

Evaluating LLM-Based 0-to-1 Software Generation in End-to-End CLI Tool Scenarios

Ruida Hu¹, Xincheng Wang¹, Chao Peng², Cuiyun Gao^{1,†}, David Lo³

Abstract—The evolution of Large Language Models (LLMs) has catalyzed a paradigm shift towards intent-driven software development, where autonomous agents are expected to design and deliver complete, runnable software systems from scratch. However, existing benchmarks fail to adequately assess this 0-to-1 generation capability due to two fundamental limitations. First, they rely on predefined structural scaffolds, which reduces the task to mere file-filling. Second, they depend on rigid white-box unit testing, which forces generated code to conform to specific internal implementations rather than validating end-to-end user-centric behavior.

To bridge this gap, we introduce CLI-Tool-Bench, a novel, structure-agnostic benchmark designed to evaluate the ground-up generation of Command-Line Interface (CLI) tools. Powered by an automated black-box differential testing framework, the benchmark comprises 94 high-quality, real-world repositories spanning diverse programming languages and complexity levels. For each task, agents are provided with an empty workspace, forcing them to autonomously handle repository planning and dependencies. We evaluate the generated software by executing it in isolated sandboxes. The system-level side effects and terminal outputs are then compared against human-written oracles using a rigorous multi-tiered equivalence metric.

Extensive evaluation of seven state-of-the-art LLMs reveals that the top-tier models achieve a maximum overall success rate of only 43.8%, highlighting that 0-to-1 software generation remains a highly challenging frontier. Furthermore, we discover that agents exhibit a strong tendency to generate monolithic code structures, and that higher token consumption does not necessarily yield better task performance.

I. INTRODUCTION

The advent of Large Language Models (LLMs) has catalyzed a paradigm shift in automated software engineering [1]–[4]. Moving beyond simple code completion, recent advancements have spurred the development of autonomous LLM-based agents [5]–[8] capable of tackling complex programming tasks. This rapid progress is driving the industry toward the era of “Vibe Coding” or intent-driven development [9]–[11]. In this new paradigm, users only need to express their high-level requirements in natural language. Autonomous agents are then expected to interpret these requirements and generate a complete, runnable software repository from scratch.

To measure the capabilities of these agents, robust evaluation benchmarks are indispensable. However, existing benchmarks fall short of evaluating the capabilities of agents for

real-world software creation. Traditional benchmarks, such as HumanEval [12] and MBPP [13], are confined to function-level generation. While recent efforts like SWE-bench [14] have elevated the evaluation to the repository level, they predominantly focus on software maintenance tasks rather than software creation from scratch. More recently, benchmarks like NL2Repo-Bench [15] have explored zero-to-one software generation. However, their evaluation methodologies reveal critical limitations that hinder the accurate assessment of modern LLM agents. Specifically, we identify two major challenges in the current evaluation landscape:

Challenge 1: Reliance on predefined repository structures. Evaluating the true 0-to-1 software generation capabilities of LLM agents remains an open challenge. Even in generation-focused benchmarks, the evaluation heavily relies on a fixed, predefined repository structure [15]. These benchmarks typically provide agents with pre-built file skeletons and directory scaffolds, thereby reducing the complex task of software generation to mere code-filling. This structure-dependent paradigm fundamentally bypasses the repository structure planning in software creation. In real-world software development, developers must decide how to organize directories, modularize files, and manage dependencies. By constraining agents to predefined structures, current benchmarks fail to assess whether LLMs can independently plan and construct a coherent repository.

Challenge 2: Absence of end-to-end black-box testing. Furthermore, existing evaluations heavily rely on white-box unit testing [12], [14]–[16]. These tests are tightly coupled with the internal implementation details of the software, forcing the generated code to conform to specific function signatures or class definitions. This approach severely limits the structural autonomy and creativity of LLMs. More importantly, from a user-centric perspective, software utilities like Command-Line Interface (CLI) tools are typically consumed as black boxes. Users care about whether the tool correctly parses command-line arguments, produces the expected terminal outputs, and produces the expected system-level side effects (e.g., file system modifications). Consequently, the reliance on rigid white-box testing fails to provide realistic and end-to-end validation of these critical external behaviors.

To bridge these gaps, we introduce CLI-Tool-Bench, a novel, structure-agnostic benchmark specifically designed to evaluate the 0-to-1 generation of CLI tools. We select CLI tools as our evaluation target because they represent fundamental software utilities. Their externally observable behaviors, including parsing standard inputs and producing terminal

¹ Harbin Institute of Technology, Shenzhen, China (email: {200111107, 200111115}@stu.hit.edu.cn, gaocuiyun@hit.edu.cn)

² Independent Researcher, China (email: chao.peng@acm.org)

³ Singapore Management University, Singapore (email: davidlo@smu.edu.sg)

[†] Corresponding author.

outputs or file system changes, make them well-suited for rigorous black-box testing without requiring internal code inspection. We curate a high-quality dataset of 94 real-world CLI repositories across three programming languages (i.e., Python, JavaScript, Go) and various complexity levels. To overcome the reliance on predefined structures, each agent is provided only with a natural language requirement and an empty workspace, forcing complete autonomy over internal structures and dependencies. We propose a novel evaluation method based on black-box differential testing. Specifically, the generated tool is executed in an isolated sandbox, and its terminal outputs alongside system-level side effects (e.g., file system changes) are rigorously compared against a human-written oracle. To ensure fair assessment without penalizing architectural diversity, we propose a multi-tiered equivalence metric encompassing Execution Reliability, Side-Effect Consistency, and Behavioral Equivalence.

In summary, the contributions of this paper are as follows:

- We introduce CLI-Tool-Bench, the first structure-agnostic benchmark for evaluating end-to-end software generation from scratch, which enforces true structural autonomy without reliance on predefined scaffolds.
- We propose an automated black-box differential testing pipeline that evaluates generated repositories in isolated sandboxes, rigorously verifying execution reliability, side-effect consistency, and behavioral equivalence.
- We conduct an extensive evaluation of seven state-of-the-art LLMs and agent frameworks on 94 real-world tasks. Our results reveal that the best-performing models achieve a maximum overall success rate of only 43.78%, underscoring the significant difficulty of this task. We also uncover critical behavioral patterns, such as the strong preferences of agents for monolithic designs and the prevalence of costly, unproductive debugging cycles.

II. CONSTRUCTION AND EVALUATION PIPELINE

To evaluate 0-to-1 software generation, we propose an automated pipeline (Figure 1) comprising three core modules: (1) Repository Curation, (2) Schema-Guided Task Synthesis, and (3) Black-Box Differential Evaluation.

A. Repository curation

To construct a representative dataset, we target three mainstream programming languages widely adopted for CLI development: Python, JavaScript, and Golang [17]. Through a multi-stage filtering process, we yield a final curated set of 94 high-quality oracle repositories.

1) *Static metadata filtering*: We initially retrieve candidate repositories from GitHub using specific criteria: (1) stars >10; (2) primary language proportion >60%; (3) presence of CLI-related keywords; (4) open-source license; and (5) the existence of language-specific build configuration files (e.g., “setup.py”, “package.json”, or “go.mod”).

2) *Dynamic entry identification*: However, not all retrieved repositories provide functional standalone CLI tools. To rigorously select valid repositories, we install each candidate globally in an isolated environment. A key challenge is to automatically identify the exact command name used to invoke the installed CLI, which frequently differs from the repository name. To address this, our pipeline monitors the system’s binary directory during installation to capture all newly generated executable files. We then compare the repository name against these captured files using a heuristic matching algorithm that prioritizes exact matches, acronyms, and string similarities. Once the most likely executable file is identified, it is validated by running the “--help” command. If this command successfully returns standard usage information without errors, the repository is considered to contain a functional CLI entry point.

3) *Stratified manual validation*: Repositories that pass this dynamic verification are further subjected to a stratified manual validation by two software engineering experts. This step ensures that the repositories can run without environmental errors. It also filters out tasks with implicit dependencies like non-standard local server environments to ensure fully unambiguous specifications. This rigorous filtering process ensures the high quality of the curated oracle repositories.

B. Schema-guided task synthesis

Generating comprehensive test cases at scale is a critical challenge. We address this by employing an LLM to extract a structured command schema from each curated oracle repository, which subsequently guides the automated generation of evaluation prompts and test suites.

1) *Iterative schema extraction*: For each oracle repository R_{oracle} , the LLM iteratively parses its README documentation and “--help” outputs. It explores nested subcommands level by level to extract a hierarchical metadata schema. This schema defines the external interface of the CLI, including available command names, subcommands, accepted parameters and their data types, and strict execution constraints such as mutually exclusive flags or required arguments.

2) *LLM-directed fuzzing for test generation*: Based on the extracted schema, we implement LLM-directed fuzzing to systematically cover the CLI’s capabilities. We first expand the schema into a set of distinct **Command patterns**. Each pattern represents a specific way to use the CLI, such as a unique subcommand paired with a valid combination of flags.

For each pattern, the LLM utilizes a predefined testing framework to generate diverse test instances covering common usage, boundary conditions, and error handling. The core of this process is an automated execution-feedback loop. Each generated test command is directly executed against the human-written oracle R_{oracle} . Crucially, the validity of these test cases is determined by the oracle itself. Any command that successfully executes on the ground-truth repository without errors is treated as a valid and supported usage. We then capture the oracle’s exact execution status, standard outputs, and any file system state changes. This combination of the

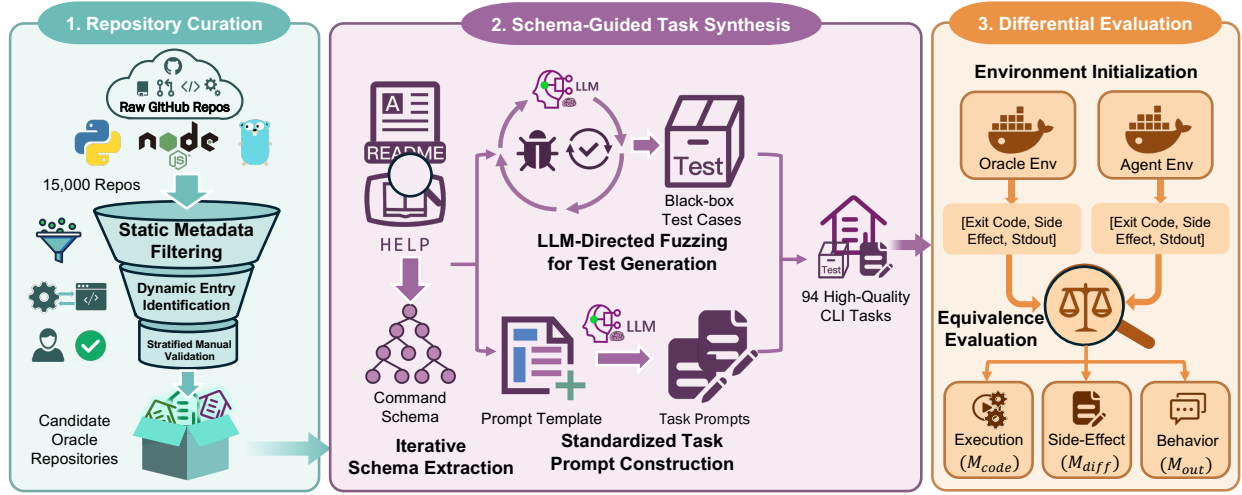


Fig. 1: Overview of the CLI-Tool-Bench framework.

input command and the captured oracle execution result forms a complete, ground-truth behavioral test case, effectively eliminating the need for manual test verification. If a generated command fails unexpectedly during this stage, the error trace is fed back to the LLM to refine the command until a valid test template is established.

Crucially, this fuzzing process acts as a final rigorous quality filter. If the LLM consistently fails to find a successful execution instance for any documented command pattern, the entire repository is discarded from our benchmark. Through this multi-stage filtering process, we ultimately yield our final dataset of 94 high-quality tasks. For the successfully verified command patterns in these remaining repositories, we synthesize exactly 50 end-to-end test cases to balance evaluation coverage and computational cost. These instances deliberately include valid positive inputs for normal execution paths and invalid or edge-case negative inputs, such as malformed formats, out-of-bounds values, or malicious strings. This design allows us to rigorously evaluate whether the generated software accurately reproduces the error handling behavior and robustness of the original oracle.

3) *Standardized task prompt construction*: The final evaluation prompt is designed to simulate a realistic and new software development requirements without leaking internal implementation details. The prompt comprises three anonymized components: (1) the sanitized functional context derived from the README; (2) the complete, iteratively extracted “--help” documentation, serving as the strict external interface specification; and (3) one verified successful execution example for each command pattern, which is obtained from the fuzzing stage and used to clearly demonstrate the expected behavior. To reduce the risk of data contamination during evaluation (e.g., models retrieving the original source code from the internet or their training corpora), the entire pipeline inherently embeds multi-layer defense mechanisms. Specifically, the evaluation is conducted in Docker containers without network access. During the prompt construction described

above, we systematically remove all identifying metadata such as author names and GitHub links. Furthermore, most of the 94 repositories selected in the curation phase were created after the major training cutoffs of the evaluated LLMs.

C. Black-box differential evaluation

To evaluate the agent-generated repository R_{test} without relying on predefined structural scaffolds or internal implementations, we design a black-box Differential Evaluation Engine based on isolated Docker sandboxes. The pipeline consists of three key phases:

1) *Environment initialization*: For each task, we instantiate two identical Docker containers for the corresponding programming language. We mount the frozen oracle repository R_{oracle} and the generated repository R_{test} into their respective containers. After executing standard package installation commands, we capture the initial state of the workspaces as clean snapshots. This snapshot-and-restore mechanism guarantees strict stateless isolation, avoiding file system pollution (persistent side effects) across sequential tests. We denote these restorable base environments as E_{oracle} and E_{test} .

2) *Differential execution*: For each test case t generated in the fuzzing stage, we execute it independently in both E_{oracle} and E_{test} . Our engine monitors the execution process and records the resulting state transition as a tuple $Exec(R, t, E) \rightarrow \langle C_{ret}, \Delta S, O_{std} \rangle$, where C_{ret} is the return code, ΔS represents the raw system-level side effects (e.g., file system modifications), and O_{std} is the standard output stream.

3) *Multi-tiered equivalence evaluation*: Based on the captured execution states, we evaluate functional equivalence using a strict progressive 3-step process. A test case is considered successful only if it passes all three consecutive checks:

1) **Execution reliability** (M_{code}): We strictly evaluate on valid functional paths where the oracle successfully executes without errors ($C_{ret}^{oracle} = 0$). We first verify if the agent-generated tool also completes successfully without throwing

unhandled exceptions ($C_{ret}^{test} = 0$). We denote the pass rate of this metric as **Exec**.

2) Side-effect pass (M_{diff}): Many CLI tools fundamentally operate by interacting with the file system. By comparing the workspace states before and after execution, we extract the effective file system changes $\Delta S'$ (filtering out trivial artifacts like hidden caches). A test case passes this metric, hereafter referred to as **SP**, only if it executes without errors and its effective side effects perfectly match those of the oracle ($\Delta S'_{oracle} = \Delta S'_{test}$). This prevents false positives where a tool runs successfully but performs no actual operations.

3) Behavioral equivalence (M_{out}): Only test cases that pass both the Exec and SP checks proceed to standard output comparison (O_{std}^{test} versus O_{std}^{oracle}). Because functionally correct CLI tools may inherently present their outputs in diverse formats or stylistic layouts, strict string matching is often insufficient. Therefore, we implement three complementary matching criteria (**EM**, **FM**, and **SM**) to ensure a comprehensive and fair evaluation across different semantic levels:

- **Exact Match (EM)** demands strict string equivalence after basic whitespace normalization.
- **Fuzzy Match (FM)** computes a normalized Levenshtein edit distance and records a match if the string similarity exceeds a threshold τ , thereby allowing minor formatting differences.
- **Semantic Match (SM)** employs an LLM as a judge (GPT-5.4) to verify whether the core informational payload is equivalent while disregarding stylistic differences. To validate this metric, a large-scale human annotation study on 1,000 output pairs yielded a Cohen’s Kappa coefficient of $\kappa > 0.9$, confirming strong alignment with expert judgment. Two annotators with at least 4-year software engineering experience independently labeled whether each pair was semantically equivalent from the perspective of CLI users, ignoring superficial formatting differences but requiring preservation of core information, error messages, and observable behavior. Disagreements were resolved through discussion.

To provide a clear overview of the constructed benchmark, Table I summarizes the statistical distribution of the final 94 curated tasks across three key dimensions: task difficulty, programming language, and application domain. We focus on Python, JavaScript (Node.js), and Go to provide a representative mix of interpreted and compiled languages extensively utilized in modern CLI development. Following the taxonomy established by NL2Repo-Bench [15], we stratify the task difficulty into three levels based on the oracle’s Lines of Code (Easy ≤ 1500 , Medium $1500 - 4000$, Hard ≥ 4000). We also classify the repositories into nine distinct real-world application scenarios. As shown in Table I, the final benchmark maintains a highly diverse and balanced composition, allowing the evaluation to reflect an agent’s general-purpose software generation capabilities.

TABLE I: Statistical distribution of our benchmark.

Dimension	Sub-category	Count	Avg. LOC
Difficulty	Easy (≤ 1500 LOC)	40	635.90
	Medium ($1500 - 4000$ LOC)	21	2,687.57
	Hard (≥ 4000 LOC)	33	18,509.97
Language	Python	34	4,473.50
	JavaScript	15	5,770.13
	Go	45	10,090.07
Domain	Web Development	8	5,550.50
	Testing	7	2,300.29
	Utility Libraries	20	8,567.65
	Machine Learning	12	3,334.92
	Data Analysis & Processing	11	2,887.27
	Database Interaction	6	3,509.17
	Networking Tools	7	6,384.14
	Batch File Processing	13	22,078.62
	System Tools	10	3,630.00

III. EXPERIMENTAL SETUP

A. Selected LLMs and agent frameworks

To comprehensively evaluate the state-of-the-art in autonomous software generation, we select 7 cutting-edge LLMs, encompassing both leading closed-source models and highly capable open-source models: GPT-5.4, Claude-Sonnet-4.6, DeepSeek-V3.2, Qwen-3.5-plus, GLM-5, MiniMax-M2.5, and Kimi-k2.5.

To effectively evaluate these models’ capabilities as software engineers, we employ two representative agent frameworks specifically designed for repository-level tasks:

- **OpenHands (with CodeAct)** [18]: A prominent open-source agent framework that utilizes the CodeAct paradigm, allowing the LLM to iteratively execute code, interact with a bash terminal, and observe environmental feedback.
- **Mini-SWE-Agent** [19]: A streamlined adaptation of the popular SWE-agent framework, specifically optimized for iterative repository construction and terminal-based debugging.

By evaluating seven models across two frameworks, we conduct a total of 14 distinct agent configurations for each of the 94 repositories in our benchmark.

Generation settings. To simulate a realistic autonomous software engineering scenario, we adopt an unconstrained, zero-external-feedback generation setting. For each repository, the agent is provided with the task prompt and an empty workspace. It is allowed an unrestricted number of conversational turns to freely explore, write, and execute code within its sandbox. The generation trajectory concludes entirely based on the agent’s own termination signal (e.g., executing a specific “exit” action), with no intermediate test feedback provided by our evaluation engine. This “single-trajectory” paradigm aligns seamlessly with our core objective of evaluating 0-to-1 software generation, where an agent is tasked to architect and deliver a complete, runnable tool from scratch in a single session.

B. Evaluation metrics and scoring mechanism

To systematically quantify the performance of an agent-generated repository R_{test} and populate our final evaluation tables, we compute final scores for the metrics defined in Section II-C.

First, we compute the **Build (Global Installation Success Rate)**, measuring whether R_{test} can be successfully installed using standard package managers. A repository failing this fundamental prerequisite is penalized with a score of zero across all subsequent evaluation metrics. This zero-tolerance penalty reflects the most basic requirement of the intent-driven “Vibe Coding” paradigm: if an agent fails to deliver a structurally valid and installable package, any downstream functional evaluation becomes meaningless. Thus, the Build metric serves as the primary indicator of an agent’s foundational repository planning and environment configuration capabilities.

For the successfully installed repositories, we employ a macro-averaging aggregation strategy to calculate the scores for the aforementioned metrics (Exec, SP, EM, FM, and SM) defined in Section II-C. This ensures that command patterns with simpler logic do not disproportionately dominate the overall evaluation. Specifically, let C denote the set of all identified command patterns for a given repository R_{test} . For each command pattern $c \in C$, let T_c be its corresponding suite of 50 test cases. We define the pass rate of a specific command pattern under a given metric $m \in \{\text{Exec, SP, EM, FM, SM}\}$ as:

$$P_m(c) = \frac{1}{|T_c|} \sum_{t \in T_c} \mathbb{I}_m(t) \quad (1)$$

where $\mathbb{I}_m(t) \in \{0, 1\}$ is the binary indicator of whether test case t successfully passes the specified metric check. Note that for behavioral metrics (EM, FM, SM), $\mathbb{I}_m(t) = 1$ strictly requires that the test case has already passed the prerequisite side-effect consistency gate.

The final aggregated score for the repository R_{test} under metric m is the unweighted average across all its command patterns:

$$\text{Score}_m(R_{test}) = \frac{1}{|C|} \sum_{c \in C} P_m(c) \quad (2)$$

The final values reported in our experimental results represent the average Score_m across all evaluated repositories.

C. Implementation details

To prevent data contamination, we explicitly disable network access within the agent sandboxes during generation, ensuring models cannot retrieve original repository implementations. Furthermore, we remove all artificial constraints—such as maximum iteration limits or token budgets—across all models and frameworks. The generation process is entirely open-ended, allowing the agent to autonomously determine when the software construction is complete and voluntarily terminate the execution. For all auxiliary tasks within our automated pipeline—including iterative schema extraction, LLM-directed fuzzing test generation, and the Semantic Match judge—we

strictly utilize GPT-5.4. To eliminate generation randomness and ensure highly deterministic evaluation outcomes, the temperature for these auxiliary GPT-5.4 API calls is uniformly set to zero. Moreover, τ is set to 0.8 in this work.

D. Research questions

We investigate the following four research questions (RQs):

- **RQ1 (Overall capabilities):** How do state-of-the-art LLMs perform in 0-to-1 CLI tool generation, and how do different agent frameworks influence their functional correctness?
- **RQ2 (Complexity impact):** How does the autonomous generation capability of these models scale or degrade across varying degrees of project complexity?
- **RQ3 (Generation efficiency):** What are the computational overheads and token consumption patterns of these agents under an unconstrained, open-ended generation setting?
- **RQ4 (Structural autonomy):** Given complete structural freedom, how do the repository-level designs of agent-generated software differ from human-written oracles?

IV. EXPERIMENTAL RESULT

In this section, we present the evaluation results to answer our four research questions.

Before diving into the detailed RQs, it is important to contextualize the scale and statistical reliability of our findings. Under the realistic single-trajectory setting (Section III-A), each of the seven evaluated LLMs independently generated the 94 repositories across two frameworks. Because our benchmark leverages LLM-directed fuzzing to synthesize exactly 50 end-to-end test cases per command pattern, a single generated repository yields hundreds of independent test instances. Consequently, the macro-averaged scores presented in the subsequent sections are derived from a massive distribution of tens of thousands of test executions per model.

To validate that the performance differences between models observed on this vast dataset are reliable and not just random chance, we employed McNemar’s test on the pass/fail outcomes, alongside Bootstrap resampling (10,000 iterations) to estimate the 95% Confidence Intervals (CI). Our baseline analyses confirm that the ranking of the top-tier models is statistically significant ($p < 0.05$). This proves that the differences in their aggregated scores reflect true variations in their coding capabilities, rather than sampling noise from the single generation runs, thereby establishing a rigorous foundation for the following research questions.

A. RQ1: Overall performance and framework impact

To answer RQ1, we evaluate seven state-of-the-art LLMs across two distinct agent frameworks. As detailed in Table II, the results reveal a clear performance hierarchy. Kimi-k2.5 emerges as the frontrunner, achieving the highest average Semantic Match (SM) score of 43.78%, followed by MiniMax-M2.5 (34.59% SM). A middle tier consists of GPT-5.4, Qwen-3.5-plus, and GLM-5, performing around 29% to 31% SM. Regarding framework impact, we observe a consistent trend:

TABLE II: Overall performance of the evaluated LLMs across two agent frameworks. The rightmost columns and bottom row present macro-averages. Best results are highlighted in **bold**.

Model	OpenHands						Mini-SWE-Agent						Average (Both Frameworks)					
	Build	Exec	SP	EM	FM	SM	Build	Exec	SP	EM	FM	SM	Build	Exec	SP	EM	FM	SM
GPT-5.4	79.79	59.66	38.83	21.47	38.55	28.88	85.11	61.05	51.90	23.85	40.74	31.82	82.45	60.35	45.36	22.66	39.65	30.35
Claude-Sonnet-4.6	37.23	18.42	12.34	5.95	9.36	7.21	52.13	31.67	28.15	11.77	21.03	14.02	44.68	25.04	20.24	8.86	15.20	10.61
DeepSeek-V3.2	80.85	61.82	41.53	21.48	37.72	27.44	72.34	60.08	50.67	21.25	40.33	27.41	76.60	60.95	46.10	21.37	39.02	27.43
Qwen-3.5-plus	73.40	58.96	42.71	23.50	38.34	28.86	76.60	61.34	49.38	24.15	39.99	30.75	75.00	60.15	46.05	23.83	39.16	29.81
GLM-5	76.60	59.03	40.05	22.07	37.43	28.54	78.72	63.47	53.44	26.69	43.97	33.23	77.66	61.25	46.75	24.38	40.70	30.89
MiniMax-M2.5	79.79	66.10	42.01	22.98	42.27	30.11	95.74	80.12	68.34	31.22	51.96	39.07	87.77	73.11	55.18	27.10	47.12	34.59
Kimi-k2.5	90.43	69.65	50.53	29.68	46.83	36.41	95.74	79.17	69.37	43.00	57.47	51.16	93.09	74.41	59.95	36.34	52.15	43.78
Average	74.01	56.23	38.29	21.02	35.79	26.78	79.48	62.41	53.04	25.99	42.21	32.50	76.75	59.32	45.66	23.50	39.00	29.64

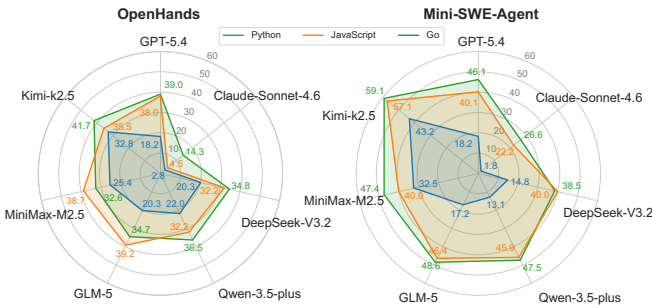


Fig. 2: Performance comparison of seven models across three programming languages. The radar charts illustrate the varying capabilities of each model, with the radial axes representing Semantic Match (SM) scores.

models generally achieve higher scores using Mini-SWE-Agent compared to OpenHands. This suggests that the action space and interface design of the agent framework substantially affect task completion in 0-to-1 generation scenarios.

Conversely, Claude-Sonnet-4.6 exhibits notably lower performance (averaging 10.61% SM), primarily bottlenecked at the initial Build stage. We note that our benchmark strictly provides no post-submission test feedback. In this setting, we observe a distinct behavioral pattern in Claude-Sonnet-4.6: it frequently concludes the generation trajectory immediately after writing the initial code. For instance, across multiple Go projects, the agent submits the repository without invoking standard compilation checks like `go build` or `go mod init`. Consequently, these repositories fail the initial build checks in our evaluation. In contrast, log analysis reveals that models like Kimi-k2.5 typically engage in more extensive in-container Bash validations before issuing the termination signal. This suggests that the performance gap at the Build stage is partially driven by differences in the models’ tendency to autonomously verify their code prior to submission.

Beyond individual model capabilities, our progressive evaluation funnel exposes fundamental bottlenecks in current autonomous workflows. Across all models, there is a steep degradation from the Build stage (average 76.75%) to Execution Reliability (59.32%), and a further drop at the Side-Effect

Pass (45.66%) gate. Ultimately, performance plunges when evaluated by Exact Match (EM , 23.50%). This massive drop demonstrates that while agents successfully write syntactically correct and executable code in many instances, they struggle to perfectly replicate the exact string outputs of the human oracle. However, the notable recovery observed in the FM score (39.00%) and SM score (29.64%) validates our methodological design: agents frequently generate functionally correct and semantically equivalent CLI tools that are unfairly penalized by stringent EM metrics.

A deeper analysis of the programming language distribution (Figure 2) further reveals a pronounced language bias inherent in current LLMs. As depicted in the radar charts, almost all evaluated models exhibit larger coverage areas for interpreted languages like Python and JavaScript, while Golang emerges as a severe bottleneck (represented by the constricted inner green polygons). It suggests that models frequently fail to navigate Golang’s strict type matching and rigid compilation constraints, often generating code that fails to build.

Furthermore, the choice of agent framework appears to influence generation outcomes. We observe that the streamlined, bash-centric Mini-SWE-Agent consistently achieves higher scores than the more complex OpenHands framework. This suggests that minimalist interfaces may naturally align better with terminal-centric workflows. A closer inspection of individual agent trajectories supports this observation. For instance, when attempting to resolve a missing library reference in the `mol-lang` repository, OpenHands agents frequently invoke browser tools, only to be flooded with multi-page HTML that overwhelms their context windows. Conversely, Mini-SWE-Agent directly reads the brief terminal `stderr`, executes the package manager, and successfully proceeds.

Finding 1: Despite the promising capabilities of top-tier models, their steep performance degradation across the evaluation funnel and severe struggles with compiled languages cap the highest overall success rate at 43.78%. This highlights that end-to-end, 0-to-1 software generation remains a highly challenging frontier.

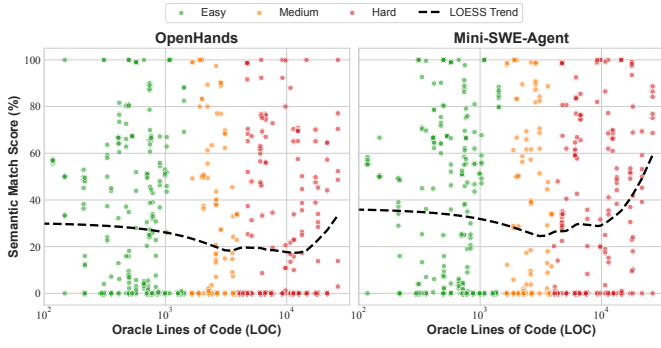


Fig. 3: Correlation between repository complexity and agent performance. Each scatter point represents an individual test case. The color of the points denotes the difficulty level: Green for Easy, Orange for Medium, and Red for Hard. The dashed black line represents the LOESS trend line.

B. RQ2: Impact of task complexity

To answer RQ2, we investigate how the software generation capability of these agents scales across varying degrees of project complexity. Figure 3 illustrates the relationship between the oracle repository’s size (measured in Lines of Code, LOC) and the corresponding Semantic Match (SM) scores. Intuitively, one might expect a strict negative correlation: as the repository grows larger, the agent’s performance should monotonically degrade due to expanding context windows and complex dependency resolutions. However, the LOESS trend lines reveal a non-monotonic (U-shaped) trajectory.

In the transition from Easy (<1500 LOC) to Medium (1500–4000 LOC) repositories, we observe a steep decline in performance. The overall average SM score drops from 32.66% to 25.08%, with competitive models like GPT-5.4 plummeting from 34.60% to 21.90%. To understand this degradation, we analyze the structural characteristics of these repositories. Our data reveal that while Easy tasks are predominantly flat, single-file scripts (averaging 1.62 files), Medium tasks demand multi-file, cross-directory modularization (averaging 2.33 files and 1.14 subdirectories). This structural leap appears to force the agent to handle complex relative imports and cross-file state management. The sharp drop in scores suggests that current LLMs struggle significantly when transitioning from localized coding to repository-level architectural planning.

Surprisingly, as the LOC scales into the Hard category (>4000 LOC), the average SM score rebounds to 28.87%. To unpack this counterintuitive phenomenon, we investigate the contextual richness associated with massive repositories. A quantitative analysis of our dataset reveals that the scale of a project strongly correlates with the comprehensiveness of its documentation: the average length of the README in Hard tasks (11,728 characters) is nearly three times that of Easy tasks (4,501 characters). In our intent-driven, 0-to-1 generation setting, this meticulously structured documentation provides a highly unambiguous specification. We hypothesize

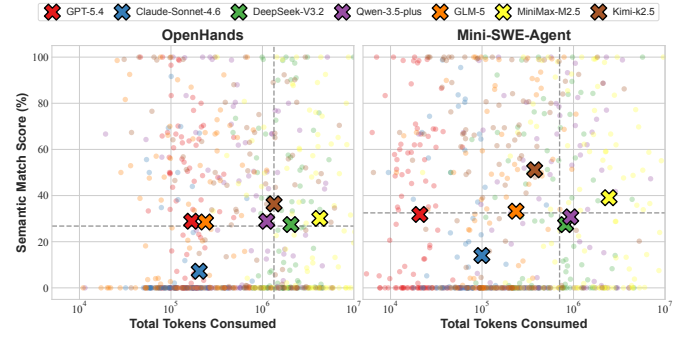


Fig. 4: Cost-effectiveness analysis of LLM agents across different frameworks. The scatter plots illustrate the trade-off between task performance and execution cost. Small translucent dots represent individual task runs, while the large cross markers denote the centroid for each model. The dashed gray lines indicate the overall mean score and mean token consumption, dividing the space into four quadrants.

that this “specification dividend” significantly offsets the raw implementation difficulty. While agents likely still struggle to perfectly architect the underlying logic of massive systems, the exceptionally rich and explicit documentation may enable them to more accurately extract and mirror the expected CLI command schemas, thereby achieving a higher SM score.

Finding 2: Agent performance exhibits a non-monotonic, U-shaped trend with repository complexity. The initial sharp decline highlights models’ severe struggles with multi-file architectural planning. However, the performance rebound in massive repositories suggests that highly comprehensive, unambiguous documentation significantly counteracts raw implementation difficulty, allowing agents to accurately reconstruct semantic schemas.

C. RQ3: Agent cost and generation Efficiency

To answer RQ3, we analyze the cost-effectiveness of different LLMs by plotting their Semantic Match Scores against their total token consumption, as shown in Figure 4. The dashed lines represent the average score and token usage, dividing the performance space into four quadrants. Ideally, a highly capable agent should fall into the top-left quadrant, achieving above-average scores with below-average token costs.

Across both frameworks, GPT-5.4 and GLM-5 consistently demonstrate this optimal behavior. GPT-5.4, in particular, emerges as a highly efficient engine. In the Mini-SWE-Agent framework, it achieves an average SM score of 31.82% while requiring only 2.33 interaction steps and consuming less than 21K tokens per task (costing a mere \$0.09). This suggests that GPT-5.4 possesses strong zero-shot reasoning capabilities, generating accurate CLI schemas without relying on extensive, token-heavy trial-and-error loops. Similarly, Kimi-k2.5 stands out as the overall best performer. It achieves the highest SM score (51.16%) with a moderate token footprint (roughly

TABLE III: Generation overhead and estimated API cost per repository, categorized by LLM and agent framework.

Model	Framework	Steps			Prompt Tokens (K)			Comp. Tokens (K)			Cost (\$)		
		Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
GPT-5.4	OpenHands	12.03	5	26	159.43	43.75	722.91	9.07	1.69	25.89	0.53	0.15	1.94
	Mini-SWE-Agent	2.33	2	6	17.68	2.81	82.45	3.21	0.30	8.26	0.09	0.01	0.26
Claude-Sonnet-4.6	OpenHands	7.48	1	34	196.74	31.84	1,917.41	7.09	0.12	81.78	0.70	0.10	6.15
	Mini-SWE-Agent	7.17	2	26	92.01	8.96	483.82	7.58	0.51	39.36	0.39	0.03	1.77
DeepSeek-V3.2	OpenHands	60.87	24	132	2,031.91	352.04	9,851.36	18.14	4.30	40.04	0.58	0.10	2.77
	Mini-SWE-Agent	39.64	14	112	802.97	43.56	5,626.74	15.66	1.82	51.70	0.23	0.01	1.60
Qwen-3.5-plus	OpenHands	34.03	2	130	1,099.83	1.23	5,599.58	14.20	2.08	47.24	0.47	0.03	2.35
	Mini-SWE-Agent	35.40	8	131	911.92	18.97	6,907.64	18.90	1.32	71.86	0.41	0.01	2.88
GLM-5	OpenHands	25.67	1	119	229.16	0.23	1,073.98	9.62	0.23	37.56	0.26	0.00	1.19
	Mini-SWE-Agent	39.19	2	116	221.77	9.45	1,005.59	13.48	0.98	57.60	0.26	0.01	1.19
MiniMax-M2.5	OpenHands	88.02	20	276	4,206.61	285.08	23,257.85	29.77	3.82	93.59	1.30	0.09	7.08
	Mini-SWE-Agent	68.48	10	153	2,412.56	21.37	9,503.50	33.32	1.04	105.40	0.76	0.01	2.93
Kimi-k2.5	OpenHands	47.38	4	130	1,317.52	24.92	6,844.40	23.40	1.54	238.72	0.86	0.03	4.33
	Mini-SWE-Agent	49.38	3	193	337.53	6.04	4,883.85	38.28	0.46	342.21	0.32	0.00	3.96

375.8K tokens and 49.38 steps) in the Mini-SWE-Agent framework, effectively dominating the top-left quadrant.

Conversely, the right half of the scatter plots reveals a phenomenon of “diminishing returns” in agent trajectories. Models like Minimax-M2.5 and DeepSeek-V3.2 frequently fall into the right-side quadrants. For example, within OpenHands, Minimax-M2.5 consumes a staggering average of 4.21 million tokens and engages in 88.02 steps per task, yet fails to achieve top-tier performance (scoring only 30.11% SM). This extreme token consumption and interaction count typically indicate “thrashing”—situations where the agent becomes trapped in repetitive debugging cycles or generates overly verbose, unhelpful commands without making actual progress toward task resolution. On the other extreme, Claude-Sonnet-4.6 consistently occupies the bottom-left quadrant. Its remarkably low token usage (averaging roughly 92K tokens in Mini-SWE-Agent and 197K in OpenHands) and minimal interaction count (averaging 7.58 and 7.09 steps, respectively), coupled with the lowest semantic scores, imply a distinct “fail fast” tendency. As previously observed, this behavior likely stems from Claude’s lack of extensive self-verification and debugging loops; it tends to produce an initial solution and terminate the trajectory prematurely, rather than engaging in the iterative troubleshooting necessary for 0-to-1 software construction.

Finding 3: Higher token consumption does not equate to better task resolution. While certain models experience diminishing returns due to costly debugging loops, efficient models like GPT-5.4 achieve highly competitive performance with minimal tokens and interaction steps, demonstrating superior cost-effectiveness.

D. RQ4: Structural Autonomy and Diversity

To answer RQ4, we investigate how different LLMs organize repository structures and manage workspace complexity when granted full structural autonomy. Figure 5 compares the



Fig. 5: Distribution of total files generated by the human Oracle and various LLMs. The box plots, combined with overlaid strip plots, show the median and quartile distributions while highlighting individual task runs as semi-transparent dots. The y-axis is truncated at 20 files. Extreme outliers are annotated with red stars and their maximum values at the top.

file count distributions of agent-generated repositories against the human Oracle.

Human Modularity vs. Agent Monolithic Preference. The evident trend is the divergence in structural design between human developers and autonomous agents. The human Oracle exhibits a broad distribution, reflecting standard software engineering practices where code is modularized into distinct components. In contrast, all evaluated LLMs demonstrate a strong preference for monolithic structures, with their medians tightly clustered between 1 and 3 files. This indicates a shared behavioral strategy among LLMs: centralizing logic into a single or very few files. From an agentic perspective, this monolithic approach seems to be a practical adaptation to minimize cross-file dependency issues (e.g., ImportError) and to keep the entire system state easily accessible within the model’s limited context window.

Workspace Management and Debugging Behaviors. Beyond the median values, the maximum file counts (annotated with red stars) reveal diverse workspace management behaviors during the generation process. Models like GPT-5.4 and Claude-Sonnet-4.6 maintain strictly low file counts across all tasks (maximums of 16 and 10, respectively). Conversely, several other models (e.g., DeepSeek-V3.2, Qwen-3.5-plus, and MiniMax-M2.5) occasionally exhibit extreme file sprawl, generating hundreds or even thousands of files (e.g., up to 1,629 files). Upon inspecting these repositories, we found that this extreme sprawl does not represent manually written source code, but rather the inclusion of massive local dependency directories (e.g., `node_modules` or local virtual environments) directly within the workspace. This indicates that while some models carefully manage dependencies globally or outside the target directory, others resort to heavy local installations during their execution. Furthermore, this behavior is heavily influenced by the environment. For instance, GLM-5 and Kimi-k2.5 maintain compact workspaces under OpenHands but exhibit severe dependency sprawl under Mini-SWE-Agent, suggesting that the underlying framework’s prompt structure and feedback mechanisms impact agents’ file system management. Interestingly, we also observe that a notable proportion of cases generate exactly zero valid files¹, particularly within the high-freedom OpenHands framework.

Finding 4: While human developers naturally adopt modular structures, LLMs prefer monolithic designs (typically 1-3 files) to simplify context management. Furthermore, models exhibit vastly different workspace management strategies, ranging from strict in-place editing to massive local dependency sprawl, a behavior that is also highly sensitive to the chosen framework.

V. DISCUSSION

A. Case Study: Phase-based Failure Analysis

Phase 1: Build failures (23.3% of trajectories). Nearly a quarter of all runs fail to produce a structurally installable package. Figure 6 (a) demonstrates a prevalent “build failure” driven by a *text-to-file bias*. In this Go project scenario, the agent generated the correct logic but attempted to manually hardcode dependencies into `go.mod` instead of executing standard commands like `go mod tidy`. Lacking “environmental intuition,” agents default to static text manipulation rather than interacting with dynamic toolchains.

Phase 2: Execution failures (13.2% of trajectories). Among repositories that install successfully, 13.2% crash entirely during execution. Figure 6 (b) illustrates a catastrophic workspace mismanagement scenario. While attempting to build a Python tool, the agent trapped itself in a recursive directory generation loop, creating deeply nested `build/lib/` structures. This chaotic file system state broke the package’s

entry points, highlighting a critical spatial blindness when diagnosing corrupted environments.

Phase 3: Behavioral mismatch (6.7% of trajectories). Perhaps the most deceptive failure mode occurs when a tool executes flawlessly (Exit Code 0) but produces no system-level side effects (SM=0). As shown in Figure 6 (c), the agent-generated `deletor` tool ran without crashing but failed to remove the target files. This silent failure firmly validates the necessity of our multi-dimensional evaluation approach, proving that “running without crashing” does not equate to task completion.

B. Implications of findings

Researchers: Our research demonstrates that evaluating autonomous agents requires moving beyond static code analysis. Specifically:

- **Evaluation paradigms.** The significant gap between exact match, semantic match, and execution success proves that traditional string-based evaluations are insufficient for agentic tasks. End-to-end behavioral validation must become the new standard. Therefore, it is imperative to propose novel equivalence metrics and methods that can accurately capture actual software behaviors.
- **Agent architecture.** The prevalent issues of “thrashing” and catastrophic workspace mismanagement highlight a structural gap. Future research must equip agents with better spatial awareness of the file system and self-reflection mechanisms to break out of unproductive debugging loops.

Developers: Agentic software generation is a new paradigm that introduces unique challenges in code structure and quality. Based on our findings, we conclude the following insights and takeaways for developers using LLM agents:

- **Structural scaffolding.** Since agents universally prefer monolithic designs and struggle with complex dependency resolutions, developers should provide explicit structural scaffolding or mandate the use of standardized CLI frameworks (e.g., `Click` or `Cobra`) to guide generation.
- **Defensive programming.** While agent-generated tools often execute faster than human baselines due to their stripped-down, goal-oriented coding style, they severely lack defensive programming practices. Developers must treat these outputs as functional prototypes, actively reinforcing them with robust error handling.
- **Tailored “vibe coding”.** Developers must adopt tailored interaction strategies: mandate explicit self-testing steps for rapid-delivery models (e.g., Claude), while proactively monitoring verification-driven agents (e.g., Kimi) to prevent infinite loops.

C. Threats to validity

- **Dataset Scale and Selection.** Our dataset consists of 94 curated task instances. This scale represents a deliberate trade-off: we prioritized a rigorously cleansed and manually inspected dataset over a larger, automated collection to ensure high-quality and unambiguous task specifications.

¹These “zero-file” cases primarily result from unauthorized workspace relocation (agents writing to hallucinated absolute paths) or premature termination due to safety filters. Detailed case analyses are available in our repository.

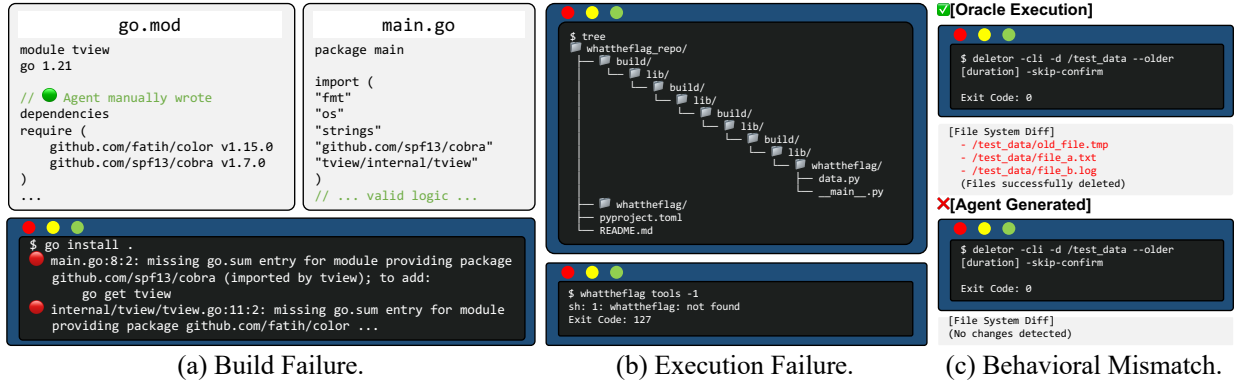


Fig. 6: Representative failure cases of LLM agents in CLI tool development.

- **Execution reliability criteria.** Regarding the execution reliability metric (M_{code}), our pipeline strictly treats a non-zero exit code from the generated tool as a failure if the oracle executes successfully with a zero exit code. In practice, a non-zero exit code might not always signify a complete runtime crash, but rather a custom warning or specific state. However, such behavior violates common POSIX design principles [20] where successful operations should return zero. Enforcing this strict alignment provides the objective and standardized baseline for differential testing.
- **Evaluation setting.** Our evaluation employs an unconstrained, zero-external-feedback setting where agents deliver the final repository in a single trajectory. While incorporating external test feedback (e.g., feeding error logs back to the agent) to enable multi-turn iterative debugging is a highly promising direction for Automated Program Repair, it falls beyond the scope of this paper, which focuses on evaluating the baseline 0-to-1 construction capability. Future iterations will aim to scale the dataset and explore these feedback-driven generation paradigms.

VI. RELATED WORK

A. Agents for Software Engineering

The application of LLMs in software engineering has evolved from static code completion to autonomous, environment-interacting agents [5], [6], [21], [22]. Early multi-agent frameworks, such as ChatDev [23] and MetaGPT [24], utilized role-playing and Standardized Operating Procedures (SOPs) to orchestrate software design via simulated collaboration. To enable real-world repository interaction, Agent-Computer Interfaces (ACIs) were introduced. Frameworks like SWE-agent [25] and OpenHands [18] allow LLMs to interact directly with command-line terminals, execute tests, and iteratively debug within isolated sandboxes. Recently, minimalist scaffolds like Mini-SWE-agent [19] have demonstrated that state-of-the-art models require remarkably little scaffolding to achieve high success rates, relying primarily on their native reasoning capabilities.

B. Software Generation and Agent Evaluation

As agentic capabilities advance, evaluation methodologies have shifted to complex systems. Early benchmarks like HumanEval [12] and MBPP [13] focused on static, function-level synthesis but suffer from data contamination and a lack of repository context. Subsequent efforts addressed this through dynamic problem sourcing (LiveCodeBench [26]) or complex API integration (BigCodeBench [27]). For system-level tasks, SWE-bench [14] established the standard for software maintenance by evaluating patch generation for GitHub issues, while OSWorld [28] and Terminal-Bench [29] assess multi-step command execution in live operating systems. However, evaluating zero-to-one software generation remains a critical challenge. Recent works like Commit0 [16] track an agent’s ability to rebuild a repository commit-by-commit. While it scales to larger repositories, Commit0 provides pre-existing skeleton code and evaluates correctness through rigid static analysis and predefined unit tests. NL2Repo-Bench [15] explores repository-level generation from natural language, yet it also fundamentally relies on white-box unit tests. This rigid approach severely penalizes functionally correct but structurally diverse solutions. In parallel, Terminal-Bench [29] and OSWorld [28] evaluate multi-step bash command execution within live operating systems. While they test agent interactions with the terminal (e.g., navigating directories or installing packages), they neither focus on compiling a deliverable software entity from scratch nor strictly verify the resulting system-level file side-effects. In contrast, CLI-Tool-Bench bridges these gaps by requiring full repository construction from an empty workspace and employing dynamic, black-box differential testing to rigorously validate both system side-effects and output equivalence.

CLI-Tool-Bench bridges these paradigms, extending black-box differential testing to the repository level to rigorously evaluate end-to-end software generation without imposing structural constraints.

VII. CONCLUSION

We introduce CLI-Tool-Bench, a novel benchmark evaluating end-to-end software generation from scratch. Extensive

evaluations reveal a maximum success rate of just 43.8%, highlighting substantial room for improvement. Furthermore, our analysis uncovers key insights: performance exhibits a U-shaped trend with complexity, higher token consumption does not guarantee better results, and agents universally favor monolithic designs over modular ones to simplify context management. Currently, we focus on CLIs. However, our core evaluation methodology of structure-agnostic, black-box differential testing is highly generalizable. In future work, we plan to extend this methodology to broader software domains, such as web services, API backends, and GUI applications, further advancing the assessment of fully autonomous software creation.

VIII. DATA AVAILABILITY

All scripts, datasets, and raw data are available at <https://github.com/kinesiaticsscxi14/CLI-Tool-Bench>.

REFERENCES

- [1] C. Arora, J. Grundy, and M. Abdelrazek, "Advancing requirements engineering through generative ai: Assessing the role of llms," in *Generative AI for Effective Software Development*. Springer, 2024, pp. 129–148.
- [2] Z. Mao, J. Li, D. Jin, M. Li, and K. Tei, "Multi-role consensus through llms discussions for vulnerability detection," in *2024 IEEE 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*. IEEE, 2024, pp. 1318–1319.
- [3] L. Chen, Q. Guo, H. Jia, Z. Zeng, X. Wang, Y. Xu, J. Wu, Y. Wang, Q. Gao, J. Wang, W. Ye, and S. Zhang, "A survey on evaluating large language models in code generation tasks," *CoRR*, vol. abs/2408.16498, 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2408.16498>
- [4] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *CoRR*, vol. abs/2406.00515, 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2406.00515>
- [5] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2025, pp. 2188–2200.
- [6] D. Chen, S. Lin, M. Zeng, D. Zan, J. Wang, A. Cheshkov, J. Sun, H. Yu, G. Dong, A. Aliev, J. Wang, X. Cheng, G. Liang, Y. Ma, P. Bian, T. Xie, and Q. Wang, "Coder: Issue resolving with multi-agent and task graphs," *CoRR*, vol. abs/2406.01304, 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2406.01304>
- [7] J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou, "Large language model-based agents for software engineering: A survey," *CoRR*, vol. abs/2409.02977, 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2409.02977>
- [8] Z. Wang, Z. Liu, Y. Zhang, A. Zhong, J. Wang, F. Yin, L. Fan, L. Wu, and Q. Wen, "Rcagent: Cloud root cause analysis by autonomous agents with tool-augmented large language models," in *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, 2024, pp. 4966–4974.
- [9] P. P. Ray, "A review on vibe coding: Fundamentals, state-of-the-art, challenges and future directions," *Authorea Preprints*, 2025.
- [10] S. H. Maes, "The gotchas of ai coding and vibe coding. it's all about support and maintenance," *OSF Preprints*, 2025.
- [11] C. Meske, T. Hermanns, E. Von der Weiden, K.-U. Loser, and T. Berger, "Vibe coding as a reconfiguration of intent mediation in software development: Definition, implications, and research agenda," *IEEE Access*, vol. 13, pp. 213 242–213 259, 2025.
- [12] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: <https://arxiv.org/abs/2107.03374>
- [13] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *CoRR*, vol. abs/2108.07732, 2021. [Online]. Available: <https://arxiv.org/abs/2108.07732>
- [14] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. [Online]. Available: <https://openreview.net/forum?id=VTF8yNQm66>
- [15] J. Ding, S. Long, C. Pu, H. Zhou, H. Gao, X. Gao, C. He, Y. Hou, F. Hu, Z. Li, W. Shi, Z. Wang, D. Zan, C. Zhang, X. Zhang, Q. Chen, X. Cheng, B. Deng, Q. Gu, K. Hua, J. Lin, P. Liu, M. Li, X. Pan, Z. Peng, Y. Qin, Y. Shan, Z. Tan, W. Xie, Z. Wang, Y. Yuan, J. Zhang, E. Zhao, Y. Zhao, H. Zhu, C. Zou, M. Ding, J. Jiao, J. Liu, M. Liu, Q. Liu, C. Tao, J. Yang, T. Yang, Z. Zhang, X. Chen, W. Huang, and G. Zhang, "Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents," *CoRR*, vol. abs/2512.12730, 2025. [Online]. Available: <https://doi.org/10.48550/arXiv.2512.12730>
- [16] W. Zhao, N. Jiang, C. Lee, J. T. Chiu, C. Cardie, M. Gallé, and A. M. Rush, "Commit0: Library generation from scratch," in *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. [Online]. Available: <https://openreview.net/forum?id=MMwaQEVsAg>
- [17] W. Li, A. Marino, H. Yang, N. Meng, L. Li, and H. Cai, "How are multilingual systems constructed: Characterizing language use and selection in open-source multilingual software," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 3, pp. 1–46, 2024.
- [18] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, and et al., "Openhands: An open platform for AI software developers as generalist agents," in *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. [Online]. Available: <https://openreview.net/forum?id=OJd3ayDDoF>
- [19] SWE-Agent, "The 100 line AI agent that solves GitHub issues or helps you in your command line." <https://github.com/SWE-agent>.
- [20] T. Group et al., "The open group base specifications issue 7," 2018.
- [21] S. Qiao, N. Zhang, R. Fang, Y. Luo, W. Zhou, Y. E. Jiang, C. Lv, and H. Chen, "Autoact: Automatic agent learning from scratch for QA via self-planning," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 3003–3021. [Online]. Available: <https://aclanthology.org/2024.acl-long.165>
- [22] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, M. Christakis and M. Pradel, Eds. ACM, 2024, pp. 1592–1604. [Online]. Available: <https://doi.org/10.1145/3650212.3680384>
- [23] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, and M. Sun, "Chatdev: Communicative agents for software development," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 15 174–15 186. [Online]. Available: <https://doi.org/10.18653/v1/2024.acl-long.810>
- [24] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "Metagpt: Meta programming for A multi-agent collaborative framework," in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. [Online]. Available: <https://openreview.net/forum?id=VtmBAGCN7o>
- [25] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated

software engineering,” in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024.

- [26] N. Jain, K. Han, A. Gu, W. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, “Livecodebench: Holistic and contamination free evaluation of large language models for code,” in *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. [Online]. Available: <https://openreview.net/forum?id=chfJJYC3iL>
- [27] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, S. Brunner, C. Gong, J. Hoang, A. R. Zebaze, X. Hong, W. Li, J. Kaddour, M. Xu, Z. Zhang, P. Yadav, and et al., “Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions,” in *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025.
- [28] T. Xie, D. Zhang, J. Chen, X. Li, S. Zhao, R. Cao, T. J. Hua, Z. Cheng, D. Shin, F. Lei, Y. Liu, Y. Xu, S. Zhou, S. Savarese, C. Xiong, V. Zhong, and T. Yu, “Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments,” in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024.
- [29] M. A. Merrill, A. G. Shaw, N. Carlini, B. Li, H. Raj, I. Bercovich, L. Shi, J. Y. Shin, T. Walshe, E. K. Buchanan *et al.*, “Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces,” *arXiv preprint arXiv:2601.11868*, 2026.