76 (0.81, 0.71, 0.77) for faithfulness with line existence, indicating that our
 1395 metrics correspond well with human judgments of minimality and faithfulness.
 1396

1397
 1398
 1399
 1400
 1401
 1402
 1403

```

1404
1405
1406
1407
1408 class TestGetNetrcAuth:
1409     def test_works(self, tmp_path, monkeypatch):
1410         netrc_path = tmp_path / ".netrc"
1411         monkeypatch.setenv("NETRC", str(netrc_path))
1412         with open(netrc_path, "w") as f:
1413             f.write("machine example.com login aaaa password bbbb\n")
1414         auth = get_netrc_auth("http://example.com/thing")
1415         assert auth == ("aaaa", "bbbb")

```

(a) Original Test Case

```

1416
1417 if __name__ == "__main__":
1418     # Reproduce tests/test_utils.py::TestGetNetrcAuth::test_works
1419     with tempfile.TemporaryDirectory() as tmpdir:
1420         netrc_path = os.path.join(tmpdir, ".netrc")
1421         os.environ["NETRC"] = netrc_path
1422         with open(netrc_path, "w") as f:
1423             f.write("machine example.com login aaaa password bbbb\n")
1424         auth = get_netrc_auth("http://example.com/thing")
1425         assert auth == ("aaaa", "bbbb")

```

(b) Gistified File

Figure 20: Test Modification Case 1: The test `TestGetNetrcAuth.test_works` is converted from a pytest unit test into a standalone script.

```

1427
1428
1429
1430
1431
1432
1433
1434
1435
1436 # Test class and method - preserved unchanged
1437 class TestArgparseOptionsProviderMixin:
1438     """Tests for the argparse implementation of OptionsProviderMixIn.
1439
1440     The logger checker is used as an example checker for this
1441     implementation.
1442     """
1443
1444     @staticmethod
1445     def test_logger_without_options() -> None:
1446         """Check that we raise messages when we do not supply any options
1447         ."""
1448         with pytest.raises(SystemExit) as ex:
1449             Run([LOGGING_TEST])
1450         assert ex.value.code == 2
1451
1452     # Main execution for pytest
1453 if __name__ == "__main__":
1454     test = TestArgparseOptionsProviderMixin()
1455     test.test_logger_without_options()

```

Figure 21: Test Modification Case 2: Adding unnecessary `"__main__"` guard

```

1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472 # Create a minimal in-memory 'requests' package with required submodules.
1473 requests_mod = types.ModuleType('requests')
1474 requests_mod.__path__ = []
1475 compat_mod = types.ModuleType('requests.compat')
1476 structures_mod = types.ModuleType('requests.structures')
1477
1478 # Populate compat with only what's needed by this test suite import paths
1479 .
1480 compat_mod.Mapping = Mapping
1481 compat_mod.MutableMapping = MutableMapping
1482 compat_mod.urljoin = urljoin
1483
1484 # Populate structures with the classes.
1485 structures_mod.CaseInsensitiveDict = CaseInsensitiveDict
1486 structures_mod.LookupDict = LookupDict
1487
1488 # Wire the package hierarchy and register in sys.modules.
1489 requests_mod.compat = compat_mod
1490 requests_mod.structures = structures_mod
1491 sys.modules['requests'] = requests_mod
1492 sys.modules['requests.compat'] = compat_mod
1493 sys.modules['requests.structures'] = structures_mod
1494
1495 if __name__ == '__main__':
1496     import pytest
1497     raise SystemExit(pytest.main(['-q', 'tests/test_structures.py::',
1498                                 'TestCaseInsensitiveDict::test_list']))
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511

```

Figure 22: Test Modification Case 3: Manually mocking a minimal in-memory package to bypass missing dependencies and force the test to run.

1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523

User Study Instruction

1524
1525 You are given two code files produced by a coding LLM. Both files attempt to complete
1526 the same task: create a single, minimal, self-contained file that reproduces a specific
1527 functionality of a codebase.
1528

This is the prompt we provide to the model to describe the task:

[Figure C.3]

Your task:

Evaluate the two files and determine which one is more minimal and more faithful
according to the criteria below.

1. A file is “minimal” if:

- It contains only the code truly required to reproduce the runtime behavior.
- Unused functions, classes, variables, or imports should be removed.
- The evaluator must understand the code well enough to identify which lines are actually
executed and are essential. However, we do not penalize minor extra lines that exist solely
because of formatting or structure constraints (e.g., keeping an unused method such as
try-except because the format requires it).

2. A file is “faithful” if:

- No hallucinated code is introduced. Every piece of code must come directly from the
original codebase.
- The structure of the code must stay consistent with the original. For example, moving a
method that originally lived inside a class to the top level is considered incorrect.
- Simple changes such as incorrect indentation or broken multi-line statements are not pe-
nalized as long as the intended semantics are still clear. However, structural changes that
alter the meaning of the code are penalized. For example: If the original code defines a
class with inheritance, such as class ABC(DE):, but the generated file changes it to class
ABC; then this is considered incorrect, because removing the parent class changes the ac-
tual operation of the code.

Figure 23: Instruction for User Study

1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565