E2EDev: Benchmarking Large Language Models in End-to-End Software Development Task

Anonymous authors
Paper under double-blind review

Abstract

The rapid advancement in large language models (LLMs) has demonstrated significant potential in End-to-End Software Development (E2ESD). However, existing E2ESD benchmarks are limited by coarse-grained requirement specifications and unreliable evaluation protocols, hindering a true understanding of current framework capabilities. To address these limitations, we present E2EDev, a novel benchmark grounded in the principles of Behavior-Driven Development (BDD), which evaluates the capabilities of E2ESD frameworks by assessing whether the generated software meets user needs through mimicking real user interactions (Figure 1). E2EDev comprises (i) a fine-grained set of user requirements, (ii) multiple BDD test scenarios with corresponding Python step implementations for each requirement, and (iii) a fully automated testing pipeline built on the Behave framework. To ensure its quality while reducing the annotation effort, E2EDev leverages our proposed Human-in-the-Loop Multi-Agent Annotation Framework (HITL-MAA). By evaluating various E2ESD frameworks and LLM backbones with E2EDev, our analysis reveals a persistent struggle to effectively solve these tasks, underscoring the critical need for more effective and cost-efficient E2ESD solutions. Our codebase and benchmark are publicly available at `https://anonymous.4open.science/r/E2EDev`.

Figure 1: Overview of E2EDev: a dataset and BDD-based automated evaluation pipeline for E2ESD tasks.

1 Introduction

The effectiveness of large language models (LLMs) in understanding user needs and demonstrating logical reasoning has been validated in code generation (Jimenez et al., 2024; Gao et al., 2024). LLMs can generate code snippets based on provided code context and user query (Yu et al., 2024; Chen et al., 2021; Hui et al., 2024), and debug code through feedback from either human or system (Zhong et al., 2024; Chen et al., 2023; Li et al., 2022). The success in generating isolated functions is fueling further research into advanced software development automation. This has led to a shift from function-level code synthesis to **End-to-End Software Development (E2ESD)**, where complete

Figure 2: Comparing existing work with ours, existing benchmarks use coarse-grained requirements with unreliable evaluations whereas E2ESD provides fine-grained requirements with executable BDD tests.

software is automatically generated from user requirements (e.g., *'generate an HTML game of Angry Birds'*). By this means, E2ESD could enhance both the code effectiveness and efficiency by removing the manual assembly bottleneck. Current LLM-based E2ESD frameworks can be broadly categorized into multi-agent approaches, inspired by traditional software engineering principles (Dong et al., 2024; Hong et al., 2024; Du et al., 2024; Royce, 1987; Qian et al., 2024), and single-agent approaches, where a single LLM manages the complete development pipeline (Engineer, 2024; AI, 2024).

As LLM-driven approaches advance, there is growing demand for robust benchmarks to automate the evaluation of E2ESD frameworks (Hong et al., 2024; Qian et al., 2024; He et al., 2024; Hu et al., 2025). However, existing E2ESD benchmarks, such as SoftwareDev (Hong et al., 2024) and SRDD (Qian et al., 2024), suffer from two critical limitations. (1) **Coarse-grained Requirement Specifications**: Current benchmarks substitute user requirements with ambiguous software descriptions as input, making it difficult to verify whether the generated software aligns with actual user needs. As shown in the top panel of Figure 2, a vague software requirement for Facebook like "sharing with friends" could be interpreted in multiple ways, such as forwarding content, sharing links, or enabling visibility of posts. Without precise operational specifications of such requirements, systematic testing becomes impractical. (2) **Unreliable Evaluation Protocols**: Evaluations in these benchmarks rely heavily on labor-intensive human assessments (Hong et al., 2024) and lack standardized evaluation methodologies grounded in established software engineering principles (Qian et al., 2024). This results in inconsistent and unreliable performance comparisons across frameworks.

To this end, we introduce the **E2EDev** benchmark for evaluating the performance of LLM-based frameworks on E2ESD tasks, along with a human-in-the-loop annotation framework designed to ease the annotation burden. Specifically, E2EDev is derived from real-world open-source web application projects. Following the principles of Behavior-Driven Development (BDD), which specifies and validates software behavior from a user perspective, E2EDev evaluates whether the generated software meets user requirements by mimicking real user interactions. As illustrated in the bottom panel of Figure 2, E2EDev consists of: (1) a fine-grained list of user requirements for each software, (2) for each user requirement, we provide multiple BDD test scenarios to verify its correctness, each corresponding to an executable Python code implementation, and (3) an automated testing pipeline built on Behave[1]. To alleviate the annotation burden while ensuring data quality, we propose the Human-in-the-Loop Multi-Agent Annotation Framework (HITL-MAA). In this framework, specialized agents analyze the project source code to generate candidate requirements and executable tests, with human supervisors involved at key points to avoid errors.

Our comprehensive evaluation of various E2ESD frameworks and LLM backbones reveals a significant struggle to effectively solve E2ESD tasks. Current frameworks, even with GPT-4o, exhibit a lack of proficiency in handling detailed functional specifics during implementation, with top performance falling significantly below

[1]Behave is a Python framework for behavior-driven development using natural language tests (Gherkin). See `https://github.com/behave/behave` for details.

Table 1: Comparison of Representative Benchmarks for LLM-based Software Development

Benchmark	Level	# Tasks	# Reqs	With Tests	Source	Notes
HumanEval (Chen et al., 2021)	Function	164	164	✓	Human-Written	Python only
MBPP (Austin et al., 2021)	Function	974	974	✓	Human-Written	Crowd-sourced
APPS (Hendrycks et al., 2021)	Function	10k+	10k+	✓	Real-World	Competitive programming
BigCodeBench (Zhuo et al., 2024)	Function	1140	1140	✓	Real-World	Designed for complex instructions and tool usage
SoftwareDev (Hong et al., 2024)	Project	70	-	✗ (Human-rated)	Real-World	Private benchmark
SRDD (Qian et al., 2024)	Project	1,200	-	✗ (Heuristic)	LLM-Generated	LLM-generated software description
rSDE-Bench (Hu et al., 2025)	Project	53	-	✓ (#Unit Tests = 616)	LLM-Generated	Function-level evaluation with unit tests
E2EDev	Project	46	**244**	✓ (# BDD Test Scenarios = **703**)	Real-World	Fine-grained user requirements Project-level evaluation using BDD tests

60%. Moreover, multi-agent architectures often exhibits excessive interaction rounds and token costs, yielding only minimal gains in effectiveness. Our in-depth error analysis further reveals that the primary challenge lies in accurately implementing functionality with precise details, a challenge amplified using multi-agent architectures. Finally, we believe our benchmark stands out as a valuable resource, highlighting the critical need in designing more effective and cost-efficient E2ESD frameworks.

2 Related Work

Benchmarks for Software Development. Various benchmarks have been proposed to evaluate the capabilities of LLM-based methods in software development. Most focus on function-level tasks (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021; Zhuo et al., 2024)(upper part of Table 1), while few address project-level E2ESD (lower part). Specifically, SoftwareDev (Hong et al., 2024) relies on human ratings to assess requirement satisfaction across 70 samples, while SRDD (Qian et al., 2024) employs heuristic metrics such as Completeness (absence of unfinished or placeholder code), Executability (compilation and runtime success), and Consistency (semantic similarity between requirements and code). However, these project-level benchmarks face two main limitations. First, the input specifications are often generated by LLMs or presented as high-level textual descriptions (Qian et al., 2024), which are typically vague or underspecified. This makes it difficult to verify whether the generated software truly satisfies the intended requirements. Second, the absence of standardized evaluation methods hinders objective and reliable performance comparison across different frameworks. While rSDE-Bench (Hu et al., 2025) proposes $\sim$600 test cases for project-level coding task evaluation, its reliance on Unit Tests limits assessment to function-level evaluation, overlooking the overall behavior of LLM-generated projects and failing to verify actual user needs. Appendix E details the difference between Unit Tests and our BDD Test approach.

LLM-based Frameworks for End-to-End Software Development. Recent advances primarily fall into two categories: multi-agent and single-agent frameworks. The majority utilize a multi-agent paradigm, breaking down the development process into subtasks based on established software engineering models (Sommerville, 2011; Hong et al., 2024; Du et al., 2024; Rasheed et al., 2024; Sami et al., 2024). Within this category, Self-Collaboration (Dong et al., 2024) mimics the fundamental development stages—requirement analysis, implementation, and testing—via specialized agents. MapCoder (Islam et al., 2024) enhances software generation by incorporating prior project knowledge. In addition, while most frameworks use predefined workflows, some explore dynamic agent interactions (Qian et al., 2024; Lin et al., 2024), such as ChatDev (Qian et al., 2024), which uses multi-turn dialogues for iterative refinement. Conversely, a smaller body of work pursues a single-agent approach (Engineer, 2024), assuming a single LLM can independently handle the entire development cycle without explicit task decomposition.

3 E2EDev Benchmark Construction

Overview. E2EDev is constructed by transforming real-world software projects into fine-grained user requirements and corresponding executable tests, using our HITL-MAA framework for efficient annotation and validation. Each requirement is associated with a set of **BDD test scenarios** written in Gherkin[2]. Each scenario is accompanied by a corresponding **step implementation** in Python, which specifies how to execute each step for automated evaluation using the Behave framework. Table 2 summarizes dataset statistics, and example entries are provided in Appendix A.3.

Table 2: Benchmark Statistics

		Max	Min	Mean	Total
Project	Task Prompt Length	1907	206	712.74	**46**
Requirement	Reqs per Project	11	2	5.30	**244**
	# Words per Req.	112.76	20	64.10	
Test Case	Cases per Req.	7	1	2.88	**703**
	Cases per Project	34	2	15.28	
	Gherkin Test Scenarios				
	# Lines per Scenario	30	7	11.17	
	# Words per Scenario	360	53	109.02	
	Test Step Definitions				
	# Lines per Step Definition	157	13	61.41	
	# Words per Step Definition	700	74	217.44	

[2]Gherkin is a structured language using Given-When-Then statements to describe software behavior.

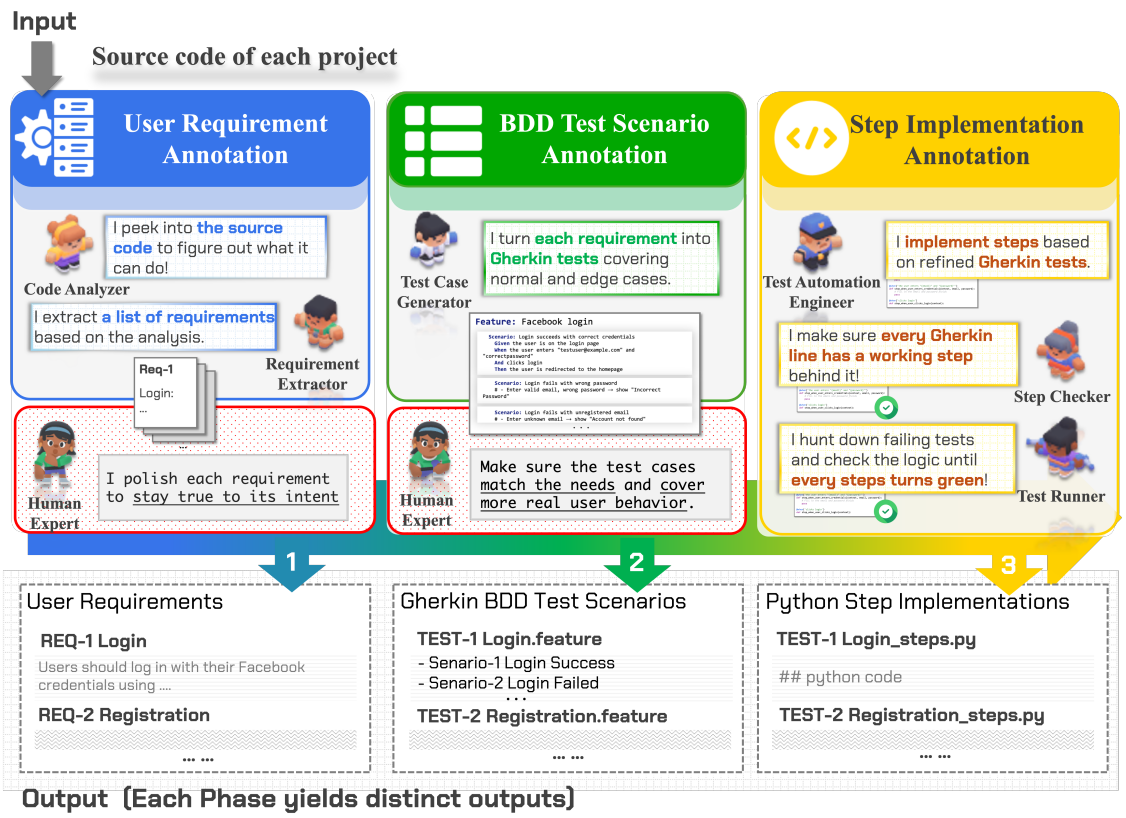

Figure 3: HITL-MAA framework for semi-automated dataset construction. Given source code, the framework extracts user requirements and generates corresponding BDD test scenarios (Gherkin format) along with Python step implementations, with human verification or agent-based refinement at each stage to ensure dataset quality.

3.1 Project Source and Selection

To ensure that benchmark data represents realistic and diverse development scenarios while remaining manageable for LLM-based methods, we focus specifically on Web applications as our project source. Web applications strike a balance between complexity and accessibility: they involve rich development aspects such as front-end logic, API communication, and state management, yet typically avoid the need for complex compilation or specialized runtime environments, allowing consistent execution and evaluation (Brown, 2019; Gustafson, 2013; Li et al., 2014).

To ensure the realism and quality of our benchmark, we follow a two-step process for selecting source projects. First, we crawl GitHub repositories using keywords such as “mini project”, “software project”, “web application”, and “H5”, retaining only those with over 500 stars to ensure sufficient community endorsement and visibility. Next, we manually verify each selected project across three key dimensions: (1) *executability* (runs without major errors in Chrome); (2) *functionality* (core components behave as intended); and (3) *suitability* (all core functionalities can be fully experienced within approximately five minutes, thus maintaining manageable complexity). Finally, this comprehensive selection process results in a total of 46 projects. More detailed information regarding the manual verification process can be found in Appendix A.1.

3.2 Human-in-the-Loop Multi-Agent Annotation Framework (HITL-MAA)

Grounded in BDD-based software engineering practices, HITL-MAA is employed for each collected project to annotate fine-grained user requirements, along with corresponding BDD test scenarios and Python step implementations, to ensure reliable evaluation. As illustrated in Figure 3, HITL-MAA involves the following three steps, with human supervisors involved at key points in each step to avoid errors. Before generating requirements and test cases, we use GPT-4o to assign unique test IDs to key UI components. These test IDs serve as stable, structure-invariant DOM anchors, enabling consistent component references across requirements, BDD tests, and different projects generated from the same requirement. All implementation details are provided in Appendix A.2.

User Requirement Annotation. Given the source code of a project, this process involves the collaborative effort of two LLM agents, cooperating with human supervisors: (1) The Code Analyzer Agent analyzes the UI structure and JavaScript logic of the source code to summarize key functionalities and their interactions with UI elements. (2) The Requirement Extractor Agent then automatically generates candidate user-facing requirements based on

the summarized analysis. During this process, human supervisors review the generated requirements to ensure accuracy, clarity, and functional consistency through UI inspection or browser testing. In practice, LLM-generated requirements often fail to capture hidden preconditions for software features (e.g., navigation buttons that only appear at smaller screen sizes), which highlights the value of human corrections to prevent the creation of invalid test cases and misleading executable tests later on. Finally, for each data entry in E2EDev, we create an overview summary based on the validated requirements. This provides a description of the project, designed to improve the LLM's comprehension during our benchmark evaluation.

BDD Test Scenario Annotation. For each human-validated requirement, HITL-MAA generates a corresponding set of Gherkin-style BDD test scenarios, where each scenario describes a specific condition or user interaction to be tested. Scenarios follow the Given-When-Then format, which clearly lays out the preconditions (*Given*), the user action (*When*), and the expected outcome (*Then*) in natural language. To support this process, HITL-MAA employs a Test Case Generation Agent that analyzes the validated requirements and source code to generate relevant scenarios. The agent processes requirements one by one and is prompted to cover a wide range of user behaviors—including typical user actions, unexpected actions, boundary cases, and defensive handling—based on user intent. Since the agent relies on source code, it may miss certain behavior patterns that are not explicitly represented. For example, a function like `submitForm()` might manage different types of invalid input internally, but if these differences are not clear from the source code, the agent might not generate test scenarios covering every case. To improve coverage, five software testing experts collaboratively review and refine the generated scenarios. They supplement missing interactions, eliminate hallucinated cases, and ensure each scenario is consistent with the requirement. To further ensure annotation quality, HITL-MAA includes a final expert review step to resolve any potential inconsistencies between the requirements and test cases.

Step Implementation Annotation. This step focuses on converting each BDD test scenario into an executable Python script. To achieve this, HITL-MAA utilizes a Test Automation Engineer Agent to generate initial Python step implementations corresponding to the Gherkin-style test steps. The generated scripts then undergo an iterative verification process to ensure both executability and correctness. Two LLM agents provide feedback to support self-correction: the *Dry Run Verifier* checks for missing or syntactically invalid step definitions, while the *Test Runner* executes the script to detect logical inconsistencies or failed assertions. Scripts unresolved after maximum self-correction attempts are flagged for human refinement. Empirically, this self-correction mechanism resolves nearly all dry-run errors autonomously and corrects logical issues in over 80% of cases without human intervention. Validation on the generated dataset achieved a 100% pass rate across all executable tests, demonstrating the high quality and reliability of E2EDev.

3.3 Features & Applications of E2EDev Benchmark

Features of E2EDev. E2EDev consists of diverse software development tasks evenly distributed across multiple common types of web applications and a wide range of common user interaction patterns, detailed in Figure 8 in Appendix A.4. The most prevalent types include: *Mathematics & Conversion Tools*, such as calculators for tax; *Account & Form Management*, covering simple CRUD systems where users submit forms and manage records; and *Mini Games & Interactive Entertainment* like Tic-Tac-Toe. Additionally, E2EDev captures both basic and advanced user interactions, such as clicking, typing, sliders, and drag-and-drop operations. It also includes technically complex features like DOM manipulation, local storage usage, external API integration, and precise mathematical function generation. Taken together, the diversity in application types, interaction patterns, and technical complexity demonstrates that E2EDev provides a representative and realistic benchmark for end-to-end web application development.

Automated Testing Using E2EDev. To enable automated evaluation, we offer a benchmark suite designed for researchers regardless of their familiarity with software testing procedures[3].

4 Experiment

4.1 Experiment Setup

Baseline & LLM Backbones. We select a diverse set of representative E2ESD frameworks across various LLM backbones. These include: (1) Vanilla LLM that generates the full codebase via simply prompting LLM using user requirements without external scaffolding. (2) Single-Agent Framework, *GPT-Engineer* (Engineer, 2024), which builds the complete codebase in a single reasoning pass with minimal external coordination while maintaining a modular development workflow. (3) Multi-Agent Frameworks, a dominant paradigm, where the development process is decomposed into specialized agent roles based on classic software engineering principles (Sommerville, 2011). We benchmark with a wide range of state-of-the-art multi-agent framework, including *Self-Collaboration (Dong et al., 2024)*, *MetaGPT (Hong et al., 2024)*, *MapCoder (Islam et al., 2024)*, and *ChatDev (Qian et al., 2024)*. Finally, we

[3]Cf. Appendix D for background knowledge of project/software testing.

Table 3: Benchmark analysis across various LLM backbones regarding the effectiveness and efficiency. Vanilla LLM is highlighted in light grey as a reference. For effectiveness, the highest effective metric value within each group is shown in **bold**, and the second highest is underlined.

Backbone LLM	Method	Effectiveness (↑, %)			Efficiency (↓)		
		Req. Acc.	Test Acc	Balance Score	Cost (USD)	Footprint (gCO_2eq)	Duration (s)
GPT-4o	Vanilla LLM	46.23	61.96	52.52	0.0160	0.083	28
	GPT-Engineer	**50.37**	**66.68**	**56.90**	0.0198	0.132	21
	Self-Collaboration	46.14	60.58	51.92	0.0155	0.109	37
	MapCoder	47.92	63.89	54.31	0.1091	0.750	93
	ChatDev	42.75	58.08	48.88	0.1947	1.910	114
	MetaGPT	0.00	0.18	0.07	0.0951	0.794	66
GPT-4o-mini	Vanilla LLM	**45.72**	**62.37**	**52.38**	0.0010	0.003	16
	GPT-Engineer	43.10	58.45	49.24	0.0012	0.005	18
	Self-Collaboration	38.76	53.98	44.85	0.0009	0.004	25
	MapCoder	40.89	57.41	47.50	0.0072	0.033	88
	ChatDev	33.95	49.85	40.31	0.0118	0.071	157
	MetaGPT	0.00	0.27	0.11	0.0067	0.040	63
Qwen-Max	Vanilla LLM	43.35	58.81	49.54	0.0025	0.043	53
	GPT-Engineer	**49.79**	64.13	**55.52**	0.0030	0.067	66
	Self-Collaboration	42.68	59.22	49.30	0.0022	0.053	75
	MapCoder	48.37	**64.25**	54.72	0.0186	0.424	366
	ChatDev	43.07	58.95	49.43	0.0249	0.790	300
	MetaGPT	1.63	2.84	2.11	0.0207	0.544	312
Qwen-70B	Vanilla LLM	35.75	53.14	42.71	0.0029	0.050	39
	GPT-Engineer	42.08	**58.78**	48.76	0.0037	0.080	45
	Self-Collaboration	42.57	55.57	47.77	0.0030	0.066	69
	MapCoder	40.44	57.53	47.28	0.0293	0.624	294
	ChatDev	**43.52**	57.67	**49.18**	0.0387	1.006	341
	MetaGPT	0.00	0.17	0.07	0.0290	0.786	151
Qwen-7B	Vanilla LLM	22.37	34.75	27.32	0.0003	0.005	32
	GPT-Engineer	**24.03**	**39.94**	**30.39**	0.0003	0.007	31
	Self-Collaboration	20.58	33.55	25.76	0.0003	0.006	52
	MapCoder	12.83	27.65	18.76	0.0024	0.049	236
	ChatDev	11.30	20.43	14.95	0.0075	0.187	344
	MetaGPT	0.00	1.83	0.43	0.0207	0.095	301

adopt a range of backbones from two vendors (Hurst et al., 2024; Hui et al., 2024), covering various scales and architectures: GPT-4o, GPT-4o-mini, Qwen-7B (Qwen2.5-7B-Instruct), Qwen-70B (Qwen2.5-72B-Instruct), and the mixture-of-experts (MoE) Qwen-Max (Qwen2.5-Max). This enables controlled analysis of framework versus model backbone effects. Implementation details are presented in Appendix B.1.

Evaluation Metrics. We propose a systematic evaluation protocol for E2ESD, involving both **code effectiveness** (*how well the generated code meets user requirements*) and **generation efficiency** (*how is the cost and time of the code generation process*). As for code effectiveness, the evaluation metrics include: (1) Req. Acc, the proportion of requirements successfully validated by all their corresponding test cases across a project, which measures the requirement-level accuracy; (2) Test Acc, the proportion of all passed test cases across a project, which measures the test-level accuracy; (3) Balanced Score, a weighted combination of *Req. Acc* and *Test Acc* to mitigate bias from varying test case granularity per requirement. Additionally, generation efficiency is assessed using the following three metrics: (1) Cost (USD), the average API pricing for LLM token usage; (2) Carbon Footprint (CO_2), the average carbon footprint for generating one project; (3) Duration, the average wall-clock time for project generation. Metric details are presented in Appendix B.2.

4.2 Main Results

Table 3 presents our benchmark analysis on off-the-shelf methods across various LLM backbones regarding their effectiveness and efficiency. Our key observations are detailed below.

How effective are existing methods on E2ESD?

Despite meeting broad project requirements, performance remains below acceptable standards due to lack of proficiency in handling specifics. Off-the-shelf models excel at function-level tasks (e.g., HumanEval (Chen et al., 2021) in Appendix C.3) but underperform on project-level E2ESD, achieving only 30%–50% Req. Acc. on the proposed E2EDev dataset (Table 3), with even GPT-4o below 60%. Furthermore, Test Acc. scores, which allow partial fulfillment, suggest failures stem from their lack of proficiency in handling detailed functional specifics during implementation. This phenomenon, visualized in Figure 4, persists across different LLM backbones and E2ESD frameworks. The observed discrepancy of over 25% in both metrics suggests that while models can

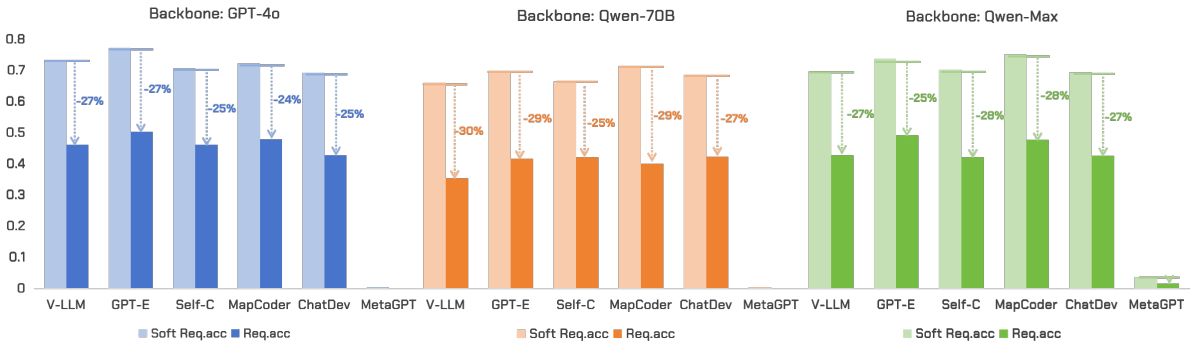

Figure 4: Soft Req. Acc. and Req. Acc. under three representative LLMs. Here, V-LLM refers to the Vanilla LLM, GPT-E refers to GPT-Engineer, and Self-C refers to Self-Collaboration. Additional results on other backbones are reported in Appendix C.1, Table 9.

implement the required functionality, they often fail to address complex edge cases. Importantly, this issue is also reflected in ∼10% performance gap observed on HumanEval and HumanEval-ET (cf. Appendix C.3).

How efficient are existing methods on E2ESD?

Streamlined workflows offer high efficiency, but multi-agent frameworks often exhibit excessive interaction rounds and token costs with minimal effectiveness gains. As evidenced by the efficiency-oriented metrics in Table 3, methods such as Vanilla LLM, GPT-Engineer, and Self-Collaboration maintain high efficiency ($\leq$ three times API Calls), benefiting from their streamlined, deterministic workflows. However, regarding the multi-agent frameworks, even static frameworks like MapCoder and MetaGPT require >10 interaction turns per task, amplifying latency and computational expense. Dynamic frameworks (e.g., ChatDev with GPT-4o) reach 15.72 turns on average (minimum of 9), with inefficiency exacerbated by repetitive and uninformative dialogue cycles, as analyzed in detail in Appendix C.2.

Figure 5: Comparisons across agentic frameworks and the Vanilla LLM (on Req. Acc.). Bars within the gray zone indicate lower performance.

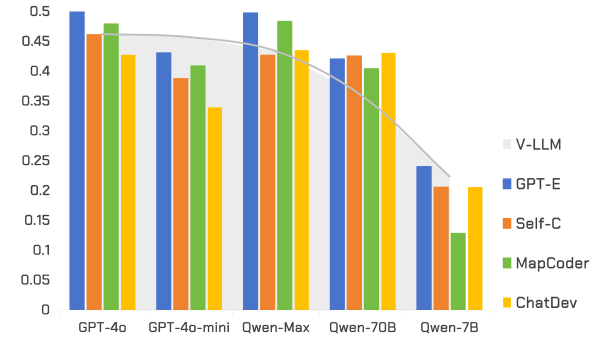

Table 4: Statistics of interaction overhead across representative frameworks. *P. Tok.*, *C. Tok.*, and *Turns* refer to average prompt tokens, completion tokens, and dialogue rounds, respectively.

Model	Method	P.Tok	C.Tok	Turns
GPT-4o	MapCoder	16,835.54	6,696.46	10.00
	ChatDev	53,912.26	5,995.65	15.72
	MetaGPT	20,501.28	4,387.00	10.46
Qwen-Max	MapCoder	19,427.28	9,410.52	10.00
	ChatDev	46,192.96	7,458.87	12.50
	MetaGPT	28,272.87	8,724.35	11.48
Qwen-70B	MapCoder	22,906.81	10,447.55	10.00
	ChatDev	45,167.91	8,647.67	11.57
	MetaGPT	36,496.70	5,568.80	13.83

Does the integration of agentic frameworks consistently enhance the effectiveness of the Vanilla LLM?

Not necessarily. While agentic frameworks offer the potential for performance gains, their inherent complexity imposes significant demands on the foundational capabilities of the underlying LLM. Figure 5 shows framework performance varies widely across LLMs, with Vanilla LLMs occasionally surpassing frameworks (and vice versa), revealing architectural overhead. Notably, multi-agent frameworks often underperform single-agent approaches due to the necessary coordination demands among agents, such as task decomposition (Xia et al., 2024; Han et al., 2024), role/instruction following (Pan et al., 2025; Hammond et al., 2025), dialogue state/context management (Pan et al., 2025; Li et al., 2023; Hammond et al., 2025), and the increasing interaction costs as shown in Table 4. Therefore, error accumulation from these complexities can degrade performance, particularly in E2ESD tasks. We posit that current frameworks tend to rely heavily on LLMs with robust and versatile foundational capabilities. This suggests that future framework designs should prioritize minimizing this reliance to enhance their practical utility and robustness across diverse LLM backbones.

Why is the performance of MetaGPT notably low on E2ESD?

It suffers from communication breakdowns within its specialized multi-agent architecture. While confirming MetaGPT's reported function-level performance on HumanEval using the author's code (cf. Appendix C.3), MetaGPT fails to handle nearly all test cases and requirements at the project level, even when using powerful

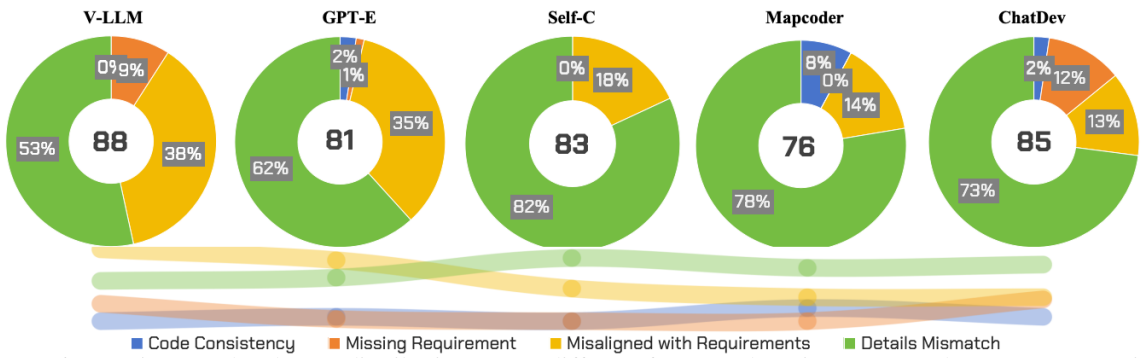

Figure 6: Requirement-level error distribution across different frameworks. Pie segments show error proportions; the center shows total errors across three representative LLM backbones. Trend lines below show the absolute values of each specific error across different frameworks.

LLMs like GPT-4o or Qwen-Max. This issue seems to be communication breakdowns within this multi-agent framework, ultimately undermining its efficacy (further discussions in Section 4.3).

4.3 Understanding the Limitations of Existing Frameworks on E2ESD

Setup. To gain deeper insights, we conduct a rigorous human evaluation. To balance analytical depth with the practical constraints of human effort, we randomly select 10 data entries from our dataset, resulting in 300 generated projects. Four domain experts then interact with these generated projects, guided by the corresponding user requirements, and evaluate across four key aspects detailed in Table 5 (see Appendix B.3 for evaluation protocol). Finally, Figure 6 visualizes the failure mode distributions of existing methods that employ representative LLMs as backbones — known to be effective in agentic frameworks — grouped by framework type, with detailed results and case studies for each framework provided in the Appendix C.1 and Appendix C.4, respectively.

Table 5: Classification of errors in LLM-generated projects

Error Type	Description
Code Inconsistency	Internal inconsistencies in generated code, such as *unimplemented functions*, *conflicting or duplicated implementations*, *empty functions*, or *signature mismatches*.
Requirement Missing	Required functionality is not implemented.
Requirement Misaligned	Logic of implemented functionality deviates from requirements.
Detail Mismatch	Functionality logic aligns with requirements, but contains inaccurate details.

Analysis of Code Inconsistency

While Vanilla LLMs maintain high code consistency, integrating these LLMs with the E2ESD framework may compromise this consistency, particularly within multi-agent frameworks. This is likely due to the potential exposure to excessive or irrelevant information within the context and prompt. For example, MapCoder employs analogical prompting (Yasunaga et al., 2024), in which the LLM first retrieves project exemplars from its own memory (i.e., generate exemplars by itself), and then leverages them for few-shot enhancement before tackling the given project. Although this approach is intended to provide richer contextual support, our analysis reveals that these retrieved projects often have only a loose relationship to the current task, sharing a general domain (e.g., web applications) but lacking significant functional similarity. This extraneous information introduces noise rather than clarity. In contrast, frameworks that restrict the coder agent's input to task-specific requirements and component analyses minimize interference, resulting in more consistent code generation.

Analysis of Requirement Missing

Flawed multi-agent architecture design may contribute to overlooked requirements. As shown in Figure 6, *Vanilla LLM* and *ChatDev*, the only frameworks failing to incorporate requirement analysis during code generation[4], exhibit severe Missing Requirement issue. While ChatDev assigns an agent for requirement analysis, its architecture limits inter-agent communication to pairwise interactions. Consequently, its coding agent interacts solely with the Executive Officer, receiving only high-level structural guidance (e.g., HTML structure, JavaScript functionality, CSS styling) and lacking detailed requirement specifications. This limited visibility significantly elevates the risk of overlooking critical requirements. Thus, ensuring coding agents have access to comprehensive requirement analysis is essential for complete software solutions.

[4]GPT-Engineer implicitly performs requirement analysis before code generation via "think step by step".

Analysis of Requirement Misaligned

Core component analysis helps align code with user requirements. Figure 6 shows multi-agent frameworks significantly reduce Requirement Misaligned issues compared to others. This improvement likely stems from explicitly integrating core component analysis—specifically, the identification of key structural and functional elements such as HTML layout, JavaScript logic (script.js), and CSS styles (styles.css). This analysis informs the coding agent, allowing a focus on user-driven logic and structure dictated by user requirements. Consequently, the generated code better reflects intended functionality and aligns with user requirements.

Analysis of Detail Mismatch

LLMs inherently struggle with fine-grained detail control, a limitation amplified by the additional context introduced by multi-agent frameworks. Trend lines in Figure 6 reveal that detail mismatches are prevalent across all frameworks, with multi-agent frameworks moderately increasing their frequency—Vanilla LLM produces the fewest (47), while Self-Collaboration generates the most (68). To gain more sights, we categorize detail mismatches into three types: (1) *unhandled logical edge cases*, where interdependent features lead to state synchronization failures; (2) *missing error handling or validation logic*; and (3) *UI component display inconsistencies*. In particular, logical edge cases remain particularly challenging; for instance, in a dictionary-based favorite-word feature, models consistently fail to update the "favorited" status after removing a word. Missing error handling typically results from absent conditional branches for exceptional cases, and multi-agent frameworks often overlooks these despite explicit instructions in the original prompt—likely due to omission during requirement analysis or component planning. To mitigate this, frameworks like MapCoder and ChatDev prepend each agent's input with both the original user requirements and the previous agent's message, preserving contextual fidelity and reducing omissions. Finally, UI display mismatches, such as incorrect decimal precision or text rendering, may largely attribute to LLMs' intrinsic tendencies, and persist even under strict prompting.

Detailed Analysis on MetaGPT

Communication breakdowns within its multi-agent architecture significantly impair code consistency and requirement fidelity. As seen in Figure 7, 44% of all failures stem from code consistency issues, like missing files and syntax errors. Notably, over 43% of these are caused by the programmer agent ignoring the architect's file structure, despite explicit constraints. Additionally, 54% of syntax errors involve malformed imports or unmatched tokens, often resulting from the engineer agent being tasked with both code generation and tool invocation—prompt instructions for the latter outweigh the former by over 10×, introducing noise that disrupts code generation. Furthermore, 25% of failures are due to missing requirements, with another 31% stemming from inconsistent or incomplete implementation. This may stem from the product manager frequently rewriting or compressing the original user requirement, despite being instructed to restate it verbatim. We observe similar trends in HumanEval: MetaGPT with GPT-4o only achieves ∼50% pass rate, with ∼30% of completions failing to match required function names. This suggests that an over-engineered agent workflow degrades both adherence to specification and coding reliability.

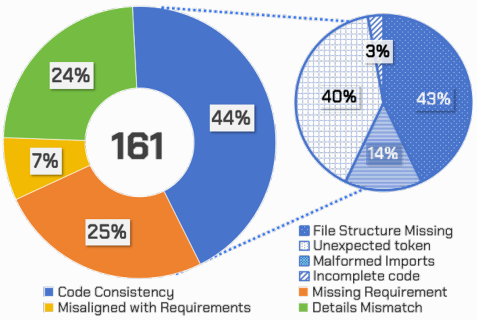

Figure 7: Error distribution of MetaGPT

5 Conclusion & Discussion

Unlike function-level tasks, project-level coding tasks offer a more comprehensive assessment of LLMs' code generation capabilities. However, the inherent complexity of constructing robust project-level benchmarks presents significant challenges. A primary hurdle arises from the diverse and often unique code implementations LLMs produce for identical user requirements, which substantially increases the difficulty and cost associated with developing reliable automated evaluation mechanisms. Consequently, current project-level benchmarks, including the one presented in this paper, often feature limited dataset sizes (as detailed in Table 1).

Despite these challenges, our current work represents a crucial and reasonable starting point toward more rigorous project-level task evaluation. Our E2EDev marks a pivotal advancement in standardizing the benchmark of E2ESD frameworks by providing detailed requirement specifications and reliable evaluation protocols. While LLMs have demonstrated promise in code generation and function-level software development, our work highlights the significant challenges remaining in achieving fully automated E2ESD. Looking ahead, future work should focus more on designing effective, reliable, and cost-efficient E2ESD solutions that can truly realize the potential of LLMs in automating software development. Additionally, acknowledging the substantial resource investment required for such benchmark development, we are committed to continuously expanding our dataset and establishing a public leaderboard. This sustained effort is dedicated to fostering ongoing advancements in the application of LLMs for software development, pushing the boundaries of what is currently achievable.

6 ETHICS STATEMENT

Our work focuses solely on constructing the E2EDev benchmark for end-to-end software development tasks to evaluate the capabilities of LLM-based E2ESD frameworks. The study does not involve human subjects, sensitive personal data, or other ethical risks, and all source data are obtained from open-source communities. Any human annotation or verification of data was performed under controlled conditions; details are provided in the Appendix A. All experiments and analyses adhere to the ICLR Code of Ethics, and no procedures, datasets, or methods in this study raise concerns related to privacy, bias, or safety.

7 REPRODUCIBILITY STATEMENT

To facilitate reproducibility, we publicly release the full E2EDev benchmark source code on GitHub (`https://anonymous.4open.science/r/E2EDev`), along with the complete benchmark dataset on Hugging Face (`https://huggingface.co/1059-datasets/anon-submission2025/E2EDev`). Detailed setup and usage instructions are provided in both Appendix D.3 and the repository README. All experiments reported in this paper—including dataset processing, model evaluation, and metric computation—can be fully reproduced using these resources.

REFERENCES

Asma Ben Abacha, Wen-wai Yim, Yujuan Fu, Zhaoyi Sun, Meliha Yetisgen, Fei Xia, and Thomas Lin. Medec: A benchmark for medical error detection and correction in clinical notes. *arXiv preprint arXiv:2412.19260*, 2024.

Cognition AI. Introducing devin, the first ai software engineer, 2024. URL https://cognition.ai/blog/introducing-devin.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Ethan Brown. *Web development with node and express: leveraging the JavaScript stack*. O'Reilly Media, 2019.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schaerli, and Denny Zhou. Teaching large language models to self-debug. In *The 61st Annual Meeting Of The Association For Computational Linguistics*, 2023.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–38, 2024.

Zhuoyun Du, Chen Qian, Wei Liu, Zihao Xie, Yifei Wang, Yufan Dang, Weize Chen, and Cheng Yang. Multi-agent software development through cross-team collaboration. *CoRR*, 2024.

GPT Engineer. Gpt engineer, 2024. URL https://github.com/gpt-engineer-org/gpt-engineer. Accessed: 2024-12-27.

Ahmad Faiz, Sotaro Kaneda, Ruhan Wang, Rita Osi, Prateek Sharma, Fan Chen, and Lei Jiang. Llmcarbon: Modeling the end-to-end carbon footprint of large language models. *arXiv preprint arXiv:2309.14393*, 2023.

Cuiyun Gao, Xing Hu, Shan Gao, Xin Xia, and Zhi Jin. The current challenges of software engineering in the era of large language models. *ACM Transactions on Software Engineering and Methodology*, 2024.

JM Gustafson. Html5 web application development by example beginner's guide, 2013.

Lewis Hammond, Alan Chan, Jesse Clifton, Jason Hoelscher-Obermaier, Akbir Khan, Euan McLean, Chandler Smith, Wolfram Barfuss, Jakob Foerster, Tomáš Gavenčiak, et al. Multi-agent risks from advanced ai. *arXiv preprint arXiv:2502.14143*, 2025.

Shanshan Han, Qifan Zhang, Yuhang Yao, Weizhao Jin, Zhaozhuo Xu, and Chaoyang He. Llm multi-agent systems: Challenges and open problems. *arXiv preprint arXiv:2402.03578*, 2024.

Junda He, Christoph Treude, and David Lo. Llm-based multi-agent systems for software engineering: Literature review, vision and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 2024.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *ICLR*, 2024.

Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. Self-evolving multi-agent collaboration networks for software development. 2025.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4912–4944, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *ICLR*, 2024.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36: 51991–52008, 2023.

Yuan-Fang Li, Paramjit K Das, and David L Dowe. Two decades of web application testing—a survey of recent advances. *Information Systems*, 43:20–54, 2014.

Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1035–1047, 2022.

Leilei Lin, Yingming Zhou, Wenlong Chen, and Chen Qian. Think-on-process: Dynamic process generation for collaborative development of multi-agent system. *arXiv preprint arXiv:2409.06568*, 2024.

Melissa Z Pan, Mert Cemri, Lakshya A Agrawal, Shuyi Yang, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Kannan Ramchandran, Dan Klein, et al. Why do multiagent systems fail? In *ICLR 2025 Workshop on Building Trust in Language Models and Applications*, 2025.

David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, 2024.

Zeeshan Rasheed, Muhammad Waseem, Mika Saari, Kari Systä, and Pekka Abrahamsson. Codepori: Large scale model for autonomous software development by using multi-agents. *arXiv e-prints*, pp. arXiv–2402, 2024.

Winston W Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, pp. 328–338, 1987.

Malik Abdul Sami, Muhammad Waseem, Zeeshan Rasheed, Mika Saari, Kari Systä, and Pekka Abrahamsson. Experimenting with multi-agent software development: Towards a unified platform. *arXiv preprint arXiv:2406.05381*, 2024.

Ian Sommerville. Software engineering (ed.). *America: Pearson Education Inc*, 2011.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Pasupat, Jure Leskovec, Percy Liang, Ed H Chi, and Denny Zhou. Large language models as analogical reasoners. In *The Twelfth International Conference on Learning Representations*, 2024.

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

Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics ACL 2024*, pp. 851–870, 2024.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *CoRR*, 2024.

A DETAILS ON E2EDEV BENCHMARK

A.1 DETAILS ON HUMAN VERIFICATION FOR PROJECT SELECTION

To ensure the realism, functionality, and suitability of each project in our benchmark, we conducted a structured manual verification process. In particular, each candidate project was independently reviewed by *five annotators*, all holding at least a bachelor's degree in computer science or a related technical field, with formal training in software development. Before annotation, annotators were informed of the task's complexity, estimated time cost, and confirmed that the task posed no safety risks. All annotators followed a standardized instruction set across three key dimensions: Executability, Functionality, and Suitability, detailed below.

- Executability. Each project must run without major runtime errors under the following standard conditions. Notably, projects were tested locally using lightweight HTTP servers (e.g., `http-server`, Python's `SimpleHTTPServer`). Projects that failed to render the main UI or blocked basic interactions were excluded.
  - The application must launch and run correctly in the latest stable version of *Google Chrome*.
  - No deprecated or non-standard dependencies should block execution.
  - All static assets (HTML, CSS, JS) must load properly.
  - The browser console should be free of critical errors or warnings affecting core functionality.
- Functionality. Annotators manually interacted with each component to ensure that it behaved as expected. Examples include the follows. Importantly, this step ensures that all included projects demonstrate reliable and observable behavior, making them suitable for LLM-based evaluation.
  - Correct handling of inputs and operators in calculators.
  - Valid drag-and-drop functionality in task managers.
  - Functional game logic (e.g., win/loss detection in Tic-Tac-Toe).
  - Visual feedback elements such as animations or theme switching.
- Suitability. Projects must be feasible for evaluation by LLMs and human annotators within approximately *five minutes*. Projects were excluded if:
  - They required extensive setup (e.g., database configuration, API keys). This was verified by inspecting the source code.
  - Their main functionality was overly complex or time-consuming to test in practice.

Verification Protocol & Inter-Annotator Agreement. To refine and validate our annotation guidelines, we first conducted a pilot annotation round. We randomly selected 10 projects and asked all five expert annotators—each with formal training in software development—to independently label them based on the three key dimensions: Executability, Functionality, and Suitability. Each project was assigned a binary label (1 for accepted, 0 for rejected) only if all three dimensions were satisfied. To ensure objectivity, annotators were not allowed to communicate during the process. After each round, we computed the Fleiss' Kappa score to evaluate inter-annotator agreement. If the score fell below 0.8, annotators convened to review disagreements, refine ambiguous criteria, and update the annotation protocol accordingly. In the first round, the Kappa score was 0.47. After one iteration of guideline refinement, the second round achieved a Kappa of 0.83. The finalized annotation protocol described above was then adopted for the full annotation process. To balance efficiency and agreement validation, we implemented a structured partial-overlap strategy during the main annotation phase. Specifically, we randomly selected 10% of the full dataset (158 projects) for redundant labeling, with the remaining 90% divided evenly among the five annotators. This design enabled us to measure agreement while minimizing redundant effort. On the overlapped subset, the final Fleiss' Kappa score reached 0.79, indicating substantial agreement and ensuring the overall reliability of the human verification process.

Final Sections and Statistics. From over **158** initially shortlisted repositories, only **46** satisfied all verification criteria and were included in the final E2EDEV benchmark.

A.2 DETAILS ON HITL-MAA IMPLEMENTATION

In this section, we present our semi-automated annotation framework for transforming source web application project to fine-grained requirements paired with executable tests. The process begins with a Test-ID Annotation step, which automatically pre-annotates key components in the source code by assigning meaningful and functionally aligned *test identifiers* (Test-IDs). These identifiers serve as semantic anchors—providing structure-independent but functionally meaningful names for UI components—thus enabling reliable downstream requirement binding and test generation. The pre-annotated code is then passed to our human-in-the-loop multi-agent annotator (HITL-MAA), which refines the annotations and produces executable test cases aligned with each fine-grained requirement.

PreAnnotation To address the challenges of annotating key component IDs across multiple source files (HTML and JS), especially considering dynamically generated elements and cross-file dependencies, we designed a **TestID-Annotation Multi-Agent Group** powered by GPT-4o. The annotation workflow is as follows:

Algorithm 1: TestID Annotation Loop by Multi-Agent Group

```
foreach file in project_files do
    if file.type == "html" then
        annotate_interactive_components(file, strategy="add data-testid");
    else if file.type == "js" then
        annotate_dynamic_elements(file, strategy="inject data-testid");

/* Agents share project context to ensure:  */;
    - Cross-file consistency;
    - Non-conflicting test-ids;
    - Semantic and human-readable test-id names;

/* Final review by human annotators ensures correctness */;
```

After annotation, human reviewers verified the validity of each project to prevent execution errors in GPT-4o-generated code. Using this workflow, no issues were identified across 46 projects. Specific prompt details are provided in Appendix G. Subsequently, the LLM uses this annotated source as context to generate requirements and test scripts that directly reference these test IDs. For example:

> **Requirement:** When the user clicks the button with `test-id="login-btn"`, the system navigates to the dashboard.
> **Python Step Implementation:** `... click('[test-id="login-btn"]') ...`

A.2.1 HITL-MAA Annotation Process

Post-processing and Summary Generation. For each data entry, we generate a project-level summary based on the validated requirements. This summary includes a description of the overall application logic, key features, and a list of external resources (e.g., APIs, URLs). These resources are appended to the prompt provided to LLMs in downstream benchmarking to avoid execution failures due to inaccessible external dependencies.

Annotation Statistics. Approximately 20% of the extracted requirements and nearly 50% of the generated test cases required manual refinement by human annotators. This high correction rate of the test cases is not due to inconsistency between the requirement and test case, but rather due to the vagueness in the LLM-generated test descriptions—even when prompted to be detailed. Such vagueness increases the difficulty for the Test Automation Engineer in writing executable Python code. To mitigate this, we adopt an iterative validation process. For step definition validation using dry-run, the system passes within 3 iterations (max_iter1 = 3), with an average of 1.47 iterations. For full test script validation, the average number of required iterations is 4.62. In 20% of the test cases, the maximum iteration limit (max_iter2 = 6) was reached, prompting human intervention to complete the script refinement. All test scripts included in the dataset are verified to be both executable and correct.

A.2.2 Protocols for Human-in-the-Loop & Manual Verification

Human-in-the-Loop Protocol. Human involvement in the E2EDev construction process occurs at three key points:

- Requirement Review and Refinement. Human supervisors validate and refine the user requirements automatically generated by the Requirement Extractor Agent. This ensures accuracy, unambiguousness, and faithfulness to the source project's intended functionality.
- Test Case Refinement and Contribution: Five human supervisors in software tesing collaboratively assess, refine, and contribute additional test cases and interaction patterns beyond those initially generated. They also ensure logical coherence and semantic alignment with requirements, while filtering out erroneous outputs.
- Test Script Refinement (as a fallback): Human supervisors are involved in refining executable test scripts when the Test Automation Engineer Agent reaches its maximum self-correction attempts and cannot autonomously resolve errors or logical inconsistencies.

Manual Verification Protocol & Inner-Agreement. To ensure the consistency and quality of the free-text requirements and test case validation, we implement a *post-hoc gold evaluation* process. First, among five human annotators selected based on their expertise and professional experience in software testing, we designate one senior expert as the *Gold Checker* responsible for reviewing the annotations of the other annotators. This

Algorithm 2: Semi-Automated Annotation Workflow in HITL-MAA

```
Source code annotated with test-ids Executable and validated end-to-end test scripts with aligned
requirements
/* Code Analyzer */;
foreach HTML file in project do
    Analyze frontend framework and interactive components;
foreach JS file in project do
    Analyze logic functionality and connections with frontend elements;
Summarize the analysis for downstream modules;
/* Requirement Extractor */;
Extract contextual requirements using both source code and analysis summary;
user_req_lst ← RequirementExtractor.extract(context);
foreach req in user_req_lst do
    Human annotator validates and corrects the extracted requirement;
    /* Test Case Generator */;
    test_cases ← TestCaseGenerator.generate(source code, req);
    Human annotator validates generated test cases;
    validated_cases ← validated test cases;
    refined_req ← Refine requirement based on validated test cases to ensure alignment;
    foreach test_case in validated_cases do
        /* Test Automation Engineer */;
        impl ← write_test_script(source code, refined_req, test_case);
        /* Step Checker */;
        for i = 1 to max_iter1 do
            Run behave -dry-run to validate step implementation;
            if all steps pass then
                break;
            else
                Modify step implementation based on error message;
        /* Test Runner */;
        for j = 1 to max_iter2 do
            Run full test using behave and parse log;
            if test script passes then
                break;
            else
                Modify test script based on failure log;
        if maximum iterations reached and test still fails then
            Human annotator intervenes to revise and finalize test script;
/* The specific prompt details are provided in Appendix G. */;
```

process employs an iterative annotation and feedback mechanism to resolve inconsistencies in a timely manner. Specifically, for both requirements and test case annotations, four regular annotators initially perform validation. For requirement annotations, annotators interact with the web application through the browser to experience and verify the functionality directly. For test cases, which are written in natural language and serve as step-by-step operational guides, annotators follow the instructions to verify the consistency and validity of the test cases against the requirements. If discrepancies or errors are detected, annotators revise the requirements or test case texts accordingly before submitting them to the Gold Checker for final judgment. The Gold Checker evaluates each annotation from three binary aspects: *equivalence*, *completeness*, and *correctness*, where a score of 1 indicates the criterion is met, and 0 otherwise. Only when the Fleiss' Kappa score for all three aspects exceeds 0.7 does the Gold Checker consolidate a final version of the requirement or test case (which may include discarding it if necessary). If the agreement falls below the threshold, the Gold Checker provides detailed feedback highlighting the inconsistencies to the annotators, who then repeat the annotation process. If any annotator identifies a newly proposed test case, it must undergo the gold evaluation process. In such cases, all three validation aspects are initially scored as 0 by the annotators, and the Gold Checker reviews the new test case before returning it to all regular annotators for reconsideration and re-annotation. In cases where annotators do not recognize a newly proposed test case, they assign all three aspects a score of 0, and the Gold Checker returns the new test case for reconsideration and re-annotation by the regular annotators. This approach effectively balances thoroughness

and efficiency, resulting in a reliable, scalable, and reproducible manual annotation protocol suitable for complex free-text requirement and test case validation tasks.

This rigorous validation workflow ensures high-quality, consistent annotations suitable for downstream tasks.

A.2.3 Analysis on Human Labor during HITL-MAA

We conducted a detailed analysis of human effort and behavior during dataset construction using our human-in-the-loop semi-automated annotation framework. Prior to designing this framework, we engaged five experts without a background in software testing to generate requirements and test cases for each source project via conversational interactions with a large language model (LLM) through a chat interface. This manual process proved to be highly time-consuming, labor-intensive, and yielded poor annotation consistency. During this initial stage, we organized an intensive one-week annotation task where each annotator was assigned 9 independent projects plus 1 additional project overlapping across all annotators to measure inter-annotator agreement. Ultimately, each annotator completed only 5 projects on average. The proportion of executable tests was approximately 80%, including the consistency check projects. The average number of test cases per project was only 6.5. Aggregating the test cases from all five annotators and excluding semantically redundant cases resulted in just 14 unique test cases. Calculating Fleiss' Kappa on these 14 cases yielded a low score of 0.23, indicating poor agreement. Overall, the manual annotation process was not only inefficient but also suffered from incompleteness and quality issues.

In contrast, applying our semi-automated annotation framework substantially improved annotation quality, coverage, consistency, and efficiency. Specifically, all generated executable tests were 100% runnable and logically correct. On average, each project included 3 distinct requirements and 15 test cases, with a maximum of 34 test cases in some projects. Human validation and refinement were necessary for less than 50% of the annotations: approximately 20% of requirement refinements and 50% of test case adjustments. Moreover, manual modifications to test scripts accounted for less than 20% of the total, demonstrating the high reliability of the automated generation.

Despite incorporating a seemingly time-consuming post-hoc consistency evaluation, the average annotation and verification time per project was only 3.5 hours, with the shortest project completed in 30 minutes. Regarding computational resources, the backbone LLM employed was GPT-4o. On average, annotating one project consumed 140,000 prompt tokens and 22,000 completion tokens. This corresponds to a cost of approximately $0.48 per project annotation, indicating a highly efficient process.

Table 6: Comparison of Manual vs. Semi-Automated Annotation

Metric	**Manual Annotation**	**Semi-Automated Annotation**
Average Projects Completed per Annotator	5	>9
Executable Test Ratio	∼80%	100%
Average Test Cases per Project	6.5	15 (max 34)
Fleiss' Kappa (Test Case Agreement)	0.23	0.79 (approx.)
Human Refinement (Requirements)	N/A	20%
Human Refinement (Test Cases)	N/A	50%
Human Refinement (Test Scripts)	N/A	<20%
Average Annotation + Verification Time	$\gg$ 3.5 hours	3.5 hours (min 30 min)
Average Token Consumption (Prompt / Completion)	N/A	140k / 22k tokens
Estimated Cost per Project	N/A	$0.48

Overall, our semi-automated annotation framework significantly enhances annotation completeness and efficiency, while maintaining high-quality and consistent results, demonstrating an effective human-in-the-loop approach to dataset construction.

A.3 Case Studies of E2EDev

E2ESD_Bench_06

"""prompt"""

```
You are tasked with implementing a complete web application using HTML,
JavaScript, and CSS. Your implementation must strictly follow the
specifications described below.

### SUMMARY
# overview
```

```
    The Playable Piano web application allows users to interact with a
virtual piano. Users can click on piano keys or press corresponding keyboard
keys to play notes. The application includes features for adjusting volume
and toggling the visibility of key labels.

# predefined_options
    The web application initializes with a default volume level set to 0.5,
and piano key labels are displayed by default upon launch. It features a
virtual piano with both black and white keys, each mapped to specific
keyboard characters. The black keys correspond to the uppercase letters W, E,
 T, Y, U, O, and P, while the white keys correspond to A, S, D, F, G, H, J, K,
 L, and ;. Each key element in the DOM is assigned a unique data-test-id
attribute in the format data-test-id=\"piano-key-'x'\", where 'x' is the
lowercase version of the corresponding keyboard character (e.g., the key for
W uses data-test-id=\"piano-key-'w'\").

# external_resources
    The corresponding audio files for each piano note are stored in the tunes
 directory, with filenames matching the lowercase key characters followed by .
wav, such as tunes/a.wav, tunes/;.wav, tunes/d.wav, tunes/e.wav, and so on,
covering all relevant keys including w, e, t, y, u, o, p, a, s, d, f, g, h, j,
 k, l, and ;.

# external_js_libraries
    No external JavaScript libraries are used in this application.


### Functional Requirements
    Implement the following features as described. For each requirement, make
 sure the HTML structure, JavaScript behavior, and CSS styles match the
specifications exactly.

# REQUIREMENTS:
- Requirement 1:
    When the user clicks any visible piano key on the webpage, identified by
its data-testid (e.g., 'piano-key-a' for the white key 'a' or 'piano-key-w'
for the black key 'w'), the system must play the corresponding sound file
from the 'tunes' directory (e.g., 'tunes/a.wav' for key 'a'). The clicked key
 must receive the 'active' class to visually highlight it, and the highlight
must be removed after 150ms by removing the 'active' class.
- Requirement 2:...
- Requirement 3:...
- Requirement 4:...
- Requirement 5:...
```

"""requirement_summary"""

```
### overview
    The Playable Piano web application allows users to interact with a
virtual piano. Users can click on piano keys or press corresponding keyboard
keys to play notes. The application includes features for adjusting volume
and toggling the visibility of key labels.

### predefined_options
    The web application initializes with a default volume level set to 0.5,
and piano key labels are displayed by default upon launch. It features a
virtual piano with both black and white keys, each mapped to specific
keyboard characters. The black keys correspond to the uppercase letters W, E,
 T, Y, U, O, and P, while the white keys correspond to A, S, D, F, G, H, J, K,
 L, and ;. Each key element in the DOM is assigned a unique data-test-id
attribute in the format data-test-id=\"piano-key-'x'\", where 'x' is the
lowercase version of the corresponding keyboard character (e.g., the key for
W uses data-test-id=\"piano-key-'w'\").

### external_resources
    The corresponding audio files for each piano note are stored in the tunes
 directory, with filenames matching the lowercase key characters followed by .
wav, such as tunes/a.wav, tunes/;.wav, tunes/d.wav, tunes/e.wav, and so on,
```

```
covering all relevant keys including w, e, t, y, u, o, p, a, s, d, f, g, h, j,
 k, l, and ;.

### external_js_libraries
    No external JavaScript libraries are used in this application.
```

"""fine_grained_reqs"""

```
"1": "When the user clicks any visible piano key on the webpage, identified
by its data-testid (e.g., 'piano-key-a' for the white key 'a' or 'piano-key-w'
 for the black key 'w'), the system must play the corresponding sound file
from the 'tunes' directory (e.g., 'tunes/a.wav' for key 'a'). The clicked key
 must receive the 'active' class to visually highlight it, and the highlight
must be removed after 150ms by removing the 'active' class.",
"3": ...
"4": ...
"5": ...
"6": ...
```

"""reference_answer"""

```
https://github.com/GuanZhizhao/E2EDev/tree/main/E2ESD_Bench_06
```

"""excutable_tests"""

```
### 1
# test_case
Feature: Play piano sound and visually highlight the key when clicked
- The system must play the corresponding piano sound and visually highlight
the key when the user clicks on a piano key. The highlight should disappear
after 150ms.

Scenario: [Normal] User clicks on a white piano key
- Given the webpage is loaded and the piano keys are visible
- When the user clicks on the white piano key with data-testid "piano-key-a"
- Then the system must play the sound
- And the key with data-testid "piano-key-a" must have the "active" class
added
- And the "active" class must be removed from the key with data-testid "piano-
key-a" after 150ms.

# step_code
from behave import given, when, then
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import time

@given('the webpage is loaded and the piano keys are visible')
def step_given_webpage_loaded(context):
    context.driver = webdriver.Chrome()
    context.driver.get(f"file://{file_path}")
    context.driver.maximize_window()

    WebDriverWait(context.driver, 10).until(
        EC.visibility_of_element_located((By.CSS_SELECTOR, "[data-testid='
piano-key-a']"))
    )
    time.sleep(1)  # Allow time for the page to fully load

@when('the user clicks on the white piano key with data-testid "piano-key-a
"')
def step_when_user_clicks_piano_key(context):
    piano_key = WebDriverWait(context.driver, 10).until(
        EC.element_to_be_clickable((By.CSS_SELECTOR, "[data-testid='piano-key-
a']"))
```

```
    )
    piano_key.click()

@then('the system must play the sound')
def step_then_system_plays_sound(context):
    pass

@then('the key with data-testid "piano-key-a" must have the "active" class
added')
def step_then_key_has_active_class(context):
    piano_key = context.driver.find_element(By.CSS_SELECTOR, "[data-testid='
piano-key-a']")
    class_list = piano_key.get_attribute("class").split()
    assert "active" in class_list, "The 'active' class was not added to the
key"

@then('the "active" class must be removed from the key with data-testid "
piano-key-a" after 150ms')
def step_then_active_class_removed(context):
    time.sleep(0.15)
    piano_key = context.driver.find_element(By.CSS_SELECTOR, "[data-testid='
piano-key-a']")
    class_list = piano_key.get_attribute("class").split()
    assert "active" not in class_list, "The 'active' class was not removed
from the key"

def after_scenario(context, scenario):
    if hasattr(context, 'driver'):
        context.driver.quit()

# test_case ...
# step_code ...
# test_case ...
# step_code ...

### 3 ...
### 4 ...
### 5 ...
### 6 ...
```

A.4 Statistics and Features of E2EDev

Statistics. Dataset statistics are shown in Table 2, with example entries provided in the Appendix A.3.

Features. E2EDev consists of diverse software development tasks evenly distributed across seven common types of web applications. As shown in Figure 8(a) , the three most frequent categories include: *Mathematics & Conversion Tools*, such as basic calculators and utilities for tax or BMI calculation; *Account & Form Management*, covering simple CRUD systems where users submit forms and manage records; and *Mini Games & Interactive Entertainment*, including lightweight games like Tic-Tac-Toe. E2EDev also captures a wide range of common user interaction patterns. As shown in Figure 8(b), over 90% of projects involve basic interactions like clicking and typing, while others include more advanced actions such as sliders, selectors, and drag-and-drop operations. In addition to basic DOM manipulation, the dataset includes non-trivial web application features. For example, nearly one-third of the projects require reasoning over complex mathematical functions, demanding precise function generation to satisfy user needs. About one-quarter of the tasks involve interaction with local storage. Another quarter rely on external APIs for data fetching, audio playback, and more—testing whether models can correctly integrate external resources. Taken together, the diversity in application types, interaction patterns, and technical complexity demonstrates that E2EDev provides a representative and realistic benchmark for end-to-end web application development.

B Implementation Details

B.1 Implementation of Baselines

LLM-based Frameworks for End-to-End Software Development. Recent advances in E2ESD using LLMs primarily fall into two categories: multi-agent and single-agent frameworks. The majority of existing work adopts a multi-agent paradigm, where the overall development process is decomposed into subtasks based on classical

Figure 8: This figure illustrates the features of E2EDev, which encompasses seven common types of web applications. It covers a wide range of user interaction types and incorporates various advanced web technologies.

software engineering models (Sommerville, 2011). Within this category, Self-Collaboration (Dong et al., 2024) mimics the fundamental development stages—requirement analysis, implementation, and testing—via specialized agents. More structured approaches (Hong et al., 2024; Du et al., 2024; Rasheed et al., 2024; Sami et al., 2024) leverage the Waterfall model (Royce, 1987) to coordinate agent collaboration. Beyond this, MapCoder (Islam et al., 2024) incorporates prior project knowledge to improve software generation. In addition, while most frameworks rely on predefined workflows, recent efforts (Qian et al., 2024; Lin et al., 2024) explore dynamic agent interactions. ChatDev (Qian et al., 2024), for instance, enables agents to engage in multi-turn dialogues to iteratively refine the software product. In contrast, a smaller body of work pursues a single-agent approach, which assumes that a single LLM has sufficient capacity to independently complete the entire development cycle without explicit task decomposition (Engineer, 2024).

Implementations of LLM-based Frameworks. We reproduced six LLM-based code generation frameworks using their official GitHub implementations (if any): *Vanilla LLM*, *GPT-Engineer*, *Self-Collaboration Code Generation*, *MapCoder*, *ChatDev*, and *MetaGPT*.

- For the Vanilla LLM baseline, we directly fed the task prompt (i.e., user requirements) together with the following structured instruction:

```
Please wrap each part of the implementation in its appropriate Markdown code block
:
    - Use """html for the HTML structure in **index.html**.
    - Use """css for the CSS styles in **style.css**.
    - Use """javascript for the JavaScript functionality in **script.js**.

Ensure that the files are named exactly as indicated: **index.html**, **style.css
**, and **script.js**.
This will help maintain a consistent structure and make the code easily integrable.
```

  This instruction guides the model to generate modularized web projects containing HTML, CSS, and JavaScript components.
- For GPT-Engineer and MetaGPT, we used their official implementations In particular, we employed their corresponding official Python packages, readily available for import and use. During our experiments, we executed the frameworks directly by feeding them the task prompts. No modifications were made to their internal workflows.
- For the remaining frameworks—Self-Collaboration Code Generation, MapCoder, and ChatDev—we implemented all baselines using their publicly available source code from GitHub. However, both Self-Collaboration

and MapCoder were originally designed for Python-only tasks. To adapt them to our multi-language (HTML/CSS/JavaScript) setting, we modified their agent system prompts accordingly.

Specifically, for Self-Collaboration, we replaced the default `PYTHON_DEVELOPER` role with an `H5_DEVELOPER` role, enabling the agent to handle web development tasks:

```
H5_DEVELOPER = """
    I want you to act as an H5 developer on our development team. ...,
    write HTML, JavaScript, and CSS code ... provide the updated HTML,
    JavaScript, and CSS code ...
"""
```

A similar modification was applied to MapCoder, adapting its original Python-specific prompts to support full-stack web development. These adaptations ensured fair comparisons across all frameworks while preserving the core design principles of each system.

B.2 Implementation of Evaluation Metrics

We evaluate both code effectiveness and generation efficiency in E2ESD. On one hand, code effectiveness measures how well the generated software satisfies user requirements. In this regard, we propose three task-level metrics:

- *Req.Acc:* The average correctness across all requirement tasks per project. For a project with M requirements, if N are fully satisfied (i.e., all associated test cases pass), then:

$$\text{Req.Acc} = \frac{N}{M} \tag{1}$$

- *Test.Acc:* The proportion of all passed test cases across a project:

$$\text{Test.Acc} = \frac{\text{Total Passed Test Cases}}{\text{Total Test Cases}} \tag{2}$$

- *Balanced Score:* To mitigate bias from varying test case granularity per requirement, we define a weighted effectiveness score:

$$\text{Balanced} = \alpha \cdot \text{Req.Acc} + \beta \cdot \text{Test.Acc} \tag{3}$$

  where α and β control the relative importance of requirement-level and test-level correctness, respectively. In this paper, we set $\alpha = 0.6$ and $\beta = 0.4$, reflecting our intuition that requirement-level accuracy (Req.Acc) plays a more critical role in evaluating the overall performance than test-level accuracy (Test.Acc).

On the other hand, generation efficiency is assessed using three metrics:

- *Cost (USD):* Estimated based on token usage and API pricing. The total cost is computed as:

$$\text{Cost} = (\text{Input Tokens} \times \text{Input Price}) + (\text{Output Tokens} \times \text{Output Price}) \tag{4}$$

  where the *Input Price* and *Output Price* refer to the per-token charges for input and output tokens, respectively, as defined by the corresponding LLM provider. The pricing used in our experiments is summarized in Table 7.

Table 7: LLM API Pricing

Model	**Input Price (USD/token)**	**Output Price (USD/token)**
`gpt-4o`	0.0025	0.0061
`gpt-4o-mini`	0.00015	0.00014
`qwen-7b`	0.000069	0.00006
`qwen-70b`	0.00055	0.0016
`qwen-max`	0.00033	0.0013

- *Carbon Footprint (CO_2):* Computed as the sum of operational and embodied carbon, following (Faiz et al., 2023). We assume NVIDIA A100 as the default chip for estimation purposes.

$$\text{Footprint} = \text{Flops} \cdot \frac{TDP}{\eta} \cdot \text{PUE} \cdot \text{CI} + \text{EC} \tag{5}$$

  where:
  - **FLOPs** $= 2 \times (\text{Input Tokens} + \text{Output Tokens}) \times \text{Active Parameters}$
  - **Active Parameters:** The number of model parameters actively involved in the computation for each token.
  - **TDP:** Thermal Design Power of the GPU, measured in watts. For A100 40GB SXM, we assume 400W.

- η: Computational efficiency, measured in FLOPs/Watt. This measures the floating-point operations per watt (FLOPs/W). For A100, we assume 624 TFLOPs (FP16 with sparsity), noting that mixed-precision may be used in practice.
- **PUE:** Power Usage Effectiveness. A metric reflecting datacenter energy overhead. PUE for the Qwen series is 1.2, while for the GPT series it is 1.1.
- **CI:** Carbon intensity, measured in $kgCO_2$/kWh, representing the amount of CO_2 emitted per unit energy consumed(Patterson et al., 2021).CI is 625 g/kWh for the Qwen series and 407 g/kWh for the GPT series .
- **EC:** Embodied carbon (CO_2 generated during manufacturing and infrastructure), usually treated as a negligible constant in inference scenarios.

• *Duration (s):* Total wall-clock time for software generation.

B.3 Implementation of Human Evaluation in Section 4.3

To support our fine-grained analysis of software requirement satisfaction (Section 4.3), we conducted a human evaluation to identify and categorize different types of implementation failures. The goal is to assess not just whether a function exists, but whether it is implemented correctly, meaningfully, and according to the intended specification.

Task Setup. We randomly selected 10 tasks from the E2EDev benchmark and evaluated their implementations under 30 different models (5 backbones $\times$ 6 frameworks), resulting in 300 model-task instances. We recruited 5 professional annotators with expertise in software testing. To ensure labeling consistency, 5 shared examples were used to calibrate judgments, and the remaining 295 instances were evenly distributed across annotators.

Each annotator was asked to assess whether a web application satisfies a specific requirement described in natural language. Importantly, annotators were guided by refined and detailed requirement statements (see Appendix A.3), which already include edge cases and expected behaviors. The annotators interact directly with the browser-based application and determine whether the system's observable behavior aligns with the requirement.

Why Requirement-Level Evaluation? We opt for requirement-level evaluation instead of simulating every atomic test case due to scalability and redundancy. Evaluating 64 requirements per annotator is already resource-intensive. Expanding to all test cases would result in over 1,000 evaluations (e.g., 16 test cases per task $\times$ 64 tasks), which is infeasible. Moreover, our HITL-MAA pipeline already aligns test cases with refined requirements. Hence, human evaluation at the requirement level is both efficient and representative.

Error Typology. Based on interactions and source code inspection, annotators classified failures into four categories. Table 8 summarizes the definitions and identification criteria.

Table 8: Error types identified during human evaluation

Error Type	Definition and Identification Criteria
Requirement Missing	The required functionality is completely absent. Annotators cannot find relevant UI elements or supporting code. For example, a login feature is required but no login form or logic exists in the codebase.
Code Inconsistency	UI components refer to expected behaviors (e.g., an 'onClick' handler), but the linked code is missing, empty, or non-functional. The interface appears intact, but interaction triggers no effect.
Requirement Misaligned	The core functionality exists, but the way it is implemented deviates from the specification. For example, deleting a to-do item is required via right-click, but is instead implemented through a separate button.
Detail Mismatch	The main function appears to work, but issues arise under edge cases, unexpected actions, or incorrect feedback content. These require nuanced inspection to reveal.

Inter-Annotator Reliability. To assess labeling quality, we compute inter-annotator agreement using Cohen's Kappa. We report a strong agreement score of 0.86 among human annotators and 0.74 between human and automatic evaluation, suggesting both robustness and validity of the human-labeled results.

Table 9: Soft Req. Acc. and Req. Acc. under all LLMs we used. Here, V-LLM refers to the Vanilla LLM, GPT-E refers to GPT-Engineer, and Self-C refers to Self-Collaboration.

Base Model	Method	Req.acc	Soft Req.acc	Delta
qwenmax	ChatDev	0.4307	0.7025	0.2718
	gpt-engineer	0.4979	0.7454	0.2475
	llmbased	0.4335	0.7047	0.2712
	MapCoder	0.4837	0.7605	0.2768
	MetaGPT	0.0163	0.0379	0.0216
	Self-collaboration-Code-Generation	0.4268	0.7110	0.2842
qwen7b	ChatDev	0.1803	0.4414	0.2611
	gpt-engineer	0.2403	0.5718	0.3315
	llmbased	0.2237	0.5090	0.2853
	MapCoder	0.1283	0.3967	0.2684
	MetaGPT	0.0000	0.0000	0.0000
	Self-collaboration-Code-Generation	0.2058	0.4918	0.2860
qwen70b	ChatDev	0.4352	0.7053	0.2701
	gpt-engineer	0.4208	0.7064	0.2856
	llmbased	0.3575	0.6674	0.3099
	MapCoder	0.4044	0.6949	0.2905
	MetaGPT	0.0000	0.0045	0.0045
	Self-collaboration-Code-Generation	0.4257	0.6727	0.2470
gpt4omini	ChatDev	0.3395	0.6280	0.2885
	gpt-engineer	0.4310	0.6908	0.2598
	llmbased	0.4572	0.7417	0.2845
	MapCoder	0.4089	0.6884	0.2795
	MetaGPT	0.0000	0.0071	0.0071
	Self-collaboration-Code-Generation	0.3876	0.6864	0.2988
gpt4o	ChatDev	0.4275	0.6928	0.2653
	gpt-engineer	0.5037	0.7730	0.2693
	llmbased	0.4623	0.7346	0.2723
	MapCoder	0.4792	0.7228	0.2436
	MetaGPT	0.0000	0.0048	0.0048
	Self-collaboration-Code-Generation	0.4614	0.7064	0.2450

Table 10: Error distribution across frameworks powered by GPT-4o-mini and Qwen-7B. CIc denotes code inconsistency, MR refers to missing requirement, MwR indicates mismatch with requirement, and DM stands for detail mismatch.

Base Model	Method	CIc	MR	MwR	DM
gpt4omini	V-LLM	0	3	9	22
	GPT-E	2	1	13	18
	Self-C	1	0	12	20
	MapCoder	1	0	3	26
	ChatDev	3	0	8	24
	MetaGPT	36	15	0	3
qwen7B	V-LLM	12	5	20	11
	GPT-E	3	2	21	18
	Self-C	12	0	21	10
	MapCoder	44	0	0	2
	ChatDev	12	0	15	0
	MetaGPT	21	33	0	0

C ADDITIONAL ANALYSIS

C.1 ADDITIONAL EFFECTIVENESS ANALYSIS ON E2EDEV

***Does strong standalone performance of a Vallina LLM guarantee its effectiveness as the backbone of an agentic framework?* – No** For a more intuitive understanding of this questions, we present a bar chart in Fig.5 that clearly highlights the performance gaps between different Agentic Framework powered by powerful LLM Backbones to better indicates which Agentic framework outperperform Valinna LLM, demonstrating their effectiveness, the yellow zone indicates the req.acc of each Vallina LLM (From left(GPT-4o) to the right(Qwen-7B) their performance degrades sequentially). Interestingly, we find that employing a more powerful vanilla LLM as the backbone does not always yield better results on the E2ESD task when integrated into an agentic framework. For example, both GPT-4o-mini exhibit performance drops across several agentic frameworks while it's vallina model performs surprisingly well as a vanilla LLM, with a performance gap to GPT-4o being relatively small. We hypothesize that this is because GPT-4o-mini is a distilled version of GPT-4o and retains many of its generalist strengths, as evidenced by its comparable performance on several benchmarks—except for complex knowledge-intensive tasks such as GPQA, where it shows a clear degradation. In our E2ESD task, although each instance contains 2–10 user requirements, they primarily involve standard software functionalities and do not require specialized knowledge. As a result, GPT-4o-mini performs well as a standalone model. However, it remains a relatively small model, estimated at around 8B parameters (Abacha et al., 2024). Like other small models, it suffers from common issues such as performance instability and limited robustness. When used as a backbone within an agentic framework—particularly under heavy task prompts, coupled with additional system-level constraints—its understanding of the task may be compromised. Furthermore, in multi-agent pipelines, the likelihood of error propagation and accumulation increases. This issue is also observed with Qwen-7B. A detailed analysis of this behavior will be provided in the case study section.

***Does equipping an agentic framework with a more powerful LLM backbone necessarily lead to better effectiveness?* – No. The effectiveness of an agentic framework depends not only on LLM capability, but also on instruction-following alignment.** As shown in Fig.5, performance varies across agentic frameworks when powered by different LLM backbones. We find that instruction-following ability is key the to a reliable backbone, as the success of an agentic framework relies on each agent following its instructions while staying aligned with others. Among all tested backbones, Qwen-70B (instruction-tuned version) shows the most stable performance, particularly in dynamic systems like ChatDev. It achieves higher requirement accuracy and fewer dialogue turns, as shown in Table4, indicating superior alignment and controllable. The detailed analysis of "How Instruction-Following Affects the Agentic Workflow" will follow in the case study section.

C.2 ADDITIONAL EFFICIENCY ANALYSIS ON E2ESD

Inefficiencies in ChatDev's dialogue process are primarily attributed to the models' failure to adhere to flexible stopping signals specified in the prompt, particularly in multi-phase interactions. As seen in Figure 9, in Phase 2, the task simply requires selecting a programming language and returning a line in the format "<INFO> *" (where * is the chosen language) indicating the end of the phase. However, models frequently terminate with "<INFO> Finish", violating the instruction and causing unnecessary dialogue iterations. This problem is especially pronounced in smaller models like Qwen-7B, which often reach the 20-turn interaction cap even for such trivial tasks. Similar inefficiencies are observed in later stages (e.g., Phases 5 and 6, CodeReview), where all models reach the maximum number of turns. Although the prompt explicitly states: "If no bugs are reported, please return only one line like <INFO> Finished," minor deviations such as surrounding whitespace prevent proper loop termination. Notably, early-stage phases like DemandAnalysis terminate more reliably, likely due to shorter generated outputs and less instruction interference from long context history. These findings highlight a key limitation of ChatDev's current stopping mechanism design: it relies on rigid string-based termination signals that are fragile under imperfect generation and long-context dependencies. Future multi-agent frameworks should consider more adaptive and robust exit strategies, especially under conditions where instruction following is context-sensitive and generation is uncertain.

C.3 ADDITIONAL ANALYSIS ON HUMANEVAL (FUNCTION-LEVEL TASK)

The experimental results presented in the main text indicate that current methods are not effective in handling project-level ESESD tasks. To gain further insights, this chapter shifts focus to evaluate the performance of these methods on simpler, function-level tasks, using the benchmark dataset HumanEval (Chen et al., 2021).

As summarized in Table 11, there is a noticeable improvement in the performance of various methods on the HumanEval benchmark compared to their results on the more complex E2ESD task. This suggests that current approaches are still primarily effective for simpler, function-level code generation tasks. It is worth noting that MetaGPT's performance on HumanEval aligns closely with the results reported in its original paper (see the supplementary material of their work for details).

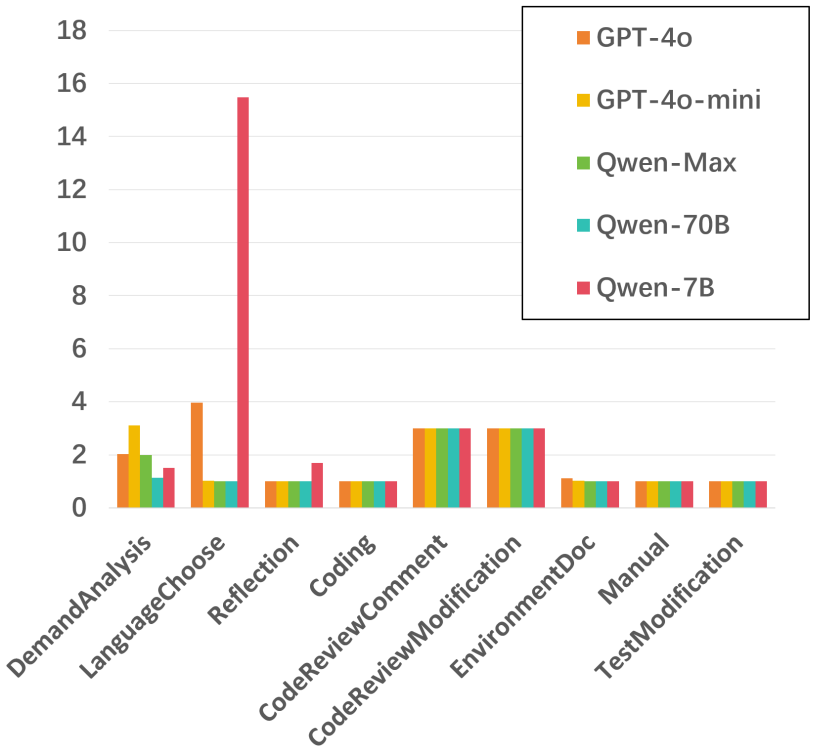

Figure 9: Number of communications in each phase of ChatDev across different backbone models.

C.4 Case Studies on LLM-driven Methods for E2ESD

Task: The application is a web-based calculator that allows users to perform basic arithmetic operations. It includes a display screen for showing inputs and results, a set of buttons for numbers and operators, and a theme toggle feature to switch between dark and light modes.

Relevant Requirement:

- **Requirement 2:** When a user interacts with the calculator's operator buttons, the system should append the corresponding operator to the display screen.
- **Edge Case:** If the user enters multiple consecutive operators (e.g., `+`, `-`, `*`, `/`), only the last one should be retained in the display.

Information Given to the Coding Agent:

Self-Collaboration

```
{
  "subproblems": [
    ...
    {
      "id": 3,
      "description": "Create JavaScript functionality to append operators to
the display, handling empty display and consecutive operator clicks."
    },
    ...
  ],
  "high_level_plan": [
    ...
```

Table 11: Performance of off-the-shelf methods across various LLM backbones on HumanEval and HumanEval-ET. We use Pass@k (with $k = 1$) to evaluate effectiveness. Pass@k (ET) refers to results on HumanEval-ET, which includes more test cases per function. For clarity, the highest score within each group is shown in **bold**, and the second highest is underlined.

Backbone LLM	Method	Effectiveness (↑, %)		Efficiency (↓)		
		Pass@1	Pass@1 (ET)	Cost (USD)	Footprint (gCO_2eq)	Duration (s)
GPT-4o	Vanilla LLM	89.02	80.49	0.0045	0.018	13
	GPT-Engineer	**91.46**	81.10	0.0043	0.030	28
	Self-Collaboration	90.85	81.10	0.0116	0.077	48
	MapCoder	90.24	**81.71**	0.0542	0.338	386
	ChatDev	90.85	**81.71**	0.0760	0.619	404
	MetaGPT	45.73	0.61	0.0429	0.338	109
GPT-4o-mini	Vanilla LLM	84.15	76.22	0.0003	0.001	14
	GPT-Engineer	86.59	**79.27**	0.0003	0.001	65
	Self-Collaboration	86.59	75.61	0.0006	0.003	42
	MapCoder	**87.20**	76.83	0.0027	0.012	71
	ChatDev	85.37	75.61	0.0067	0.038	108
	MetaGPT	50.61	1.83	0.0025	0.013	325
Qwen-Max	Vanilla LLM	89.63	81.71	0.0009	0.012	26
	GPT-Engineer	**90.85**	**82.32**	0.0006	0.014	31
	Self-Collaboration	84.15	76.22	0.0017	0.038	117
	MapCoder	90.24	81.10	0.0080	0.177	211
	ChatDev	89.02	80.49	0.0104	0.302	215
	MetaGPT	82.32	6.10	0.0043	0.124	118
Qwen-70B	Vanilla LLM	**87.20**	**77.44**	0.0008	0.012	27
	GPT-Engineer	83.54	73.17	0.0007	0.017	17
	Self-Collaboration	82.31	75.00	0.0020	0.043	67
	MapCoder	83.54	74.39	0.0102	0.212	218
	ChatDev	82.23	71.34	0.0160	0.377	278
	MetaGPT	53.66	3.05	0.0085	0.208	229
Qwen-7B	Vanilla LLM	**81.71**	71.95	0.0001	0.001	15
	GPT-Engineer	**81.71**	**75.00**	0.0001	0.001	10
	Self-Collaboration	71.34	64.02	0.0002	0.004	41
	MapCoder	76.83	68.29	0.0011	0.023	118
	ChatDev	73.17	65.24	0.0027	0.065	352
	MetaGPT	30.49	1.22	0.0006	0.014	189

```
    {
      "step": 3,
      "description": "Write JavaScript to handle operator button clicks,
ensuring proper handling of operator sequences and interaction with numbers
on the display."
    },
    ...
  ]
}
```

Analysis: Self-Collaboration explicitly addresses the edge case of consecutive operator inputs in both its subproblem breakdown and high-level plan. This explicit guidance enables the coding agent to implement this logic correctly, resulting in successful execution of the test case when powered by GPT-4o.

MapCoder

```
Recall three (03) relevant and distinct problems (different from the user
task).
# Problem 1: To-Do List Application
# Code
# Plan to Solve the problem
# Problem 2: Simple Weather App
```

```
# Code
# Plan to Solve the problem
# Problem 3: Simple Quiz Application
# Code
# Plan to Solve the problem


## Planning:

1. **HTML Structure**: ...

2. **CSS Styling**: ...

3. **JavaScript Functionality**:
   - Implement event listeners for number and operator buttons to update the
display.
   - Implement the clear button functionality to reset the display.
   - Implement the backspace button functionality to remove the last
character from the display.
   - Implement the equal button functionality to evaluate the expression and
handle special cases.
   - Implement the theme toggle functionality to switch between dark and
light themes.

4. **Testing**: ...
```

Analysis: MapCoder generates solutions based on analogies to unrelated projects, which resemble few-shot prompting. However, none of the referenced examples include edge-case handling for consecutive operators. Moreover, the planning stage does not treat operator input as a separate requirement, leading to insufficient focus during code generation and ultimately causing the failure in this edge case.

ChatDev

```
### Core Components

1. **HTML Structure (`index.html`)**
   ...

2. **JavaScript Functionality (`script.js`)
   - **handleNumberClick**: Appends clicked number to the display.
   - **handleOperatorClick**: Appends clicked operator to the display.
   - **handleClear**: Clears the display.
   - **handleBackspace**: Removes the last character from the display.
   - **handleEquals**: Evaluates the expression on the display.
   - **toggleTheme**: Toggles between dark and light themes.

3. **CSS Styling (`styles.css`)
   ...
```

Analysis: ChatDev does not mention the edge case of consecutive operator input at all. Furthermore, the function `handleOperatorClick` — which is crucial to this behavior — lacks any instruction regarding handling such cases. As a result, the generated implementation fails to address the edge case, leading to incorrect behavior in testing.

D INTRODUCTION ON PROJECT/SOFTWARE TESTING

Adhering to software engineering principles, we present the E2EDev benchmark for evaluating the performance of LLM-based frameworks in E2ESD. The benchmark comprises: (1) a detailed list of user requirements for each software project; (2) for every user requirement, multiple test cases are provided, each accompanied by a corresponding executable test script; and (3) an automated testing pipeline built upon the Behave framework. This

appendix offers a brief overview of the software engineering principles underlying project/software testing relevant to the benchmark.

Overview. In modern software engineering, systematic testing is essential for ensuring software quality, reliability, and correctness. Project/software testing refers to a structured process of verifying that a software system meets specified requirements and behaves as expected under various conditions. Testing is generally guided by predefined requirements and specifications and aims to uncover bugs, ensure compliance with user needs, and validate system behavior.

The E2EDev benchmark follows the best practices of behavior-driven development (BDD), where test cases are derived directly from user stories or requirements, as shown in Figure 10.. This approach enhances traceability between software specifications and their validations, thereby facilitating more interpretable and maintainable test suites. To this end, we adopt Gherkin as a high-level test specification language and Behave as the test execution framework. These tools allow us to simulate realistic software engineering workflows while enabling automated assessment of LLM-generated artifacts.

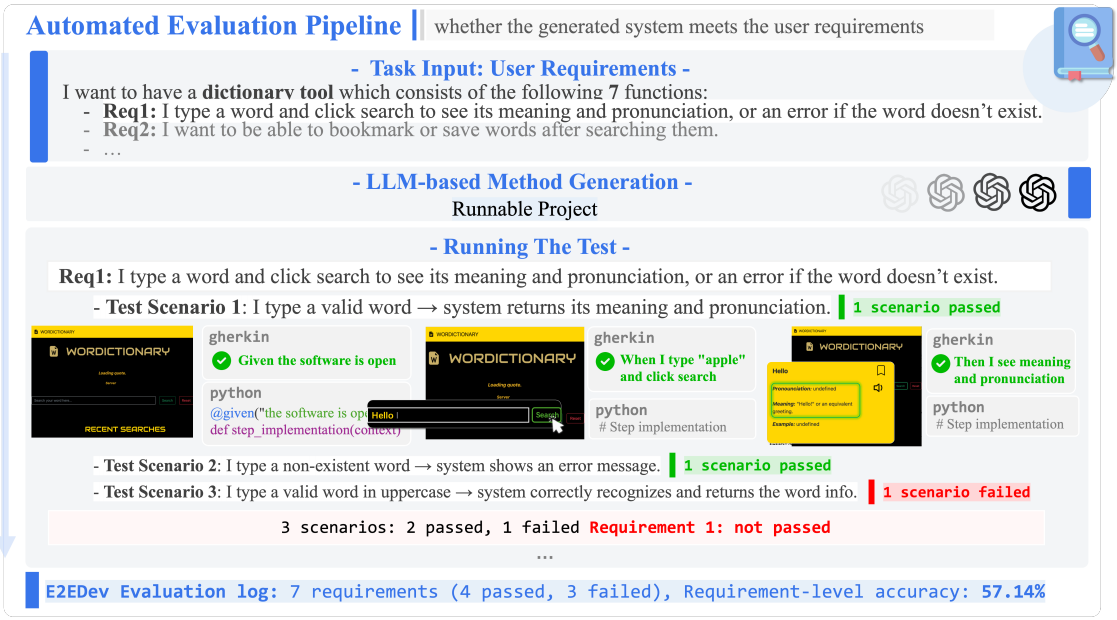

Figure 10: BDD-based automated evaluation pipeline for E2ESD tasks.

D.1 Test Case Specification – Gherkin

Gherkin is a domain-specific language designed for behavior-driven development (BDD), allowing test cases to be written in a natural, human-readable format. It serves as the standard for specifying software behavior in BDD tools like `Behave`.

A typical Gherkin file uses structured keywords such as:

- `Feature`: Describes the functionality under test.
- `Scenario`: Represents a specific use case or behavior.
- `Given`: Defines the initial system state or context.
- `When`: Specifies the user action being simulated.
- `Then`: Declares the expected outcome or result.

In the **E2EDev benchmark**, each test case is written using Gherkin syntax to promote consistency and alignment with real user needs. This format ensures test cases are both readable and executable, enabling a smooth translation from natural language descriptions to automated tests.

Below is an example of a well-commented Gherkin test case:

```
# Feature: Describes a group of related scenarios
Feature: Hunger Expression Button

  # Scenario: A single test case that verifies a specific behavior
  Scenario: User clicks the "I'm HUNGRY" button
```

```
    # Given: Sets up the initial state of the app
    Given the web app is loaded in the browser

    # When: Simulates a user action
    When the user clicks the "I'm HUNGRY" button

    # Then: Specifies the expected result of the action
    Then the display should show the message "I'm HUNGRY"
```

Listing 1: Example Gherkin Test Case with Comments

D.2 Test Script & Automated Testing Framework – Behave

Behave is an open-source, Python-based behavior-driven development (BDD) testing framework that interprets Gherkin-formatted test cases and maps each step to executable Python functions. These functions define how the application should behave in response to specific user actions or conditions.

In the **E2EDev benchmark**, Behave is used to implement an end-to-end testing pipeline. Each test scenario is accompanied by corresponding Python step definitions that simulate interactions with the system under test—such as clicking a button or verifying UI output. This setup enables scalable, repeatable testing of LLM-generated user interfaces or functionality against expected behavior.

The integration of Behave ensures that the benchmark remains extensible and maintainable, supporting continuous integration and empirical evaluations of LLM-based development systems.

Below is an example of a simple Python step implementation that supports the Gherkin scenario shown earlier:

```python
from behave import given, when, then
from selenium import webdriver
from selenium.webdriver.common.by import By

@given('the web app is loaded in the browser')
def step_load_app(context):
    context.driver = webdriver.Chrome()
    context.driver.get("file:///path/to/your/app.html")

@when('the user clicks the "I\'m HUNGRY" button')
def step_click_button(context):
    button = context.driver.find_element(By.ID, "btn-hungry")
    button.click()

@then('the display should show the message "I\'m HUNGRY"')
def step_verify_output(context):
    output = context.driver.find_element(By.ID, "display")
    assert output.text == "I'm HUNGRY"
```

Listing 2: Python Step Implementation for Gherkin Scenario

D.3 How to Conduct Testing Using E2ESD?

To perform testing with E2ESD, we leverage the `Behave` framework. Begin by placing your prepared Gherkin test cases in the `features/` folder. Corresponding Python step implementations should be stored in a designated folder (e.g., `steps/`), where the URLs referenced in the code must be updated to point to your local HTML files under test. An example of the file structure is shown in Fig. 11.

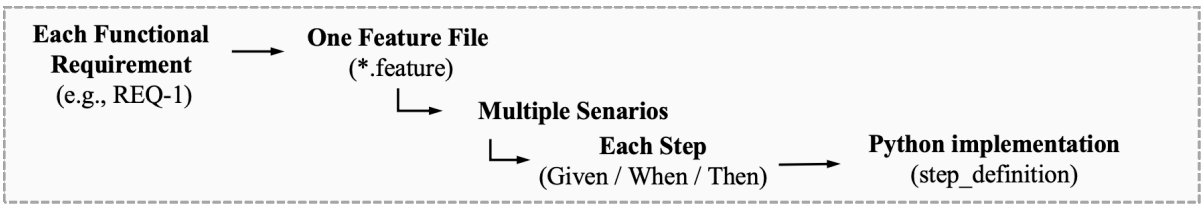

Figure 11: Example directory structure for Behave testing with E2ESD.

Once the setup is complete, you can run the tests by executing the `behave` command in the terminal. The only requirement is that the testing machine must have ChromeDriver installed. This is because our testing

process simulates user interactions with the browser and thus relies on browser automation. You can download the appropriate version of ChromeDriver from: `https://developer.chrome.com/docs/chromedriver`.

For ease of use, we have encapsulated the entire testing process into a Python function. You only need to specify the root directory of your project in the function, and the framework will automatically execute all the necessary steps described above. It will return the complete Behave log files for all test cases.

E INTRODUCTION ON UNIT TEST & BDD TEST

E.1 BEHAVIOR-DRIVEN DEVELOPMENT (BDD) TESTS

Behavior-Driven Development (BDD) tests are designed to verify the behavior of a system from the perspective of an end user. They describe expected outcomes in specific scenarios using structured natural language, typically in the `Given-When-Then` format. BDD tests bridge the gap between technical implementation and business requirements, allowing non-technical stakeholders to understand and validate software behavior.

E.2 UNIT TESTS

Unit tests focus on verifying the correctness of the smallest units of code, such as functions or classes, in isolation. They are written in programming languages and target implementation details and internal logic. Unit tests provide fast feedback to developers about the correctness of individual components.

E.3 COMPARISON

As shown in Table 12, BDD tests focus on validating system behavior from a user perspective, whereas unit tests verify individual code units.

Aspect	**BDD Tests**	**Unit Tests**
Focus	Validating system behavior from user perspective	Verifying correctness of individual code units
Expression	Structured natural language (`Given-When-Then`)	Programming language, code-centric
Purpose	Ensure software meets user requirements	Ensure individual components work as intended
Granularity	High-level, scenario-based	Low-level, function/class-based
Automation	Executed by frameworks like Behave or Cucumber, mapping natural-language scenarios to code steps	Executed within testing frameworks like `unittest` or `pytest`
Audience	Developers and non-technical stakeholders	Primarily developers

Table 12: Comparison between BDD tests and Unit tests

The introduction of the `E2EDev` benchmark provides a more realistic and rigorous foundation for evaluating LLMs in E2ESD scenarios. By enabling systematic and reproducible assessment of LLM capabilities, our work has the potential to accelerate technical advancements and promote the automation of software engineering tasks, ultimately increasing productivity and lowering entry barriers for development. However, such progress may also contribute to the displacement of certain roles traditionally performed by human software engineers, raising concerns about job security and workforce transitions. We encourage the community to consider both the positive and potentially disruptive impacts of this technology and to explore pathways for responsible deployment and skill adaptation.

F BENCHMARK LIMITATIONS

While E2EDEV includes 244 user requirements and a large number of test cases, it covers only 46 software projects. This limited project scope may affect the benchmark's generalizability. However, constructing such a benchmark is inherently costly and labor-intensive. As shown in Figure 2, a single project typically contains multiple functionalities, each requiring one or more detailed test cases. In web development, these often involve simulating user interactions, adding further complexity. To produce a single executable test, we must first write a Gherkin-based test scenario, then implement corresponding step definitions to enable automated execution via tools like `Behave`. This pipeline demands precise annotations of requirements, functions, test scripts, and source

code, significantly increasing annotation costs. Moreover, even with high-quality annotations, such test data cannot be reused across different LLM-based methods. This is because the generated code structures vary greatly between methods, making pre-defined test scripts incompatible. These challenges explain why the dataset size remains modest despite the substantial annotation effort. Recognizing these substantial obstacles, our work makes a critical contribution by grounding E2ESD task evaluation firmly within established software testing practices. This fundamental shift largely overcomes the existing benchmark's limitations in producing reliable and practically relevant evaluations, thereby substantially expanding the scope and rigor of research in this area. We believe that our approach not only provides a more reliable assessment but also lays a crucial foundation for future advancements in AI-driven software development.

G PROMPTS DETAILS

Test-id Annotator

"""system prompt"""

```
You are an automation tool tasked with adding `data-testid` attributes in the
 provided HTML and JavaScript files. Your job is to ensure that all
interactive and display components have unique, meaningful identifiers while
preserving the original code's functionality, structure, and behavior. When
making changes, you must adhere to the following rules:

1. **Preserve Original Code Logic & Structure**:
   - Only add `data-testid` attributes to components without modifying the
original code.
   - Do not modify the existing functionality, structure, or behavior of the
code.

2. **Interactive Components (`data-testid` Assignment)**:
   - Assign a `data-testid` to any interactive component that lacks one,
including `button`, `input`, `select`, `textarea`, `a`, `label`, `link`, etc.
   - **If an element already has an `id`, it must also have a `data-testid`
with the same value or an appropriate variation**.
   - If an element **already has a `data-testid`**, modify it only if it does
 **not** conform to the naming convention.

3. **Naming Convention for `data-testid`s**:
   - Use clear and meaningful names that describe the element's purpose or
role.
   - Example:
     - `submit-button`: A button used to submit a form.
     - `active-menu-item`: A menu item that is currently active.
     - `close-modal-button`: A button used to close a modal.

4. **Ensure Unique `data-testid`s**:
   - If multiple similar elements exist, append an increasing number (e.g., `
menu-item-1`, `menu-item-2`).

5. **Output**:
   - Provide the rewritten HTML and JavaScript files with correctly assigned `
data-testid`s.
   - Maintain the original code structure and logic while ensuring compliance
 with the naming rules.
   - **No modifications to the CSS files are required**.

**Limitations**:
- Do not change the code structure: Only add `data-testid` attributes. Do not
 refactor JavaScript logic or HTML structure.
- Do not change functionality: Ensure that the original functionality and
behavior remain unaffected; the only change should be the addition of the `
data-testid`.
- Maintain logical consistency: Even for dynamically generated elements,
ensure that the addition of the `data-testid` does not impact JavaScript
logic or event handling.
```

"""Specific Prompt for HTML File Annotator"""

```
Please analyze the following HTML code and add `data-testid` attributes for
all necessary components based on the rules provided in the system content.

- If an element lacks a `data-testid`, generate one following the naming
convention.
- If an element already has a `data-testid`, rename it only if it does not
conform to the convention.
- Do not modify the structure, styles, or other attributes of the HTML.

Here is the HTML code to analyze:

**Original Code from {file_name}:**
{code}

**Modified Code:**
"""html
(Your modified HTML code here)
"""
```

"""Specific Prompt for Js File Annotator"""

```
Please analyze the following JavaScript code and ensure that all referenced
components have a proper `data-testid`, while preserving functionality.

- **Assign `data-testid` only if necessary:**
  - If an element is accessed in JavaScript, ensure it has a valid `data-
testid`.
  - **For dynamically created elements:**
    - **Single Instance**: If only one instance of a dynamically created
element exists at a time, assign a unique `data-testid`.
    - **Multiple Instances**: If multiple elements of the same type are
generated dynamically, use a unique identifier (e.g., `data-id`) instead of `
data-testid` to avoid conflicts.
- **Ensure consistency with HTML:** If an element's `data-testid` was
modified in HTML, update all references in JavaScript to match.
- **ABSOLUTELY NO CHANGES TO LOGIC OR STRUCTURE:** The script must function
**EXACTLY** the same way after modifications.

Here is the JavaScript code to analyze:

**Original Code from {file_name}:**
{code}

**Modified Code:**
"""javascript
(Your modified JavaScript code here)
"""
```

Code Analyzer

"""system prompt for html"""

```
You are an expert in analyzing HTML code from Web applications. Your task is
to extract UI elements and their attributes.

Your analysis should include:
1. List of all UI elements (buttons, input fields, links, etc.) with their `
id`, `class`, and role.
2. Any form-related elements and their expected interactions.
3. A concise summary of the UI structure.

Ensure your response is structured and clear, as this information will be
used by another agent to extract user requirements.
```

"""system prompt for js"""

```
You are an expert in analyzing JavaScript code from Web applications. Your
task is to extract event handlers, functions, and their relationships with UI
 elements.

Your analysis should include:
1. JavaScript functions that handle user interactions (e.g., `onclick`, `
onchange`).
2. The `id` or `class` of the elements these functions interact with.
3. A concise summary of how JavaScript controls the page's behavior.

Ensure your response is structured and clear, as this information will be
used by another agent to extract user requirements.
```

Requirement Extractor

"""system prompt"""

```
You are an expert in extracting **functional** user requirements from web
applications. Generate a **comprehensive and testable** list of user
requirements that cover all user-facing functionalities.

### Functional Requirement Criteria
Each requirement must include the following elements to ensure it is complete
 and testable:
1. **ID**: A unique identifier (e.g., REQ-001).
2. **Description**: A clear statement of the user requirement, including:
   - **Context**: The scenario or condition under which the functionality
occurs.
   - **User Action**: What the user does (e.g., clicks, types, scrolls).
   - **System Response**: The expected outcome after the user action.

### **Rules**:
- Only include **functional requirements**  i.e., observable behaviors
triggered by user interaction via the front end (such as clicking a button,
entering text, receiving visual feedback, etc.).
- **Avoid including non-functional requirements** such as performance,
security, or scalability unless they are visible or interactive on the UI.
- Exclude backend logic unless it has a **direct effect on the UI** that is
visible or interactive.

### **Output Format (JSON)**
{
  "summary": {
    "overview": "Briefly describe the application's purpose and key
functionalities.",
    "predefined_options": "Predefined options set by the system to
standardize inputs and reduce manual configuration, such as default values
and preset selections.",
    "external_resources": "External resources used by the application,
including links, images, audio files, and other media. List the resource
names and their sources (URLs or file paths).",
    "external_js_libraries": "External JavaScript libraries or packages used
by the application, such as jQuery, React, Bootstrap, etc. Provide the
library names and their sources (e.g., CDN links)."
}
,
  "requirements": [
    {
      "id": "Unique identifier",
      "description": "User requirement description with context, user action,
 and system response."

    }
  ]
}
```

Test Case Generator

"""system prompt"""

```
You are an expert in software testing. Your task is to generate **
comprehensive** Gherkin test cases based on the provided user requirement.

### **Instructions:**
1. **Mapping Requirements to Features**:
   - Each user requirement **must** be mapped to a corresponding `Feature`.
   - The `Feature` description should clearly summarize the purpose and scope
 of the requirement.

2. **Scenario Coverage**:
   - Each `Feature` must include multiple `Scenario` blocks covering:
     - **[Normal]** Expected behavior.
     - **[Edge]** Unusual or extreme conditions.
     - **[Error]** Invalid inputs or failures.
   - **Label each Scenario** with `[Normal]`, `[Edge]`, or `[Error]`.

3. **Gherkin Syntax  & Data Specificity**:**:
   - **All Given, When, Then steps must include explicit values if they are
known.**
      - If a value is dynamic or uncertain, describe its purpose instead of
using a placeholder.
      - Reference relevant UI elements (data-testid) for stable and precise
element identification.
      - Clearly define user interactions, specifying actions like clicks,
text input, or toggling switches.
      - State expected outcomes explicitly, verifying component properties
such as displayed text, input values.

   - **DO NOT generate structured tables** (e.g., `| Column | Value |`).
        - Instead, describe inputs and outputs directly in the step
definitions. For example:
       - Incorrect: Use a table to list inputs.
       - Correct: Write "When the user enters 'testuser' into the username
field with data-testid 'username-input'."
   - Each `Scenario` **must** follow the **Given-When-Then** syntax:
     - `Given`: Defines the initial context (UI components, form fields,
buttons, etc.) present in the application's HTML structure.
     - `When`: Specifies the user action (click, input, navigation) that is
linked to an actual event handler in the JavaScript code.
     - `Then`: States the expected outcome, ensuring it matches the UI
behavior as defined in the JavaScript logic.

4. **Scenario Independence & Page Initialization**:
   - Each `Scenario` **must** be **independent, complete, and executable on
its own**.
   - **Before any interaction, the test must ensure the correct webpage is
loaded.**

5. **Output Format**:
   - Wrap the entire Gherkin test cases in a single code block with the
language tag `gherkin`.
```

Test Automation Engineer

"""system prompt(Step Implementation)"""

```
You are an expert in implementing Selenium-based automated test scripts using
 Behave. Your task is to convert Gherkin test cases into Python step
implementations that adhere to the following rules:

1. **Step Definitions**:
   - Each `Given`, `When`, and `Then` step must have a corresponding `@given`,
 `@when`, or `@then` function.
```

```
   - **DO NOT MODIFY THE ORIGINAL STEP NAMES**: The text inside the
decorators must exactly match the Gherkin step descriptions.
   - If the Gherkin test case includes a `Background`, implement it first and
 ensure all `Scenario` steps reuse its setup without reinitializing `context`
 or `driver`.

2. **Selenium Best Practices**:
   1. Selector Usage:
   - Prioritize using data-testid attributes for locating elements.
     Example:
         driver.find_element(By.CSS_SELECTOR, "[data-testid='submit-button
']")
   - If data-testid is not available, use stable alternatives like class
names or IDs.
   - Avoid using fragile or overly complex XPath expressions unless necessary.


    2. User Interaction Handling:
       - Always wait for elements to be present and interactable before
performing actions.
       - Use WebDriverWait to ensure visibility or clickability.
         Example:
             WebDriverWait(driver, 10).until(EC.element_to_be_clickable((By.
CSS_SELECTOR, "[data-testid='submit-button']")))
       - Handle interactions like clicking, typing, and checking visibility
with proper error handling.

    3. Component State Checks:
   - To check if a component is expanded or collapsed:
       - Prefer checking the value of aria-expanded or state-indicative CSS
classes.
       - Check `data-*` attributes like `data-expanded`, or look at CSS
properties (e.g., display).
       - Define a helper function to check expansion state robustly:
         Example:
             def is_expanded(element):
                 # Check aria-expanded first
                 aria = element.get_attribute("aria-expanded")
                 if aria is not None:
                     return aria == "true"

                 # Check CSS class for expanded state
                 class_list = element.get_attribute("class").split()
                 if any(cls in class_list for cls in ["expanded", "open", "
show"]):
                     return True

                 # Check data-expanded attribute
                 data_expanded = element.get_attribute("data-expanded")
                 if data_expanded is not None:
                     return data_expanded == "true"

                 # Fallback: Use display property to check visibility
                 return element.is_displayed()

       - To check if a component is collapsed:
           - Collapse can typically be indicated by the absence of an "
expanded" class or an "aria-expanded" value of "false".
           - Example:
             def is_collapsed(element):
                aria = element.get_attribute("aria-expanded")
                if aria is not None and aria.lower() == "false":
                    return True

                class_attr = element.get_attribute("class") or ""
                class_list = class_attr.split()
                if "collapsed" in class_list:
                    return True
```

```
                data_expanded = element.get_attribute("data-expanded")
                if data_expanded is not None and data_expanded.lower() == "
false":
                    return True

                style = element.get_attribute("style") or ""
                if "display: none" in style or "visibility: hidden" in style
or "height: 0" in style:
                    return True

                return not element.is_displayed()


   - To check visibility:
       - Use element.is_displayed() to determine if an element is visible.
       - Alternatively, check visibility with JavaScript or CSS properties
like `visibility: hidden;` or `display: none;`.
       - You can also check for non-zero element size (`offsetWidth`, `
offsetHeight`).
         Example:
             is_visible = driver.execute_script("return arguments[0].
offsetWidth > 0 && arguments[0].offsetHeight > 0;", element)

   - To validate text content:
       - Use case-insensitive, partial match assertions.
         Example:
             expected_text = "submit"
             assert expected_text.lower() in element.text.lower(), f"Expected
 '{expected_text}' in '{element.text}'"
       - Consider dynamic content and validate after page updates.
       - Handle extra spaces or newline characters by trimming input.
         Example:
             assert expected_text.strip() in element.text.strip(), f"Expected
 '{expected_text}' in '{element.text}'"

   - To validate redirected URLs:
       - Strip off the hash (#) part before comparing.
         Example:
             base_url = driver.current_url.split("?")[0].split("#")[0]
             expected_base_url = expected_url.split("?")[0].split("#")[0]
             assert base_url == expected_base_url, f"Expected URL '{
expected_base_url}', but got '{base_url}'"

3. **Test Setup and Teardown**:
   - Load the test page from a local file using `file_path`.
   - Ensure the browser driver is properly initialized and closed at the end
of the test.
   - Include the placeholder `file_path = "file_path_placeholder"` in the
implementation for dynamic file path handling.

4. **Code Quality**:
   - Follow best practices for maintainability:
     - Use explicit waits (`WebDriverWait`) instead of implicit waits.
     - **After each interaction with a web element (e.g., `.click()`, `.
send_keys()`, `.get()`), insert `time.sleep(1)` to improve test robustness.**
     - Avoid hardcoding values such as URLs or element locators when possible.

     - Write clear and concise code with meaningful variable names.


5. **Output Format**:
   - Provide the corrected Python code wrapped in a code block with the
language tag `python`.
```

"""system prompt(Step Definition Fixer)"""

```
You are an AI assistant that helps users fix issues in Behave step
definitions (step.py).
Your task is to analyze the errors reported during a Behave dry run and
modify the code while adhering to the following rules:

1. **Step Definitions**:
   - Each `Given`, `When`, and `Then` step must have a corresponding `@given`,
 `@when`, or `@then` function.
   - Do not modify the content inside the decorators (e.g., step descriptions
).

2. **Error Analysis**:
   - Analyze the errors reported during the dry run. These errors typically
indicate missing step definitions, syntax issues, or other problems.
   - Ensure that all undefined steps are implemented correctly.

3. **Code Quality**:
   - Follow best practices for maintainability and robustness:
     - Use proper selectors (e.g., Selenium locators) where applicable.
     - Handle user interactions (clicking, inputting text, checking
visibility) correctly.
     - Avoid hardcoding values such as URLs or element locators when possible.


4. **Resource Management**:
   - Ensure the driver is closed at the end of the test if it was opened.

5. **Code Block**:
   - Provide the corrected Python code wrapped in a code block with the
language tag `python`.
```

"""system prompt(Step Logic Fixer)"""

```
You are an AI assistant that helps users fix issues in Behave step
definitions (step.py).
Your task is to analyze the failure logs and then modify the code while
adhering to the following rules:

1. **Do not alter the structure or framework of the code**:
   - Do not modify the content inside the `@given`, `@when`, or `@then`
decorators.
   - Ensure that the step definitions remain intact (e.g., function
signatures and decorator mappings).

2. **Focus only on fixing the implementation logic**:
   - Update the internal logic of the functions if there are errors or
missing parts.
   - Ensure the corrected code resolves the reported issues without altering
the intended behavior.

3. **Provide the corrected Python code in a code block**:
   - Wrap the corrected Python code in a code block with the language tag `
python`.
```

Test Runner Agent

"""Step Checker"""

```
def run_dry_run(self, project_root):
    """Run Behave in dry-run mode to verify that the definition of step.py is
 correct."""
    try:
       print("[TestRunnerAgent] Started Behave dry-run mode...")
        result = subprocess.run(
            [sys.executable, "-m", "behave", "--dry-run"],  # Using dry-run
mode
            cwd=project_root, # Make sure you run in the correct directory
```

```
            capture_output=True,
            text=True
        )

        # Print the output of dry-run
        print("[TestRunnerAgent] dry-run completed, the results are as
follows:")
        print(result.stdout)

        # Extract and analyze error information
        error_message = self.extract_error_info(result.stdout, result.stderr)

        # Determine if there is a real error
        if error_message:
            print(f"[TestRunnerAgent] dry-run failed with the following error
 message\n{error_message}")
            return error_message

        print("[TestRunnerAgent]  Dry-run succeeded! The definition of step.
py is complete and correct. ")
        return "No Faults"

    except Exception as e:
        print(f"[TestRunnerAgent] Failed to run Behave dry-run: {str(e)}")
        return f"Error: {str(e)}"
```

- Analyzing Step Definition Failures in Behave Dry-Run -

```
def extract_error_info(self, stdout, stderr):
    """
    Extract error messages from a dry-run, ignoring statistics and irrelevant
 content.
    """
    error_message = []

    # If stderr is not empty, the contents of stderr are used first.
    if stderr.strip():
        error_message.append("STDERR:")
        error_message.append(stderr.strip())

    # If stdout contains undefined steps or error messages
    if stdout.strip():
        lines = stdout.splitlines()
        for line in lines:
            # Filtering Statistics Rows
            if "steps passed" in line.lower() or "untested" in line.lower():
                continue

            # Check for undefined steps
            if "undefined" in line.lower() or "snippet" in line.lower():
                if "Undefined Steps Found:" not in error_message:
                    error_message.append("Undefined Steps Found:")
                error_message.append(line.strip())

    # If no error messages are found, return an empty list
    return "\n".join(error_message) if error_message else None
```

"""Test Runner"""

```
def run_tests(self, project_root, return_log=False):
    """Run the Behave command and make sure it is executed in the Conda
environment"""
    try:
        print("[TestRunnerAgent]  Starting to execute Behave tests...")
        result = subprocess.run(
            [sys.executable, "-m", "behave"],  # Run via Conda interpreter
            cwd=project_root,  # Make sure you are running in the correct
directory
            capture_output=True,
```

```
            text=True
        )

        print("[TestRunnerAgent] The test is completed, the results are as
follows:")
        print(result.stdout)
        if return_log:
            # if result.returncode != 0:
            #     print("[TestRunnerAgent] Test failed, error message is as
            #     print(result.stderr)
            return result.stdout

        if result.returncode != 0:
            print("[TestRunnerAgent] Test failed, error message is as follows:
 ")
            print(result.stderr)
            return result.stderr  # Returns error information for
StepFixAgent to handle

        return "No Faults"
        print(f"[TestRunnerAgent] Failed to run Behave: {str(e)}")

    except Exception as e:
        return f"Error: {str(e)}"
```

- Failure Analysis: Test Logic Errors in Behave -

```
You are an expert in analyzing Behave test logs. Your task is to extract and
summarize errors from Behave test execution results.

### **Instructions:**
1. Identify failed scenarios in the log.
2. Extract the specific step that failed.
3. Identify and summarize the error message.
4. Return the results in a structured format.

### **Output Format:**
{
  "failed_scenarios": [
    {
      "scenario": "Scenario Name",
      "failed_step": "Step that caused the failure",
      "error_message": "Summarized error message"
    },
    ...
  ]
}
Ensure accuracy and completeness in summarizing the errors.
```

H LLMs Usage Statement

LLMs have become helpful tools for efficiently processing data in many recent works. In line with this practice, we used LLMs to accelerate parts of our annotation process, while all annotations were carefully validated by humans (see Appendix A). Importantly, **all conceptual work and method design were conducted solely by the authors. LLMs were used only to polish the writing of this paper and did not contribute to the benchmark design, evaluation methods, analysis, or any core ideas of our work**.