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

Jingyao Liu[1,4] **Chen Huang**[2*] **Zhizhao Guan**[1,4] **Wenqiang Lei**[1,4] **Yang Deng**[3]
[1] College of Computer Science, Sichuan University
[2] Institute of Data Science, National University of Singapore
[3] CHAT NLP Group, Singapore Management University
[4] Engineering Research Center of Machine Learning and Industry Intelligence, Ministry of Education, China
liujingyao1@stu.scu.edu.cn, huang_chen@nus.edu.sg

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) to assess whether the generated software meets user needs through mimicking real user interactions. E2EDev comprises (i) a fine-grained set of user requirements for each target software project (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. 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 available at https://github.com/SCUNLP/E2EDev.

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; Zhang et al., 2025a). 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 (Zhang et al., 2025b; Zhong et al., 2024; Yang et al., 2025; Li et al., 2022; Yang et al., 2024). 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 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 one 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 Figure 1 (left), the vague requirement "manage words" for a dictionary tool can refer to various actions, such as editing existing entries or bookmarking words. 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** bench-

*Corresponding author.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 35032–35068

Figure 1: Comparing existing work with ours, existing benchmarks use coarse-grained requirements with unreliable evaluations whereas E2EDev provides fine-grained requirements with executable BDD tests.

mark 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 Figure 1 (right). 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 evaluation of various E2ESD frameworks and LLM backbones reveals a significant struggle to effectively solve E2ESD tasks. Even the Claude series, among the strongest coding-oriented LLMs, fails to exceed 60% performance. These failures stem from the difficulty of simultaneously performing high-level planning (i.e., deciding what to build) and faithfully following fine-grained functional details. In multi-agent staged pipelines, this challenge is often addressed by explicitly separating planning from detailed implementation across different phases. However, the effectiveness of such designs critically depends on how intermediate information is communicated. Excessive context sharing tends to dilute key information, whereas overly restricted communication weakens the influence of earlier design choices on later implementation, resulting in high interaction costs with limited performance gains. We believe this benchmark underscores the need for more effective and cost-efficient E2ESD framework design.

2 Related Work

Benchmarks for Software Development. Table 1 summarizes representative benchmarks for E2ESD. These benchmarks, which involve project-level code generation, are scarce due to the inherent complexity of their construction, often resulting in limited dataset sizes (Hong et al., 2024; Hu et al., 2025). They typically face two primary limitations. First, vague or underspecified textual specifications (Qian et al., 2024) make it difficult to verify whether the generated software aligns with user requirements. Second, a reliance on subjective human evaluation or heuristic metrics lacks grounding in established software development principles (Hong et al., 2024; Qian et al., 2024), thereby hindering reliable performance assessment.

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

Benchmark	# Tasks	# Reqs	Test Type / Scale	Source	Key Features
SoftwareDev (Hong et al., 2024)	70	Amb.	✗ Human Eval.	Real	Private benchmark; human ratings only
SRDD (Qian et al., 2024)	1,200	Amb.	✗ Heuristics	Gen.	Heuristic indicators; no execution
GitTaskBench (Ni et al., 2025)	18	54	✓ Repo Ops.	Real	Repo operation execution
Mle-Bench (Chan et al., 2024)	72	72	✓ Pipeline Exec.	Real	Machine Learning pipeline execution
rSDE-Bench (Hu et al., 2025)	53	Amb.	✓ 616 Unit Tests	Gen.	Function-level output checks
E2EDev	**46**	**244**	✓ **703 BDD Tests**	**Real**	**Requirement Verification via BDD**

Table 1: Comparison of Repo-Level Code Generation Benchmarks. **Gen.**: LLM-Generated. **Real**: Real-World Repositories. **Amb.**: Ambiguous or unspecified number of requirements.

		Max	Min	Mean	Total
Repo.	Prompt Len.	1907	206	712.7	**46**
Req.	Reqs / Repo	11	2	5.3	**244**
	Words / Req.	113	20	64.1	
Tests	Tests / Req.	7	1	2.9	**703**
	Tests / Repo.	34	2	15.3	
	Gherkin Scenarios				
	Lines / Scen.	30	7	11.2	
	Words / Scen.	360	53	109.0	
	Step Definitions				
	Lines / Def.	157	13	61.4	
	Words / Def.	700	74	217.4	

Table 2: Benchmark Statistics

Although some benchmarks incorporate executable evaluation, they are typically confined to specific commands (Ni et al., 2025), workflows (Chan et al., 2024), or individual functions (Hu et al., 2025), failing to verify whether the overall system behavior aligns with the user's specific intent.

LLM-based Frameworks for E2ESD. Most frameworks employ a multi-agent paradigm, decomposing the development process into explicit subtasks following established software engineering workflows (Sommerville, 2011; Hong et al., 2024; Du et al., 2024; Rasheed et al., 2024; Sami et al., 2024). These workflows can be either predefined (Dong et al., 2024; Islam et al., 2024) or adaptively constructed during interaction (Qian et al., 2024; Lin et al., 2024). In contrast, some work employs single-agent frameworks (Engineer, 2024), where a single LLM manages the entire development process 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.

3.1 Project Construction

Project Selection. Focusing on Web applications, we selected 46 highly starred GitHub projects that are executable, functionally complete, and suitable for rapid evaluation (details in Appendix A.1).

Features. These projects cover a broad range of Web application types, such as calculators and mini games, and support diverse user interactions, such as typing, clicking, and dragging (cf. Figure 7, Appendix A.4). They also include technically complex features, ranging from simple DOM manipulation to more advanced tasks such as local storage usage and external API integration. Overall, the diversity, technical depth, and realism of these projects make them a qualified and representative ground for evaluating E2E-REP methods.

3.2 HITL-MAA Annotation Framework

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 2, 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

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

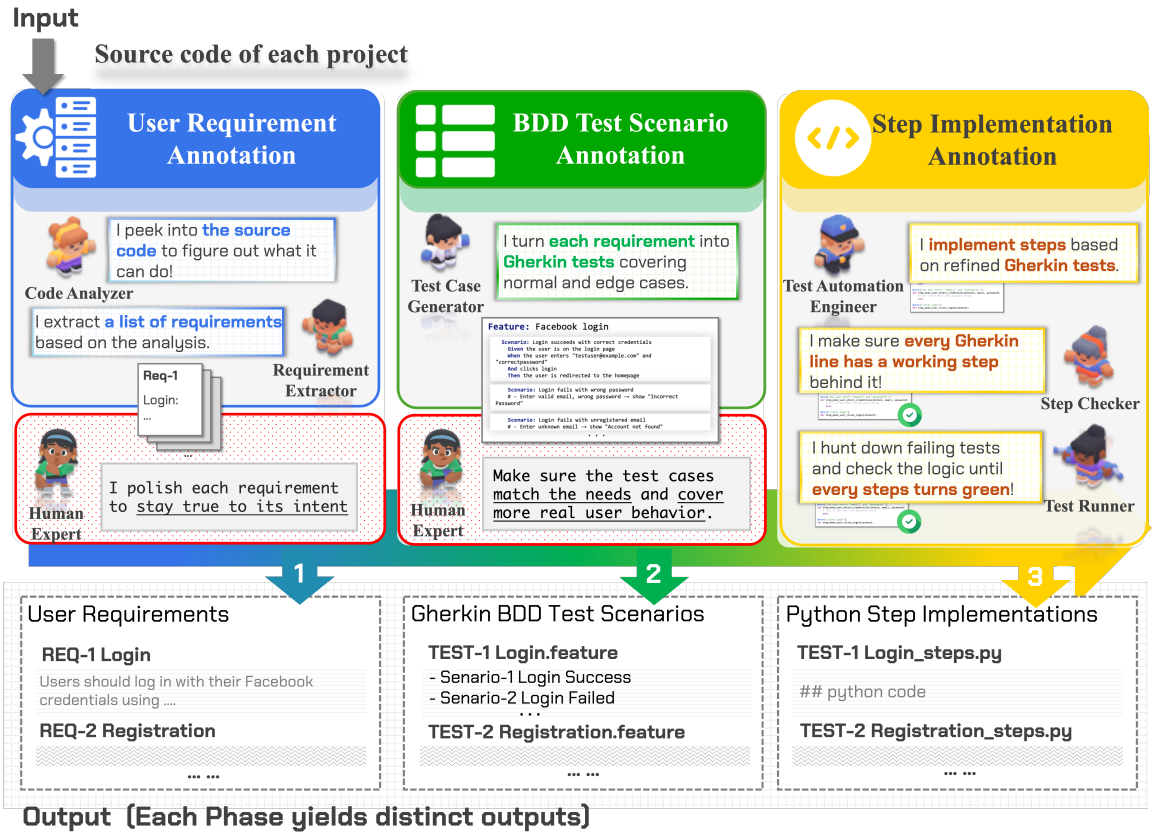

Figure 2: HITL-MAA framework for semi-automated dataset construction. Given source code, the framework extracts user requirements and generates corresponding BDD test scenarios (Gherkin format) along with Python step implementations, with human verification or agent-based refinement at each stage to ensure dataset quality.

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. This process transforms a source project into a list of fine-grained user requirements through the collaborative effort of two LLM agents under human supervision. (1) The Code Analyzer Agent analyzes the project's core functionalities and their interactions with UI elements by reading the source code. (2) The Requirement Extractor Agent then automatically generates candidate user-facing requirements based on this analysis. Human supervisors review the generated requirements to ensure accuracy, clarity, and functional consistency by verifying them against the actual system behavior.

BDD Test Scenario Annotation. For each human-validated requirement, HITL-MAA generates a set of Gherkin-style BDD test scenarios. Each scenario describes a specific condition or user interaction to be tested and follows the Given–When–Then format, clearly specifying 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 both validated requirements and source code to produce relevant scenarios. The generation process is guided to cover a wide range of test cases, including both typical behaviors and unexpected or edge cases. Since the agent relies solely on source code—which does not explicitly capture all user interaction patterns—some unconventional scenarios may be overlooked and require supplementation by human experts. To ensure comprehensive coverage and high quality, five software testing experts collaboratively review and refine the generated scenarios. Meanwhile, HITL-MAA incorporates a final expert review step to resolve any remaining inconsistencies between requirements and test cases, guaranteeing reliable and high-quality annotations.

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 implementations corresponding to the Gherkin-style test steps. The generated scripts then undergo an iterative verification process to ensure both executability and correctness. Two 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.

Automated Evaluation. E2EDev adopts the behave framework for fully automated evaluation. Gherkin scenarios are executed through their corresponding Python step implementations to test the generated project against defined requirements, enabling deterministic pass/fail verification of model-generated code. (Details in Appendix D.3.)

4 Experiment

This section enables automated benchmark analysis. E2EDev is an automated testing benchmark suite, accessible even to researchers less familiar with software testing procedures[3].

4.1 Experiment Setup

Baselines and LLM Backbones. We consider a representative set of E2ESD methods, covering three categories: (1) *Vanilla LLM*, which directly generates an executable project via prompting; (2) *Single-agent frameworks*, represented by *GPT-Engineer* (Engineer, 2024), where a single LLM constructs the entire project in a single reasoning process; and (3) *Multi-agent frameworks*, including *Self-Collaboration* (Dong et al., 2024), *MetaGPT* (Hong et al., 2024), *MapCoder* (Islam et al., 2024), and *ChatDev* (Qian et al., 2024). We additionally include two widely used coding agents, *OpenHands* (Wang et al., 2024) and *SWE-Agent* (Yang et al., 2024), as reference systems, with results reported in Appendix C.1 since they target a different task setting from E2EDev. To disentangle the effects of framework design from model capacity, we run all baselines across diverse LLM backbones with varying scales and architectures (Hurst et al., 2024; Hui et al., 2024), including Claude-Haiku 4.5, GPT-4o, GPT-4o-mini, Qwen2.5-7B, Qwen2.5-70B, and the mixture-of-experts Qwen2.5-Max. Implementation details are provided in Appendix B.1.

Evaluation Metrics. For comprehensive evaluation, we assess **code effectiveness** (how well the generated code meets user requirements) and **generation efficiency** (the cost and time of code generation). Effectiveness is averaged over three runs per task, while efficiency is measured from a single run due to negligible variance. For **code effectiveness**, we evaluate how well a generated project satisfies requirements at two levels. (1) Req. Acc measures the fraction of requirements that are fully satisfied across a project, i.e., all test cases associated with a requirement pass; (2) Test Acc measures the fraction of all test cases that pass across a project; (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 **per project**; (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.

Limited effectiveness: While existing methods meet broad project requirements, their performance remains below acceptable standards due to limited proficiency in handling specifics. Off-the-shelf models excel at function-level tasks (e.g., HumanEval (Chen et al., 2021) in Appendix C.4) but underperform on E2ESD, achieving only 30%–50% Req. Acc. on the proposed E2EDev dataset (Table 3), with even advanced models like Claude-Haiku 4.5 and GPT-4o failing to exceed 60%. Furthermore, Soft Req. Acc., which allow partial fulfillment, suggest failures stem from their lack of proficiency in handling detailed functional specifics during implementation. This phenomenon, visualized in Figure 3, persists across

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

Backbone	Method	Effectiveness (↑, %)			Efficiency (↓)		
		Req. Acc.	Test Acc	Bal. Score	Cost ($)	CO_2 (g)	Time (s)
Claude-Haiku 4.5	Vanilla LLM	48.69 ±1.30	63.08 ±0.50	54.45 ±0.98	0.015	0.028	53
	GPT-Engineer	**53.75** ±0.43	**69.41** ±1.07	**60.01** ±0.69	0.016	0.038	59
	Self-Collab.	49.01 ±1.74	61.50 ±1.26	54.01 ±1.55	0.014	0.033	75
	MapCoder	49.61 ±0.50	65.65 ±0.44	56.03 ±0.48	0.099	0.081	129
	ChatDev	44.73 ±1.50	58.09 ±1.03	50.07 ±1.31	0.174	0.188	335
	MetaGPT	5.39 ±0.76	10.66 ±1.77	7.50 ±1.16	0.090	0.091	366
GPT-4o	Vanilla LLM	45.95 ±1.25	60.88 ±1.97	51.92 ±1.54	0.0160	0.083	28
	GPT-Engineer	**50.83** ±1.33	**66.59** ±2.43	**57.13** ±1.77	0.0198	0.132	21
	Self-Collab.	46.83 ±1.12	61.15 ±1.47	52.56 ±1.26	0.0155	0.109	37
	MapCoder	47.70 ±2.57	63.97 ±2.25	54.21 ±2.44	0.1091	0.750	93
	ChatDev	42.71 ±2.44	58.93 ±2.66	49.20 ±2.53	0.1947	1.910	114
	MetaGPT	0.00 ±0.00	0.17 ±0.05	0.07 ±0.02	0.0951	0.794	66
GPT-4o-mini	Vanilla LLM	**44.82** ±2.01	**60.65** ±2.49	**51.15** ±2.20	0.0010	0.003	16
	GPT-Engineer	42.13 ±1.81	57.51 ±1.98	48.28 ±1.88	0.0012	0.005	18
	Self-Collab.	37.90 ±1.92	52.99 ±2.51	43.94 ±2.16	0.0009	0.004	25
	MapCoder	41.30 ±1.49	57.87 ±0.85	47.93 ±1.23	0.0072	0.033	88
	ChatDev	33.16 ±2.71	48.40 ±2.75	39.26 ±2.73	0.0118	0.071	157
	MetaGPT	0.00 ±0.00	0.23 ±0.05	0.09 ±0.02	0.0067	0.040	63
Qwen-Max	Vanilla LLM	43.33 ±0.66	58.57 ±0.73	49.43 ±0.69	0.0025	0.043	53
	GPT-Engineer	**49.61** ±0.60	63.70 ±1.41	**55.25** ±0.92	0.0030	0.067	66
	Self-Collab.	42.30 ±1.88	60.44 ±1.33	49.56 ±1.66	0.0022	0.053	75
	MapCoder	48.83 ±1.68	**64.00** ±1.82	54.90 ±1.74	0.0186	0.424	366
	ChatDev	43.93 ±1.94	59.44 ±0.99	50.13 ±1.56	0.0249	0.790	300
	MetaGPT	1.65 ±0.25	2.80 ±0.58	2.11 ±0.38	0.0207	0.544	312
Qwen-70B	Vanilla LLM	35.75 ±0.00	53.14 ±0.00	42.71 ±0.00	0.0029	0.050	39
	GPT-Engineer	42.08 ±0.00	**58.78** ±0.00	48.76 ±0.00	0.0037	0.080	45
	Self-Collab.	**42.61** ±0.06	55.63 ±0.10	47.82 ±0.08	0.0030	0.066	69
	MapCoder	40.59 ±0.26	57.49 ±0.07	47.35 ±0.18	0.0293	0.624	294
	ChatDev	43.15 ±0.51	57.62 ±0.08	**48.94** ±0.34	0.0387	1.006	341
	MetaGPT	0.00 ±0.00	1.73 ±0.06	0.69 ±0.02	0.0290	0.786	151
Qwen-7B	Vanilla LLM	22.37 ±0.00	34.75 ±0.00	27.32 ±0.00	0.0003	0.005	32
	GPT-Engineer	**24.03** ±0.00	**39.94** ±0.00	**30.39** ±0.00	0.0003	0.007	31
	Self-Collab.	20.65 ±1.00	33.37 ±0.63	25.74 ±0.85	0.0003	0.006	52
	MapCoder	11.90 ±1.61	26.96 ±1.12	17.92 ±1.41	0.0024	0.049	236
	ChatDev	10.96 ±1.73	21.35 ±2.10	15.12 ±1.88	0.0075	0.187	344
	MetaGPT	0.00 ±0.00	1.98 ±0.28	0.79 ±0.11	0.0207	0.095	301

Table 3: Benchmark analysis of effectiveness and efficiency. The grey row denotes the Vanilla baseline. Red and green indicate performance better and worse than Vanilla, respectively. Standard deviations are shown in small grey text. For effectiveness, the best result is shown in **bold** and the second-best is underlined.

different LLM backbones and E2ESD frameworks. The observed discrepancy of over 25% in both metrics suggests that while models can implement the required functionality, they often fail to address complex edge cases. Also, this issue is reflected in ∼10% performance gap observed on HumanEval and HumanEval-ET (cf. Appendix C.4).

Limited efficiency: Multi-agent methods exhibit excessive interaction rounds and token costs with marginal 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 (≤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.3.

Agentic frameworks do not consistently improve the effectiveness of base LLMs. While agentic frameworks offer the potential for performance gains, their inherent complexity imposes significant demands on the foundational capabilities of the underlying LLM. Figure 4 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 co-

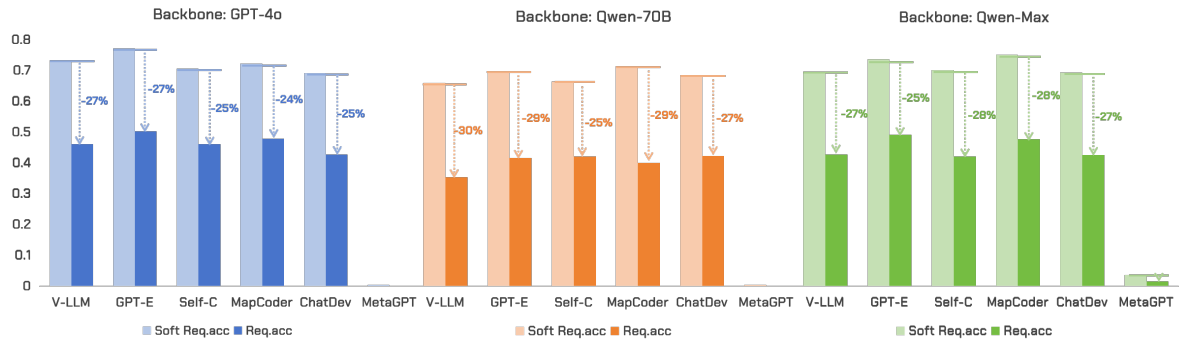

Figure 3: Soft-Requirement and Requirement Accuracy for Vanilla LLM (V-LLM), GPT-Engineer (GPT-E), and Self-Collaboration (Self-C). Additional results for other backbones are in Appendix C.2, Table 10.

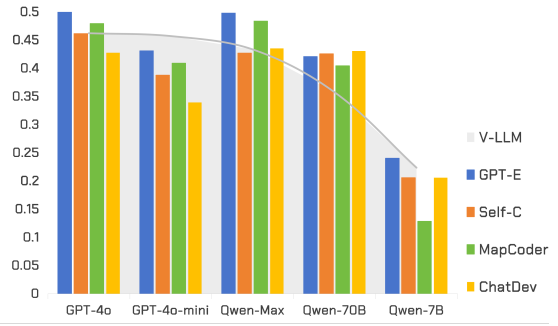

Figure 4: Comparisons across agentic frameworks and the Vanilla LLM (on Req. Acc.).

ordination 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. Overall, our results indicate that current frameworks rely heavily on strong base LLMs, suggesting that future designs should reduce this reliance to improve robustness across diverse LLM backbones.

MetaGPT suffers from communication breakdowns in its multi-agent architecture. Even at the function level, we observe evidence of communication failures on HumanEval using the official implementation (Appendix C.4), which already limits its performance. At the project level, the same issue becomes more severe, causing MetaGPT to fail on nearly all test cases and requirements, even with strong LLMs (see Section 4.3).

4.3 Failure Mode Analysis

Setup. To understand underlying failure causes, we conducted a rigorous human evaluation on 360 projects (randomly sampled from 10 entries across 6 LLMs and 6 frameworks). Four domain experts evaluate each project along four dimensions: *Code Inconsistency* (e.g., missing, conflicting, or empty functions), *Requirement Missing* (required features not implemented), *Requirement Misaligned* (implementation logic deviates from the requirements), and *Detail Mismatch* (mostly correct but with minor errors). Figure 5 presents the distribution of failure modes for existing frameworks on selected LLM backbones (GPT-4o, Qwen2.5-70B, and Qwen-Max). Detailed results and case studies are in Appendices C.2 and C.5. Human evaluation protocol and agreement are in Appendix B.3.

Model	Method	$\text{Tok}_{\text{prompt}}$	$\text{Tok}_{\text{compl.}}$	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

Table 4: Statistics of interaction overhead across representative frameworks. Turns = Dialogue Turns

Analysis of Code Inconsistency – Vanilla LLMs generally maintain high code consistency, whereas this consistency often degrades under multi-agent settings. This degradation is largely caused by exposure to excessive or irrelevant context. For instance, MapCoder employs analogical prompting (Yasunaga et al., 2024), retrieving prior projects from memory for few-shot guidance. However, these exemplars often share only a coarse domain similarity (e.g., web applications) rather than functional relevance, introducing noise instead of actionable guidance. In contrast, frameworks that restrict coder inputs to task-specific requirements and component analyses effectively reduce interference, leading to more stable code generation.

Analysis of Requirement Missing – Flawed

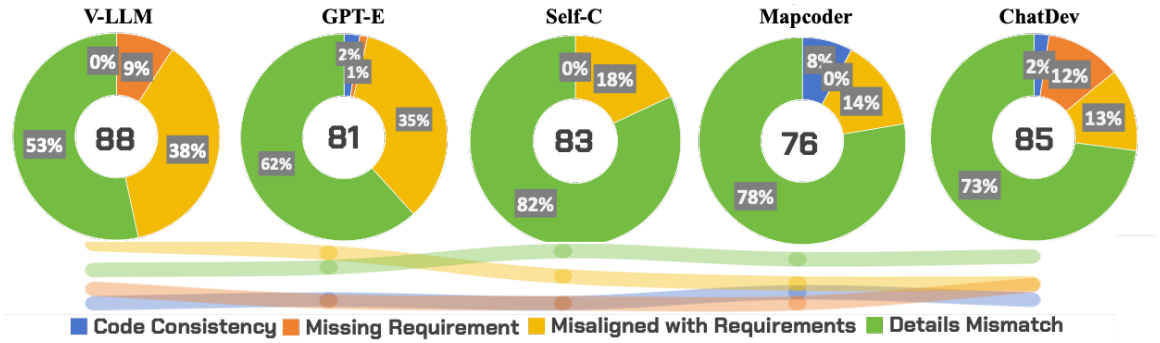

Figure 5: Requirement-level error distribution across frameworks. Pie charts show error proportions (centers indicate total errors), and trend lines report absolute counts across frameworks.

multi-agent designs can lead to overlooked requirements. As shown in Figure 5, *Vanilla LLM* and *ChatDev*—the only methods without direct access to requirement analysis during coding[4]—exhibit the most severe requirement omissions. In ChatDev, although a requirement analysis agent exists, the coding agent only receives high-level guidance from the Executive Officer, without access to detailed requirement specifications. This limited visibility substantially increases the risk of missing essential requirements, highlighting the necessity of exposing coding agents to explicit requirement analyses.

Analysis of Requirement Misalignment – Core component analysis helps align code with user requirements. As shown in Figure 5, multi-agent frameworks significantly reduce requirement misalignment. This benefit mainly arises from identifying key structural and functional components, such as HTML layout and JavaScript logic, guiding the coding agent toward user-specified logic and structure. Thus, the generated code more faithfully reflects the intended functionality.

Analysis of Detail Mismatch – LLMs struggle with fine-grained detail control, which is exacerbated in multi-agent systems due to longer contextual chains. Figure 5 shows that detail mismatches occur across all frameworks, with Vanilla LLM exhibiting the fewest and Self-Collaboration the most. We group these errors into three categories: (1) *Unhandled logical edge cases*, which often span multiple modules and complex state dependencies and remain difficult for all frameworks; (2) *Missing validation or error handling*; and (3) *UI display inconsistencies*. The latter two mainly result from progressive dilution of detailed

[4]GPT-Engineer implicitly performs requirement analysis via a "think step by step" process.

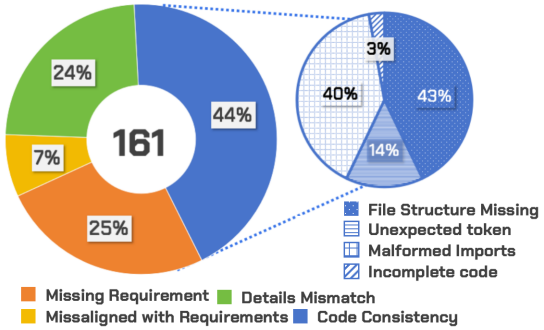

Figure 6: Error distribution of MetaGPT

constraints as requirements pass through multiple agents that prioritize high-level functionality. Frameworks such as MapCoder and ChatDev partially mitigate this issue by repeatedly prepending original requirements and prior messages, helping preserve critical details.

Analysis of MetaGPT – Inter-agent communication breakdowns impair code consistency and requirement adherence. As shown in Figure 6, 44% of MetaGPT failures arise from code inconsistency, including missing files and syntax errors. Over 43% of these occur when the programmer agent ignores the architect's prescribed file structure. In addition, 54% of syntax errors stem from malformed imports or unmatched tokens, likely because the engineer agent is simultaneously responsible for code generation and tool invocation, with tool-related prompts outweighing coding instructions by over 10×. Moreover, 25% of failures are caused by missing requirements, and 31% by incomplete or inconsistent implementations. This is partly due to the product manager repeatedly rewriting or compressing the original requirements, rather than restating them as instructed. Similar behavior is observed on HumanEval, where MetaGPT with GPT-4o achieves only ∼50% pass rate, and

nearly 30% of outputs fail to match required function names. Overall, these results indicate that excessive agent decomposition and communication overhead can undermine both specification adherence and coding reliability.

5 Conclusion

Our E2EDev represents a pivotal step toward standardizing the evaluation of E2ESD frameworks by providing detailed requirement specifications and reliable evaluation protocols. While LLMs have shown strong capabilities in isolated engineering tasks, our work highlights the significant challenges remaining in achieving fully automated E2ESD, constrained by both inherent model limitations and communication inefficiencies. Consequently, future work should prioritize designing effective, reliable, and cost-efficient E2ESD solutions that can fully realize the potential of LLMs in automating software development.

Limitations

Dataset Scale and Construction Complexity. Unlike function-level tasks, repository-level coding tasks provide a more comprehensive assessment of LLMs' capabilities in project construction and requirement fulfillment. However, constructing robust repository-level benchmarks remains challenging. A primary difficulty arises from the diverse and often idiosyncratic implementations that LLMs generate for identical user requirements, which substantially increases the cost and complexity of reliable automated evaluation. As a result, existing repository-level benchmarks, including E2EDev, are necessarily limited in scale (Table 1).

Scope Validity and Evaluation Protocol. Our strategic focus on Web applications is grounded in three methodological imperatives. First, this domain serves as a necessary "lower bound" for assessing E2ESD capabilities; the failures observed in this relatively deployable scenario reflect intrinsic limitations in LLM reasoning or framework design rather than domain-specific artifacts, suggesting that models struggling here are unlikely to succeed in more complex system-level settings. Second, to ensure rigorous reproducibility, we exclude CI/CD workflows and deep backend inspections, which rely heavily on volatile environmental dependencies that hinder consistent measurement. Third, aligned with Requirements Engineering standards (ISO/IEC/IEEE 29148), we adopt a black-box testing approach that verifies system correctness through user-perceptible behaviors via browser-based automation. This protocol allows us to validate whether the integrated system truly satisfies user intent without the noise of environment-specific internal implementation details.

Despite these limitations, our work serves as a reasonable starting point for more rigorous evaluation of E2ESD tasks. E2EDev contributes a standardized benchmark by providing explicit requirement specifications and reliable evaluation protocols, enabling systematic analysis of requirement fidelity and code consistency. While LLMs perform well on function-level and isolated programming tasks, our results reveal substantial gaps in scaling these capabilities to fully automated E2ESD settings. Given the significant resource investment required for benchmark construction, we plan to continuously expand E2EDev and maintain a public leaderboard to support reproducible and longitudinal evaluation.

Ethical Considerations

Our work focuses solely on constructing the E2EDev benchmark for end-to-end requirement-to-executable code generation 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. Human annotators were internal CS Master's and PhD students, with priority given to those who had taken software testing or development courses. They were invited via email, participation was voluntary and unpaid, and written instructions explained the tasks and their intended use. Details are provided in Appendix A. All experiments and analyses adhere to the ACL Code of Ethics, and no procedures, datasets, or methods in this study raise concerns related to privacy, bias, or safety.

Reproducibility To facilitate reproducibility, we publicly release the full E2EDev benchmark source code on GitHub (https://github.com/SCUNLP/E2EDev), along with the complete benchmark dataset on Hugging Face (https://huggingface.co/datasets/GuanZhiZhao/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.

Artifacts License and Usage Terms All code and data used and created in this work comply with their respective licenses and usage terms. External code and datasets are open-source or publicly available, and their licenses (e.g., CC-BY 4.0 for data, MIT for code) are respected, including proper attribution to the original creators. The E2EDev benchmark we create is released under CC-BY 4.0 for data and MIT for code, and is intended for research purposes.

LLMs Usage 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.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (No. U25B201508, No. 62272330, and No.U24A20328); in part by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant (Proposal ID: 24-SIS-SMU-002).

References

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

Cognition AI. 2024. Introducing devin, the first ai software engineer.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, and 1 others. 2024. Mle-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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

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

GPT Engineer. 2024. Gpt engineer. Accessed: 2024-12-27.

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

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

Lewis Hammond, Alan Chan, Jesse Clifton, Jason Hoelscher-Obermaier, Akbir Khan, Euan McLean, Chandler Smith, Wolfram Barfuss, Jakob Foerster, Tomáš Gavenčiak, and 1 others. 2025. Multi-agent risks from advanced ai. *arXiv preprint arXiv:2502.14143*.

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

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

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, and 1 others. 2024. Metagpt: Meta programming for a multi-agent collaborative framework. In *ICLR*.

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

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. 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)*, pages 4912–4944.

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

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

Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and 1 others. 2022. 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*, pages 1035–1047.

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

Ziyi Ni, Huacan Wang, Shuo Zhang, Shuo Lu, Ziyang He, Wang You, Zhenheng Tang, Yuntao Du, Bill Sun, Hongzhang Liu, and 1 others. 2025. Gittaskbench: A benchmark for code agents solving real-world tasks through code repository leveraging. *arXiv preprint arXiv:2508.18993*.

Melissa Z Pan, Mert Cemri, Lakshya A Agrawal, Shuyi Yang, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Kannan Ramchandran, Dan Klein, and 1 others. 2025. Why do multiagent systems fail? In *ICLR 2025 Workshop on Building Trust in Language Models and Applications*.

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

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

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

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

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

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

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

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

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

Xinwei Yang, Zhaofeng Liu, Chen Huang, Jiashuai Zhang, Tong Zhang, Yifan Zhang, and Wenqiang Lei. 2025. ELABORATION: A comprehensive benchmark on human-LLM competitive programming. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 59–104, Vienna, Austria. Association for Computational Linguistics.

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

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

Yifan Zhang, Chen Huang, Yueke Zhang, Huajie Shao, Kevin Leach, and Yu Huang. 2025a. Pre-training representations of binary code using contrastive learning. *Preprint*, arXiv:2210.05102.

Yifan Zhang, Chen Huang, Yueke Zhang, Jiahao Zhang, Toby Jia-Jun Li, Collin McMillan, Kevin Leach, and Yu Huang. 2025b. Eyemulator: Improving code language models by mimicking human visual attention. *Preprint*, arXiv:2508.16771.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. 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*, pages 851–870.

A Details on E2EDev Benchmark

A.1 Details on Human Verification for Project Selection

We select source projects through a two-step process. First, we crawl GitHub repositories using keywords such as "mini project", "software project", "web application", and "H5", retaining only those with more than 500 stars.

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:

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:** ...

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 */;
```

```
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 free-text requirements and test case validation, we implement a *post-hoc gold evaluation* process. Five human annotators with expertise in software testing participate, with one senior expert designated as the *Gold Checker*, responsible for reviewing others' annotations. This process employs an iterative annotation and feedback mechanism to resolve inconsistencies efficiently. For both requirements and test cases, four regular annotators first perform validation. For requirement annotations, annotators interact with the web application through a browser to directly experience and verify functionality. For test cases, written in natural language as step-by-step operational guides, annotators follow the instructions to ensure consistency and validity against the requirements. Detected discrepancies or errors are revised before submission to the Gold Checker. The Gold Checker evaluates each annotation on three binary criteria: *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, and annotators repeat the annotation process. Newly proposed test cases also undergo gold evaluation. In such cases, all three criteria are initially scored as 0 by the annotators, and the Gold Checker reviews the new case before returning it to all regular annotators for reconsideration and re-annotation. If annotators fail to recognize a new test case, the Gold Checker similarly returns it for re-annotation. This approach 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. It ensures high-quality, consistent annotations for downstream applications.

Reliability Audit & Scalability Verification. To further mitigate potential bias from the initial team size and validate the generalizability of our quality standards, we conducted an extended *reliability audit*. We expanded the expert pool to **10 evaluators** (adding five distinct external experts to the original group) and performed a blind review on a stratified random sample of **100 test cases** from the final dataset. This broader panel independently assessed the sample under the same strict criteria. The resulting inter-rater agreement yielded a **Fleiss' Kappa of 0.81**, demonstrating that our original 5-expert protocol aligns with the consensus of a larger expert body and ensuring the dataset's high quality is statistically robust.

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-

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

Input: Source code annotated with `test-ids`
Output: 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. */;
```

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.

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
```

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	≫ 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

Table 5: Comparison of Manual vs. Semi-Automated Annotation

```
        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 150
    ms 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 7(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 7(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

We select a diverse set of representative **requirement-to-executable software development** 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)*. We additionally include two widely used coding

Figure 7: 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.

agents, *OpenHands* (Wang et al., 2024) and *SWE-Agent* (Yang et al., 2024), as reference systems. These agents are designed for general-purpose software engineering scenarios with extensive tool and environment interaction, which differ from the requirement-to-executable software generation setting considered in E2EDev. Their results are reported in the Appendix C.1.

Finally, we adopt a range of backbones from two vendors (Hurst et al., 2024; Hui et al., 2024), covering various scales and architectures: Claude-Haiku 4.5, GPT-4o, GPT-4o-mini, Qwen-7B (Qwen2.5-7B-Instruct), Qwen-70B (Qwen2.5-72B-Instruct), and the mixture-of-experts Qwen-Max (Qwen2.5-Max). This enables controlled analysis of framework versus model backbone effects. Implementation details are presented below.

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

Vanilla LLM. We directly fed the task prompts (i.e., user requirements) along with the following structured instruction:

```
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 ...
"""
```

GPT-Engineer. GPT-Engineer builds a complete codebase in a single reasoning pass with minimal external coordination, maintaining a modular workflow. We used the official Python packages and executed the frameworks directly with task prompts, without modifying internal workflows.

Self-Collaboration Code Generation. This framework incorporates software-development methodology by assembling a team of three LLM roles—analyst, coder, and tester—responsible for analysis, coding, and testing. Originally designed for Python-only tasks, we adapted it to our multi-language (HTML/CSS/JavaScript) setting by modifying the system prompts. Specifically, we replaced the default PYTHON_DEVELOPER role with an H5_DEVELOPER role:

```
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 ...
"""
```

MapCoder. MapCoder consists of four agents simulating the stages of software development: recalling examples, planning, code generation, and

debugging. Similar to Self-Collaboration, it was originally Python-focused; we adapted it to support HTML/CSS/JavaScript by modifying its agent system prompts accordingly.

ChatDev. ChatDev is a chat-powered software development framework where LLM-driven agents communicate via structured chat chains. Agents contribute to design, coding, and testing phases through multi-turn dialogues, guided by communicative strategies to reduce hallucinations. We implemented it using the official GitHub source without modifications.

MetaGPT. MetaGPT is a meta-programming framework that incorporates human workflow principles into LLM-based multi-agent collaboration. It encodes Standardized Operating Procedures (SOPs) into prompt sequences, enabling agents with domain expertise to verify intermediate results and reduce errors. Using an assembly-line paradigm, MetaGPT assigns diverse roles to multiple agents to efficiently decompose complex tasks. We used the official implementations and Python packages, executing them directly with task prompts without internal modifications.

OpenHands. OpenHands is an interactive human–agent software development framework that supports multi-turn collaboration for iterative project construction. The system relies on continuous human feedback and intervention throughout the development process, which makes large-scale automated evaluation time-consuming. To align it as closely as possible with the E2EDev setting, we restrict OpenHands to a single execution without any human interaction. We use the official GitHub implementation without modification and report the results in Appendix C.1.

SWE-Agent. SWE-Agent is designed for repository-level software engineering tasks, where the goal is to resolve GitHub issues by modifying an existing codebase, as evaluated in the SWE-bench setting. Its input consists of an existing repository and a natural language issue description, and the output is a patched repository rather than a newly generated project. For our setting, we replace the original prompt template—which instructs the model to read a project from `{{repo_path}}` and modify it according to `{{issue}}`—with a new template that directs the model to generate a runnable project directly from the given user requirement. We implement SWE-Agent using the official GitHub source without further modification, and report its results in Appendix C.1.

Evaluation Coverage. We consider eight LLM-based requirement-to-executable software development frameworks in total. Six frameworks (*Vanilla LLM*, *GPT-Engineer*, *Self-Collaboration*, *MapCoder*, *ChatDev*, and *MetaGPT*) are evaluated across all six LLM backbones. Due to their different design goals and substantially higher evaluation cost, *OpenHands* and *SWE-Agent* are evaluated only on *Claude-Haiku-4.5* and *GPT-4o*, with their results reported in Appendix C.1.

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). Due to the limited number of benchmark tasks at the project level, we run each framework three times on every task. The effectiveness scores reported in Table 3 and Table 8 are computed as the mean and standard deviation across runs.

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:

$$\begin{aligned}\text{Cost} = &(\text{Input Tokens} \times \text{Input Price}) \\ &+ (\text{Output Tokens} \times \text{Output Price})\end{aligned} \quad (4)$$

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 6.

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

Table 6: LLM API Pricing

- *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.

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

where:

  - **FLOPs** $= 2 \times$ (Input Tokens + Output Tokens) $\times$ 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 (6 backbones $\times$ 6 frameworks), resulting in 360 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 7 summarizes the definitions and identification criteria.

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.

C Additional Analysis

C.1 OpenHands and SWE-Agent on E2EDev

OpenHands (Wang et al., 2024) and SWE-Agent (Yang et al., 2024) are two representative and widely used LLM-based agents for software engineering tasks. Both systems are designed to support complex development workflows and have shown strong performance in their original evaluation settings. However, their design assumptions and task formulations differ substantially from the setting targeted by E2EDev, as summarized in Table 9.

E2EDev focuses on **end-to-end requirement-to-executable software development**, where a system is required to generate a complete and runnable software project directly from natural language requirements. This setting emphasizes project-level generation from scratch. In contrast, OpenHands is built around iterative human–agent collaboration, where continuous human feedback guides the development process, while SWE-Agent operates on an existing code repository and targets issue-level modification rather than full project generation. As a result, these systems are not natively aligned with the E2EDev task formulation, and a direct comparison with E2EDev-oriented frameworks is not conceptually equivalent.

Experimental Setup. Due to their substantially higher evaluation cost and different design goals, we evaluate OpenHands and SWE-Agent only on *GPT-4o* and *Claude-Haiku-4.5*. To better align these systems with the E2EDev setting, OpenHands is restricted to a single execution without human interaction. For SWE-Agent, we replace its original prompt template—which instructs the model to load and modify an existing repository—with a new template that directs the model to generate a runnable software project directly from the given user requirement. This modification prevents access to any existing repositories and enforces project generation from scratch.

Results and Discussion. The results in Table 8 show that OpenHands and SWE-Agent underperform the Vanilla LLM baseline on E2EDev. This outcome is expected given the mismatch between their original design objectives and the requirement-to-executable software development task. In particular, both systems exhibit limitations in directly interpreting and implementing fine-grained user requirements without relying on interactive guidance or pre-existing code structures. Overall, these findings further support our main conclusion that E2EDev poses distinct challenges beyond those addressed by existing agent-based software engineering frameworks.

C.2 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.4 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.

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.

Table 7: Error types identified during human evaluation.

Backbone	Method	Effectiveness (↑, %)			Efficiency (↓)		
		Req. Acc.	Test Acc	Bal. Score	Cost ($)	CO_2 (g)	Time (s)
Claude-Haiku 4.5	Vanilla LLM	48.69 ±1.30	63.08 ±0.50	54.45 ±0.98	0.015	0.028	53
	GPT-Engineer	**53.75** ±0.43	**69.41** ±1.07	**60.01** ±0.69	0.016	0.038	59
	Self-Collab.	49.01 ±1.74	61.50 ±1.26	54.01 ±1.55	0.014	0.033	75
	MapCoder	49.61 ±0.50	65.65 ±0.44	56.03 ±0.48	0.099	0.081	129
	ChatDev	44.73 ±1.50	58.09 ±1.03	50.07 ±1.31	0.174	0.188	335
	MetaGPT	5.39 ±0.76	10.66 ±1.77	7.50 ±1.16	0.090	0.091	366
	Openhands	43.45	57.77	49.18	0.021	0.054	-
	SweAgent	44.73 ±0.44	56.87 ±0.45	49.59 ±0.44	0.089	0.089	148
GPT-4o	Vanilla LLM	45.95 ±1.25	60.88 ±1.97	51.92 ±1.54	0.0160	0.083	28
	GPT-Engineer	**50.83** ±1.33	**66.59** ±2.43	**57.13** ±1.77	0.0198	0.132	21
	Self-Collab.	46.83 ±1.12	61.15 ±1.47	52.56 ±1.26	0.0155	0.109	37
	MapCoder	47.70 ±2.57	63.97 ±2.25	54.21 ±2.44	0.1091	0.750	93
	ChatDev	42.71 ±2.44	58.93 ±2.66	49.20 ±2.53	0.1947	1.910	114
	MetaGPT	0.00 ±0.00	0.17 ±0.05	0.07 ±0.02	0.0951	0.794	66
	Openhands	39.25	55.34	45.85	0.062	0.161	-
	SweAgent	40.83 ±2.23	58.05 ±1.91	47.72 ±2.10	0.089	0.089	122

Table 8: Benchmark analysis of effectiveness and efficiency. The grey row denotes the Vanilla baseline. Red and green indicate performance better and worse than Vanilla, respectively. Standard deviations are shown in small grey text. For effectiveness, the best result is shown in **bold** and the second-best is underlined.

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.4, 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.

Method	Task Description	Input	Output	Mode
E2EDev (Ours)	Generates fully runnable software projects from scratch based solely on requirements.	Detailed user requirements	Runnable project	End-to-End Repo Generation
OpenHands	Collaborates via multi-turn dialogue to iteratively build software.	Interactive dialogue	Runnable project	Human-in-the-loop Repo Generation
SWE-Agent	Resolves GitHub issues by modifying existing repositories (SWE-bench).	Existing repo + Issue	Patched repo	Repo Modification

Table 9: Comparison of task formulations. E2EDev targets ab initio generation, distinct from the collaborative or maintenance-focused nature of OpenHands and SWE-Agent.

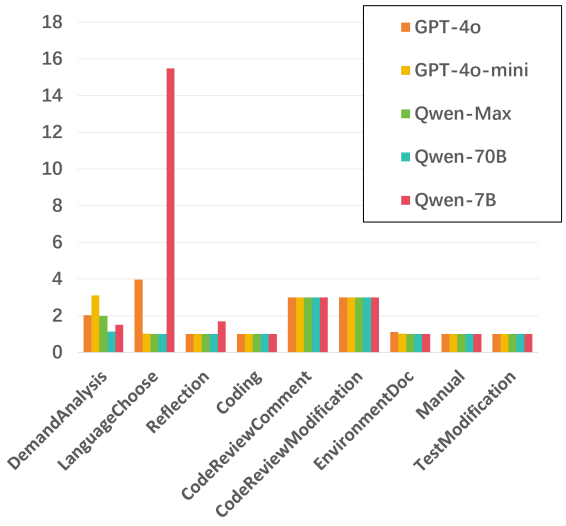

Figure 8: Number of communications in each phase of ChatDev across different backbone models.

C.3 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 8, 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.4 Additional Analysis on HumanEval (Function-level Task)

The main experimental results show that existing methods struggle to handle project-level E2ESD tasks. To better understand this limitation, we further evaluate these methods on a simpler, function-level benchmark, HumanEval (Chen et al., 2021), which isolates code generation from project-level coordination and integration.

As summarized in Table 12, most methods achieve substantially higher performance on HumanEval than on E2ESD. This gap indicates that current approaches remain more effective for isolated function-level code generation than for end-to-end project development.

For MetaGPT, we deliberately evaluate the framework without any HumanEval-specific adaptation. Under this setting, MetaGPT's performance is lower than the results reported in the main body of its original paper, but closely matches the authors' appendix results obtained without benchmark-specific tuning. This confirms that the observed performance is consistent with MetaGPT's default framework design. A closer in-

Base Model	Method	Req.acc	Soft Req.acc	Delta
claude-haiku 4.5	ChatDev	0.4464	0.7025	0.2718
	gpt-engineer	0.5402	0.7991	0.2589
	llmbased	0.4866	0.7454	0.2589
	MapCoder	0.4955	0.7723	0.2768
	MetaGPT	0.0536	0.1027	0.0491
	Self-collaboration-Code-Generation	0.4911	0.7679	0.2768
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: 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	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

Table 11: 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.

spection reveals that most performance degradation arises from the multi-agent collaboration process rather than from code generation itself. During requirement analysis and architectural planning, MetaGPT frequently modifies the target function name or signature specified by the benchmark. As a result, the generated code no longer matches the expected test interface, causing test failures even when the core logic is otherwise correct. This behavior mirrors the failure patterns observed on E2EDev, where the framework's internal analysis stages often alter the original requirements, leading to mismatches between the specified requirements and the generated implementation.

C.5 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": [
    ...
    {
      "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**: ...
```

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

Table 12: 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.

```
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 9.. 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.

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:

Listing 1: Example Gherkin Test Case with Comments

```
# 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
```

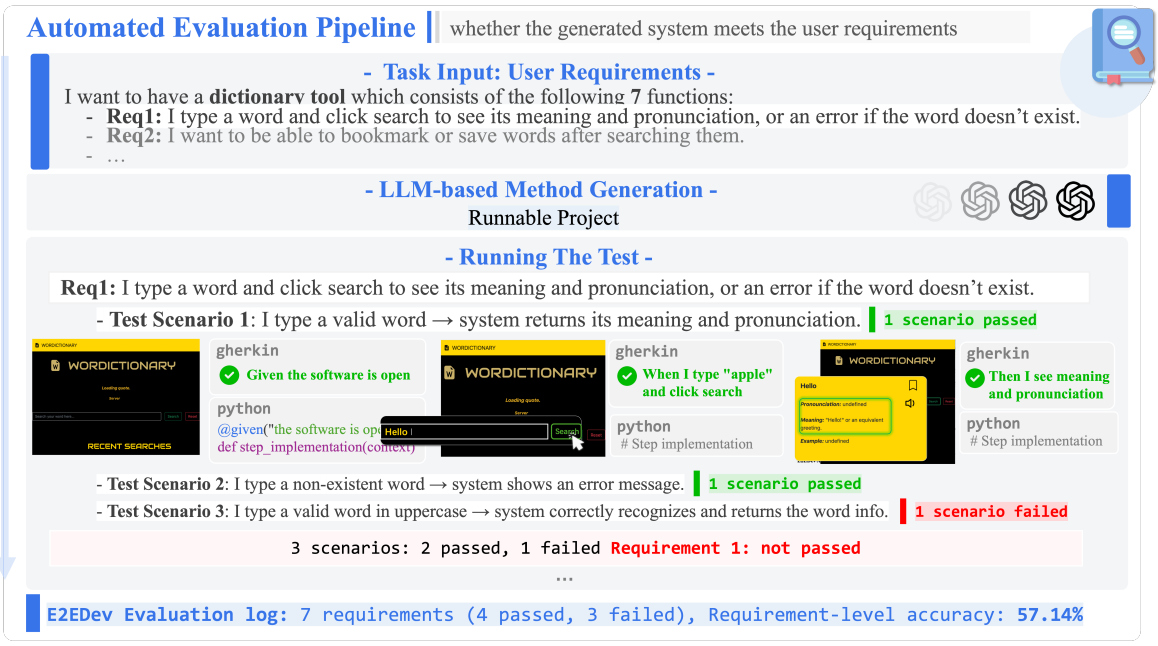

Figure 9: BDD-based automated evaluation pipeline for E2ESD tasks.

```
# 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"
```

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:

Listing 2: Python Step Implementation for Gherkin Scenario

```
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"
```

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. 10.

Once the setup is complete, you can run the tests

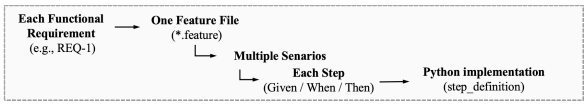

Figure 10: Example directory structure for Behave testing with E2ESD.

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 13, BDD tests focus on validating system behavior from a user perspective, whereas unit tests verify individual code units.

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 1, 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.

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, mapping scenarios to code	Executed within testing frameworks like `unittest` or `pytest`
Audience	Developers and non-technical stakeholders	Primarily developers

Table 13: Comparison between BDD tests and Unit tests.

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.
```