RealDevWorld: Benchmarking Production-Ready Software Engineering

Anonymous authors
Paper under double-blind review

Abstract

Despite rapid advancements in coding agents, current benchmarks in software engineering fail to adequately capture the complexities of real-world software development. To bridge this gap, we introduce **RealDevWorld**, an interactive benchmarking environment comprising two innovative components. First, we offer an extensive suite of realistic open-ended software tasks in diverse domains, incorporating multimodal elements to reflect real-world complexity. Second, we propose **AppEvalPilot**, a new agent-as-a-judge framework that dynamically assesses software functionality through user interface (UI) interactions. **AppEvalPilot** automates the evaluation process through three stages: generating realistic test scenarios, performing interactive adaptive tests, and systematically analyzing results without human effort. Empirical studies demonstrate the effectiveness of our system in achieving accurate, reproducible, and human-aligned evaluations, significantly improving software assessment beyond traditional static methods. Our code is available at Github.

1 Introduction

The rapid advancement of language models has raised expectations for agentic systems to independently construct production-ready software solutions. However, current software development benchmarks remain largely constrained to closed-ended code completion tasks, evaluating methods with skills of code synthesis, bug fixing, and code repair on function-level or class-level (Chen et al., 2021; Austin et al., 2021). While existing repo-level benchmarks attempt to evaluate code retrieval, completion, and maintenance across repositories (Ding et al., 2023; Tang et al., 2023; Liu et al., 2024; Jimenez et al., 2024; Miserendino et al., 2025; Hu et al., 2025), they still operate within predefined structures rather than building systems from the ground up as required in real-world software engineering contexts.

These evaluations, while valuable, fail to capture realistic software engineering challenges. Real-world requirements expressed in natural language are open-ended, making them easy to generate yet difficult to solve and evaluate. First, they are complex to solve, often underspecified and requiring interactive development processes with continuous adaptation to evolving user needs. Second, evaluation demands dynamic, interactive testing beyond static metrics to verify functionality across diverse scenarios. Current benchmarks struggle with both dimensions, lacking interactive evaluation capabilities and thus inadequately measuring production-ready software engineering abilities.

To bridge this gap, we introduce **RealDevWorld**, the first open-ended and interactive evaluation environment designed to comprehensively benchmark and assess AI capabilities in end-to-end software engineering scenarios. **RealDevWorld** consists of two innovative components: (i) **RealDevBench**, a novel open-ended benchmark providing production-ready and diverse software development tasks; and (ii) **AppEvalPilot**, an autonomous agent-based evaluation framework that simulates human-like software testing practices.

Specifically, **RealDevBench** provides 194 carefully curated open-ended software engineering tasks designed to comprehensively assess AI-driven software development from scratch. These tasks are constructed following a key principle: from real-world scenarios to real

Table 1: **Comparison of RealDevWorld with existing benchmarks.** It leverages **AppEvalPilot** (detailed in section 4) for scalable, multi-modal, and interactive software evaluation. *Note: TS = TypeScript; JS = JavaScript; Func. = Function level; Repo. = Repository level; Comp. = Completion; Gen. = Generation; Ret. = Retrieval; Maint. = Maintenance; Dev. = Development.*

Benchmark	Lang.	Level	Tasks	Eval Method	Agent Judge	Input Data	Interactive
BigCodeBench (Zhuo et al., 2024)	PY	Func.	Comp.	Unit test	✗	Text, Code	✗
NaturalCodeBench (Zhang et al., 2024)	PY, Java	Func.	Gen.	Unit test	✗	Multi-modal	✗
LiveCodeBench (Jain et al., 2025)	PY	Func.	Gen.	Unit test	✗	Text, Code	✗
CrossCodeEval (Ding et al., 2023)	Multiple	Repo.	Ret.	Similarity	✗	Text, Code	✗
RepoCoder (Zhang et al., 2023)	PY	Repo.	Comp.	Unit test	✗	Text, Code	✗
RepoBench (Liu et al., 2024)	PY, Java	Repo.	Ret.	Similarity	✗	Text, Code	✗
SWE-Bench (Jimenez et al., 2024)	PY	Repo.	Maint.	Unit test	✗	Text, Code	✗
EvoCodeBench (Li et al., 2025)	PY	Repo.	Ret.	Pass@k	✗	Text, Code	✗
rSDE-Bench (Hu et al., 2025)	PY	Repo.	Dev.	Unit test	✗	Text	✗
SWE-Lancer (Miserendino et al., 2025)	JS, TS	Repo.	Dev.	Unit test	✗	Multi-modal	✗
RealDevWorld	PY, JS, TS	Repo.	Dev.	Unit test	✓	Multi-modal	✓

requirements. Our analysis of programming community needs revealed that demand for coding agents primarily focus on four domains: *display*, *analysis*, *data*, *and game*. By sampling genuine requirements from these domains and systematically expanding them at the function level using language models, we create a diverse and representative task set. This construction methodology makes **RealDevBench** highly scalable to different scenarios, ensuring it meets the demands of production-ready software development in real-world contexts.

Notably, we make deliberate design to enhance both realism and diversity: a subset of tasks introduces multimodal complexity (structured data, images, audio) to replicate real-world constraints, while further diversity is achieved by integrating representative tasks from the *Software Requirements and Design Dataset (SRDD) (OpenBMB)*, encompassing structured software problems across more domains. The unique characteristics of **RealDevBench** compared to existing approaches are summarized in Table 1.

Evaluating software built from scratch requires more than static checks or predefined tests, as these methods fail to capture how open-ended systems truly function. Effective evaluation needs dynamic, realistic interactions—running the software and observing how it responds to actual user scenarios. To this end, we introduce **AppEvalPilot**, an autonomous evaluation agent that mimics human test-driven software engineering practices. Using task descriptions and AI-generated repositories, **AppEvalPilot** creates test plans and interacts with applications through GUI operations and data manipulations to thoroughly verify functionality. It provides interpretative feedback based on the results of dynamic testing execution, enabling capabilities akin to white-box testing in software development. Empirical validations demonstrate that **AppEvalPilot** achieves a correlation of **0.913** with expert human assessments—nearly **35%** higher than LLM-based evaluation—substantially reducing the need for manual review.

In summary, our main contributions collectively establish RealDevWorld, the first interactive evaluation environment for AI-driven software development from-scratch:

- **RealDevBench**: An open-ended and scalable benchmark uniquely designed to evaluate AI frameworks to autonomously develop realistic, fully-functional software repositories from scratch.

- **AppEvalPilot**: An autonomous evaluation method capable of intelligently generating testing scenarios, performing realistic interactive validations, and providing accurate human-aligned assessments, demonstrating significant potential for broader application. Its generalizable capabilities enable reliable automated quality feedback for evaluating diverse software developed from specified requirements, enhancing robustness across the AI development pipeline.

2 Related Work

Benchmarks for Software Engineering. Evaluating repository-level code generation in LLM-based agents presents significant challenges due to the complexity of assessing end-to-end software development processes, which involves system integration, dependency management, and dynamic interactions (Zhuge et al., 2024). Existing function-level benchmarks, such as BigCodeBench (Zhuo et al., 2024), LiveCodeBench (Jain et al., 2025), and NaturalCodeBench (Zhang et al., 2024), primarily focus on code completion or generation at the function or class level, evaluating correctness through static test cases, but unable to provide dynamic, interaction-based test cases necessary for evaluating web interfaces or gameplay functionality. Moreover, These evaluations fail to capture the complexity of software development, particularly for system integration, dependency management, and ambiguous specifications (Hou et al., 2024; Jin et al., 2024). Repo-level benchmarks (Ding et al., 2023; Li et al., 2025; Liu et al., 2024; Zhang et al., 2023; Hu et al., 2025; Jimenez et al., 2024; Miserendino et al., 2025) evaluate complex repository-level software with multiple interrelated components, yet face challenges in comprehensively assessing real-world capabilities. They address repository-wide tasks, yet primarily rely on static metrics like similarity scores or unit tests (Fan et al., 2023; Laskar et al., 2024) that may not fully capture functional completeness. Current advanced benchmarks such as rSDE-Bench (Hu et al., 2025), SWE-Bench (Jimenez et al., 2024) and SWE-Lancer (Miserendino et al., 2025) typically utilize pre-existing test cases, which presents challenges when evaluating software development aspects that require adaptation to changing requirements or creation of entirely new modules. DEVAI (Zhuge et al., 2024) and MLE-Bench (Chan et al., 2024) introduce automated AI development tasks to verify the development skills of agentic systems, but they utilize public AI benchmarks, which are easily leaked into the language model training process. Unlike existing approaches, our proposed benchmark can adapt to newly developed modules and dynamic interaction requirements, simulating human testing processes for the evaluation of software development.

Advanced Judgement Approaches Recent evaluation techniques have established new paradigms, starting with LLM-as-a-Judge (Zheng et al., 2023), which employs language models to evaluate text-based tasks instead of traditional metrics. While effective for textual outputs, this approach is limited to assessing static final result rather than development processes or intermediate outputs. Agent-as-a-Judge (Zhuge et al., 2024) builds on this by introducing a dynamic agent-based approach, leveraging multi-dimensional scoring and iterative feedback loops. However, it remains insufficient for evaluating software with complex interactive components, particularly those with GUIs. These require evaluating both interaction flows and the functionality of UI elements, which are more dynamic and nuanced. To address these challenges, we propose an innovative approach that integrates GUI agent capabilities for interactive testing, inspired by recent advances in GUI agents (Xu et al., 2024; Cheng et al., 2024), to mirror human testing processes for a more dynamic and comprehensive evaluation. We summarized the comparisons in Table 1.

3 RealDevBench: Open-Ended Software Development Benchmark

Software development encompasses complex processes beyond regular and functional code snippet generations, requiring interpretation of requirements, architectural design, dependency management, and coherent repository organization. To comprehensively evaluate AI systems on these dimensions, we introduce **RealDevBench**, a benchmark specifically designed to assess end-to-end software engineering capabilities in a more practical manner.

3.1 Overview

RealDevBench consists of **194** requirements across four practical domains, *Analysis, Display, Data, and Game*, which mirrors essential engineering demands. The dataset is characterized by three core attributes: (1) **open-ended repository construction**, requiring systems to build software from scratch rather than complete predefined structures; (2) **multimodal complexity**, incorporating diverse inputs such as text, images, audio and tabular data to test

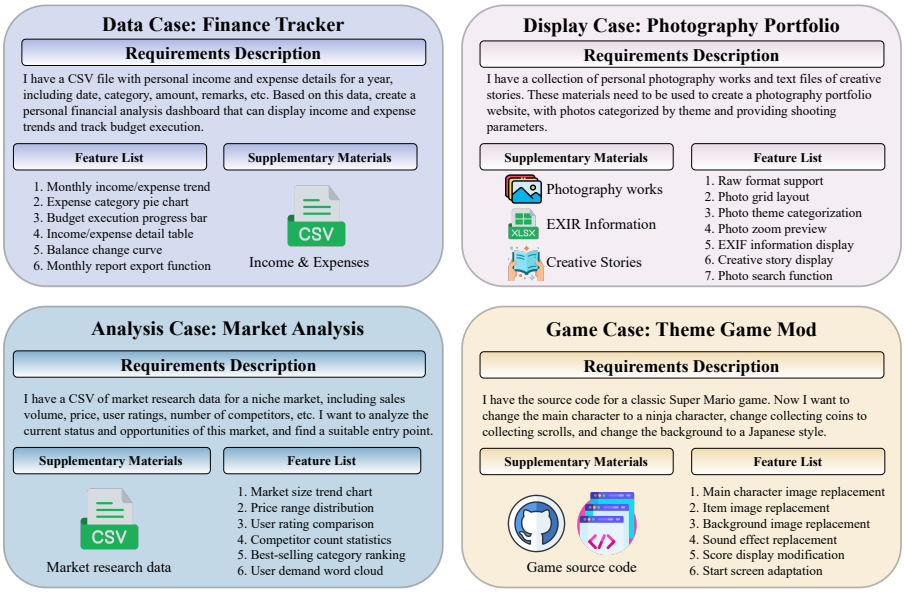


Figure 1: Representative cases from **RealDevBench** across four domains - Data, Display, Analysis, and Game. Each case follows a consistent triplet structure of requirements description, feature list, and supplementary materials, while showcasing diverse application scenarios from financial dashboards to game modifications, reflecting the multimodal challenges of real-world software engineering tasks.

integrative capabilities; and (3) **functional diversity**, addressing a wide range of software functionalities across varying complexity levels.

3.2 Benchmark Construction

The **RealDevBench** dataset is constructed through a systematic pipeline designed to ensure its relevance, complexity, and evaluative rigor.

Domain Distribution We conducted random sampling from the SRDD (OpenBMB), which contains a rich collection of prompts organized into 5 major categories and further subdivided into 40 subcategories, specifically curated to facilitate research in NL2Software. This sampling ensures a diverse representation of tasks that reflect real-world applications. Moreover, we crawled and analyzed a large number of open-source projects, extracting common tasks such as creating company homepage websites, video editing tools, investment statistics, market data analysis, and dashboards. This selection was driven by the need to evaluate the effectiveness of the prompts in a variety of practical scenarios. As illustrated in Figure 1, tasks in our benchmark are then distributed as follows: Display (50.0%), Data (14.4%), Analysis (18.6%), and Game (17.0%). This distribution highlights the focus areas within the dataset. We provide more examples in appendix B.

Task Structure and Formulation In **RealDevBench**, every task is organized as a triplet, consisting of (1) **Software Requirements Description (SRD)**: A brief textual summary outlining the project's purpose and setting; (2) **Feature List**: A comprehensive list detailing functional goals that define success criteria; (3) **Supplementary Materials:** Resources tailored to each task, like images, audio, or datasets, that add real-world complexity. The anticipated result is a completely operational software project contained in a code repository.

4 AppEvalPilot: Automated Interactive Software Evaluation

The rise of AI-driven software development demands scalable and adaptive evaluation methods that go beyond traditional static benchmarks. While **RealDevBench** (section 3.1)

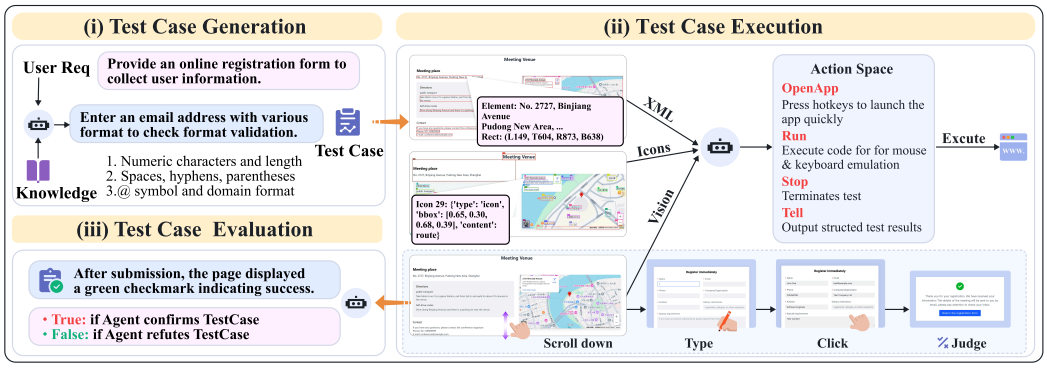


Figure 2: **Overall design of AppEvalPilot** illustrates our full automated testing workflow. **Left**: Test Case Generation transforms user requirements into structured test scenarios by leveraging domain knowledge (email validation rules shown). **Center**: Test Case Execution demonstrates multimodal perception capabilities, processing both XML accessibility trees and visual elements to navigate interfaces through scrolling, typing, and clicking actions. **Right**: Test Case Evaluation applies binary verification criteria to interaction outcomes, enabling objective assessment of software functionality without human intervention.

functions as a broad platform for evaluating end-to-end software engineering skills, traditional evaluation methods, which typically depend on manual reviews or fixed test suites, encounter three main drawbacks: (1) they are not scalable, (2) they have difficulty adjusting to changes in software behavior, and (3) they fail to account for interactive, real-time responses from AI-driven applications. To address these challenges, we introduce **AppEvalPilot**, an automated evaluation system designed for direct, interaction-based testing of software projects. Unlike static analysis or rigid test suites, **AppEvalPilot** actively engages with software interfaces, executing real-time user interactions to assess functional correctness and adaptability. Figure 2 illustrates its three-stage pipeline:

- **Test Case Generation**: Automatically creates comprehensive test cases from software requirements and domain-specific knowledge.
- **Test Case Execution**: Simulates dynamic user interactions (e.g., clicks, scrolling) using both textual and visual inputs to test software behavior in real-world scenarios.
- **Test Result Evaluation**: Analyzes outcomes against expected behaviors to determine functional correctness and completeness.

This shift from static analysis to dynamic, automated testing aligns with **RealDevBench** 's emphasis on practical software development, enabling scalable, repeatable, and rigorous evaluation of AI systems.

4.1 Core Modules

AppEvalPilot operates through three closely connected components, outlined below, that together facilitate the comprehensive automation of software evaluation.

Test Case Generation AppEvalPilot starts by automating the creation of high-quality, contextually relevant test cases that align with **RealDevBench** 's open-ended and multimodal requirements. To achieve this, it leverages few-shot learning (Wang et al., 2020) to infer requirement-to-test mappings from a small set of manually curated examples, allowing it to generalize efficiently across diverse software requirements (details and prompts is available in Appendix C). Additionally, it integrates domain-specific knowledge, such as game mechanics for *Game* tasks, and security protocols for *Data* tasks—to ensure test cases accurately reflect real-world scenarios and practical constraints. To evaluate the effectiveness of our **AppEvalPilot**, we select 27 tasks for a manual evaluation, concentrating on two key aspects: the coverage of the feature list and the precision of the test points generated.

Test Case Execution AppEvalPilot next autonomously executes the generated test cases by directly interacting with software applications through their graphical user interfaces (GUIs),

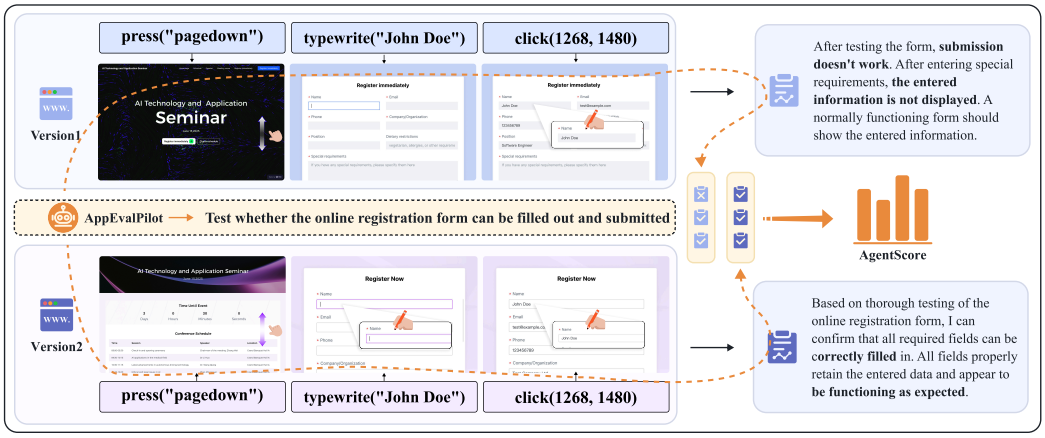


Figure 3: Evaluation pipeline of **AppEvalPilot**. demonstrating interactive software testing capabilities. The agent performs identical test sequences (press, typewrite, click) on two different web registration form implementations, systematically assesses functionality through direct interaction, documents observable differences in form submission behavior, and generates quantitative scores based on functional requirements. This visual comparison highlights **AppEvalPilot**'s ability to detect subtle implementation flaws that might escape static code analysis.

effectively simulating genuine user interactions. As shown in Figure 2, the execution agent handles multiple input types from active software, including textual data (XML) from accessibility trees (a11ytree) and visual data like icons and screenshots, to accurately interpret the interface. This facilitates a thorough understanding of the software's UI for precise interaction. Specifically, the agent operates within a structured action space consisting of four core commands, serving as the foundational components for complex interactions. These atomic actions, detailed in the appendix D and Figure 2, allow **AppEvalPilot** to execute complex tasks such as form filling, web navigation, and validation checks. During the execution of each test case, **AppEvalPilot** systematically transforms it into a structured, multi-step execution workflow, wherein each step may encompass multiple actions amalgamated to facilitate higher-level operations. To ensure efficiency and flexibility, **AppEvalPilot** employs adaptive decision-making through historical reasoning and model-based planning, following the Plan-Act framework (Wang et al., 2023) to continuously improve execution processes. This method allows **AppEvalPilot** to enhance execution by refining subtasks, minimizing redundant actions, and adapting strategies in response to unexpected UI conditions or errors, especially important for lengthy software testing tasks.

Test Result Evaluation The Test Result Evaluation module assesses execution outcomes by comparing the actual interaction results against the expected success criteria defined in **RealDevBench**. Specifically, after each test execution, **AppEvalPilot** generates a structured evaluation report that details the performed actions (e.g., entering a search query) and the observed outcomes (e.g., displayed products). A JudgeLLM analyzes this report alongside the original test case description, categorizing the results into three types: *Pass*, if the observed behavior aligns with the expectations described in the test case; *Fail*, if it explicitly contradicts the expectations; or *Indeterminate*, if the result cannot be conclusively determined from the available information. This categorization provides clear insights into software efficiency and helps detect possible problems in automated tests, as shown in Figure 3.

5 Experiments

5.1 Baseline Settings

To comprehensively evaluate performance on our **RealDevBench**, we selected the following diverse set of systems: (1) **Basic LLMs:** we utilized DeepSeek-v3 (DeepSeek-AI, 2024) and Claude-3.5-Sonnet-v2 (Anthropic, 2024), both recognized for their exceptional performance on coding tasks. (2) **Open-Source Agents:** we also included GPT-Pilot (gpt, 2023) as a representative solution from the open-source community. (3) **Commercial Products:** and

Table 2: Experimental outcomes for the **RealDevBench**, including evaluations conducted by humans and **AppEvalPilot**, along with essential statistical information. '-' indicates that the cost cannot be directly obtained.

Model/Framework	Effectiveness		Efficiency		Statistics		
	Executability↑	SQ↑	Time (s)↓	Cost ($)↓	# Files	# Code Files	LOC
DeepSeek-v3	0.35	0.16	28.53	0.01	7.3	3.5	333.2
Claude-3.5-Sonnet-v2	0.47	0.22	41.50	0.07	7.2	2.6	193.5
GPT-Pilot	0.16	0.09	227.87	0.71	8.9	8.3	489.5
Bolt	0.64	0.39	118.6	0.14	27.0	13.0	583.0
Lovable	0.49	0.22	264.7	-	96.7	73.6	2123.7
MGX	0.67	0.45	513.3	0.86	269.0	16.0	1030.3

we evaluated three production-grade commercial solutions—Bolt (StackBlitz, 2024), Lovable (Team, 2024a), and MGX (Team, 2024b)—designed to deliver end-to-end productivity in software development through LLM-based agent frameworks.

5.2 Evaluation Metrics

To comprehensively evalute the quality of generated software and record project complexity, we employed three categories of metrics: ***effectiveness***, ***efficiency***, and ***statistics***.

RealDevBench Metrics For effectiveness, we define **Software Quality(SQ)** and **Executability** as the core metric, which evaluates correctness, completeness, and usability of generated software. Specifically, **SQ** is measured by evaluating discrete feature test outcomes, recorded as $f_i = 0$ (Failed) or $f_i = 1$ (Pass). The final SQ score is

$$SQ = \frac{1}{n} \sum_{i=1}^{n} f_i$$

where n represents the total number of features and f_i denotes the test outcome of the i-th feature. **Executability** measures whether the generated project deploys and executes without errors. Projects are tested using a standard launch script, with successful execution scored as 1 and failures as 0. Moreover, For **efficiency**, we report **cost** and **time**, tracking token usage during generation and total elapsed time from input requirement to final output. Additionally, to quantitatively evaluate the software complexity, we provide statistical metrics, including the total number of files, code files, and lines of code in the generated project, details can be found in Appendix H.

Benchmark AppEvalPilot To validate the **AppEvalPilot**, we provide a comparison (including the time and cost) of humans and agents.In addition, we present an analysis of deviation distribution to illustrate the potential reduction in manual labor. We assess **AppEvalPilot**'s SQ performance by calculating the Pearson correlation coefficient to quantify inter-variable relationships.

A comparative analysis was executed to measure consistency across different evaluation methods: direct LLM evaluation, **AppEvalPilot**, and human expert assessment, with scores defined as **LLM score**, **Agent score**, and **Human score** respectively. Specifically, LLM score is calculated by scoring the generated software code utilizing the LLM (here is Claude-3.5-Sonnet-v2), and then multiplying this score by the executability Appendix I. To cover diverse complexity, domain, and quality, 60 applications were strategically sampled from our benchmark.

5.3 Baselines Performance on RealDevBench

The results in Table 2 show significant performance variations across effectiveness, efficiency, and statistical metrics. MGX has the best effectiveness with the highest executability (0.67)

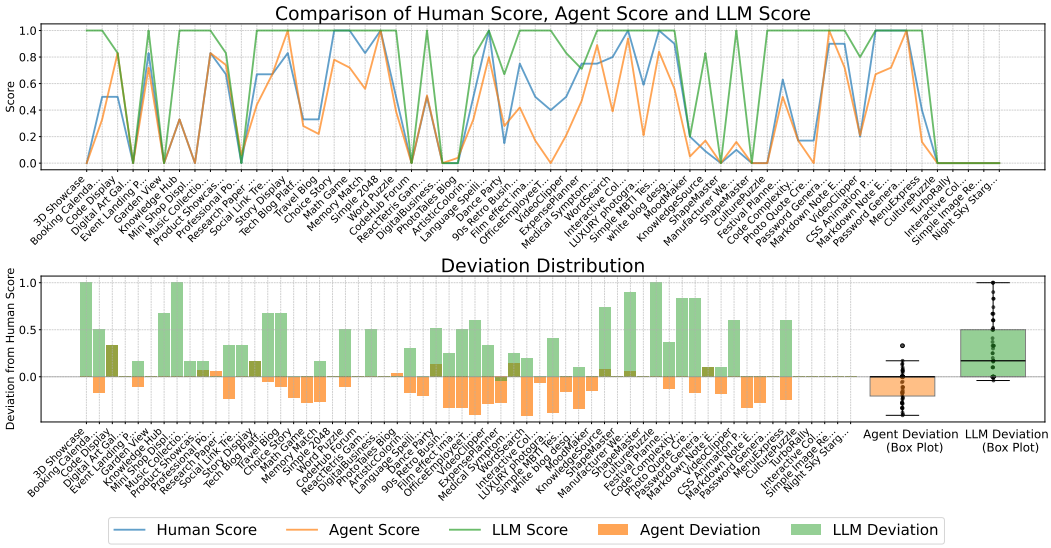


Figure 4: **Top:** Distribution of software quality scores across 60 sample applications from human, agent, and LLM. **Bottom:** Deviation from human judgment shown as per-application differences (left) and statistical summary (right) for agent and LLM evaluations.

and solution quality (0.45) scores, though at considerable efficiency costs (513.3 seconds, \$0.86). In contrast, DeepSeek-v3 and claude-3.5-sonnet-v2 has the efficiency (28.53s/\$0.01 and 41.50s/\$0.07) despite their simple input-output-alike workflow. Bolt, a bussiness product, which is well-balanced option with strong effectiveness (0.64 executability, 0.39 SQ) and reasonable efficiency (118.6s, \$0.14).

Statistical measures reveal Lovable produces the most extensive solutions (2123.7 LOC across 73.6 code files), while Claude-3.5-Sonnet-v2 generates the most concise outputs (193.5 LOC in 2.6 files). These findings highlight a clear trade-off between solution quality and computational efficiency, suggesting users should select frameworks based on their specific priorities—whether emphasizing code quality, execution speed, or cost considerations.

The data presented in Table 2 reveals a comparably low scores across basic LLM baselines, indicating significant challenges in the current SOTA language models for software generation tasks. Even the highly capable Claude-3.5-Sonnet-v2 model exhibits limited executability (0.47) and solution quality (0.22). We contend that these models face significant constraints in their capacity to produce executable and high-quality software artifacts.

Conversely, commercial software shows significant improvements in executability. This progress is due to the use of advanced project templates, leading to more file and code generation delivered. Additionally, commercial products implement automated and optimized development processes, like dependency installations and deployment, which further boost the models to produce executable software.

However, even with this improvement in executability, the commercial products' SQ remain relatively low (Bolt 0.39 and MGX 0.45). This observation suggests that while commercial products may have made advancements in generating executable software, the current AI-assisted software development capability still face challenges in producing product-ready software.

5.4 AppEvalPilot Performance

Figure 4 presents a comprehensive comparison between LLM scores, agent-based interactive evaluation (**AppEvalPilot**), and human expert judgment across 60 samples from our benchmark, which were randomly sampled according to the distribution of human scores.

Correlation with Human Judgment As demonstrated in the upper panel of Figure 4, our statistical analysis indicates that **AppEvalPilot** scores exhibit a substantially higher correlation with human judgments ($r = 0.913$) than do LLM-generated scores ($r = 0.681$). Notably, LLM scores display a bias toward binary scoring, with 84.8% of the evaluations receiving exact scores of 0 or 1, which inadequately represents the nuanced nature of software implementation. Conversely, **AppEvalPilot** generates a more nuanced and balanced distribution, with extreme scores (exactly 0 or 1) comprising only 33.9% of evaluations, closely resembling the human evaluators' distribution (42.4% extreme scores). This demonstrates **AppEvalPilot**'s enhanced capability to accurately reflect partial functionality and provide realistic assessments of implementation completeness.

Deviation Analysis The bottom panels of Figure 4 illustrate critical differences in deviation patterns from human judgment. Agent scores show a modest mean deviation of -0.08, reflecting slightly more stringent evaluations as it conducts more thorough testing across edge cases than human evaluators typically perform. In contrast, the LLM score exhibits a substantial positive mean deviation of 0.28, primarily because it evaluates code quality based solely on surface-level code appearance without verifying correctness through actual execution. Instead of relying solely on code inspection, our **AppEvalPilot** utilizes interactive testing to functional deficiencies, resulting in a distribution that closely approximates human evaluation patterns. Consequently, LLM consistently overestimates functionality, as it fails to account for potential errors or incorrect behaviors during runtime, but **AppEvalPilot** exhibits more alignment with human expert evaluators. Notably, **AppEvalPilot**'s deviation variance (0.02) is significantly lower than LLM's variance (0.1), demonstrating that agent-based evaluation provides more consistent and reliable assessments that closely align with human judgment across diverse applications. To further substantiate these findings, we present a detailed analysis of **AppEvalPilot**'s testing accuracy in appendix F and a qualitative study of its failure cases in appendix G, respectively.

Evaluation Efficiency Table 3 quantifies the substantial resource optimization achieved by **AppEvalPilot** compared to human evaluation. **AppEvalPilot** reduces evaluation time by 55% while dramatically lowering costs by 94.8%. These significant efficiency gains, combined with the high correlation to human judgment, establish **AppEvalPilot** as a practical and economical solution for large-scale software quality assessment.

Table 3: **Cost and Time Comparison of Evaluation Methods** Comparison of average time(min) and cost($) per application between human and **AppEvalPilot**. Costs for **AppEvalPilot** reflect the computational resources used. Human evaluation costs are calculated based on five annotators at an hourly rate of $15.

	Avg. Time(min) /App↓	Avg. Cost ($)/App ↓
Human	20.0	5.0
AppEvalPilot	9.0 (55%↓)	0.26 (94.8%↓)

6 Conclusion

We introduce **RealDevWorld**, a comprehensive environment for evaluating AI powers in software engineering. Our work addresses gaps in existing benchmarks through realistic tasks and efficient evaluation. **RealDevBench** offers 194 diverse tasks across multiple domains with real-world multimodal elements. We evaluate the entire development process from requirements to implementation. The agent-as-a-judge method **AppEvalPilot** automates the evaluation of open-ended software systems by generating test plans, executing interactions, and providing accurate assessments while lowering costs. **RealDevWorld** has the potential to revitalize AI-driven software engineering benchmarks.

Ethics Statement

This research adheres to scientific ethical standards with rigorous internal and external blind evaluation protocols ensuring unbiased assessment. All evaluators participated in blind review processes and received fair compensation for their contributions.

For our human evaluation process, we recruited 12 professional annotators through a specialized technical recruitment platform, all with at least 5 years of software testing experience and backgrounds in computer science. Annotators were compensated at an hourly rate of $35, above industry standards, and were not informed about which systems generated the software they were evaluating. We implemented a double-blind protocol where each application was assessed by four independent evaluators using standardized rubrics to ensure consistency. Inter-annotator agreement was measured using Cohen's Kappa (0.78), indicating substantial agreement among evaluators.

Our datasets combine publicly-sourced examples with proprietary data created specifically for this research, all developed following strict ethical guidelines free from sensitive information. RealDevWorld is designed to augment rather than replace human software engineers, supporting productivity while allowing professionals to focus on creative and strategic work. We acknowledge the computational resources required and have optimized experiments to minimize environmental impact. All data and results will be made publicly available to promote transparency and advance the field.

References

Gpt-pilot: Ai developer tool. https://github.com/Pythagora-io/gpt-pilot, 2023.

Anthropic. Claude 3.5 sonnet. https://www.anthropic.com/news/claude-3-5-sonnet, 2024. Accessed on March 28, 2025.

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

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

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

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. Seeclick: Harnessing gui grounding for advanced visual gui agents. *arXiv preprint arXiv:2401.10935*, 2024.

DeepSeek-AI. Deepseek-v3 technical report, 2024. URL https://arxiv.org/abs/2412.19437.

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

Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pp. 31–53. IEEE, 2023.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A

systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33 (8):1–79, 2024.

Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. Self-evolving multi-agent networks for software development. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=4R71pdPBZp.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*, 2024.

Md Tahmid Rahman Laskar, Sawsan Alqahtani, M Saiful Bari, Mizanur Rahman, Mohammad Abdullah Matin Khan, Haidar Khan, Israt Jahan, Amran Bhuiyan, Chee Wei Tan, Md Rizwan Parvez, et al. A systematic survey and critical review on evaluating large language models: Challenges, limitations, and recommendations. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 13785–13816, 2024.

Jia Li, Ge Li, Xuanming Zhang, Yunfei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. Evocodebench: An evolving code generation benchmark with domain-specific evaluations. *Advances in Neural Information Processing Systems*, 37:57619–57641, 2025.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=pPjZIOuQuF.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. Swe-lancer: Can frontier llms earn $1 million from real-world freelance software engineering? *arXiv preprint arXiv:2502.12115*, 2025.

OpenBMB. Srdd. https://github.com/OpenBMB/ChatDev/tree/main/SRDD. Accessed: 2025-03-29.

StackBlitz. Bolt: Ai-powered development platform. https://bolt.new, 2024.

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

Lovable Team. Lovable: Ai development solution. https://lovable.dev, 2024a.

MetaGPT Team. Mgx: Ai software development platform. https://mgx.dev, 2024b.

Aryan Vichare, Anastasios N. Angelopoulos, Wei-Lin Chiang, Kelly Tang, and Luca Manolache. Webdev arena: A live llm leaderboard for web app development, 2025.

Lei Wang et al. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*, 2023.

Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM Computing Surveys*, 53(3):1–34, 2020.

Jiaming Xu, Kaibin Guo, Wuxuan Gong, and Runyu Shi. Osagent: Copiloting operating system with llm-based agent. In *2024 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–9. IEEE, 2024.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, 2023.

Shudan Zhang, Hanlin Zhao, Xiao Liu, Qinkai Zheng, Zehan Qi, Xiaotao Gu, Yuxiao Dong, and Jie Tang. Naturalcodebench: Examining coding performance mismatch on humaneval and natural user queries. In *Findings of the Association for Computational Linguistics ACL 2024*, pp. 7907–7928, 2024.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623, 2023.

Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, et al. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934*, 2024.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

A Benchmark Construction Process

A.1 Multi-Source Requirement Analysis

Real-World Development Scenario Investigation: To ensure comprehensive coverage of real-world development scenarios, we conducted systematic analysis across multiple sources. We manully browsed website development project listings on major freelancer platforms (Upwork[1] and Freelancer[2]) to understand extensive requirement descriptions and specifications from actual client demands.

Benchmark Cross-Validation: We also examined WebDev Arena (Vichare et al., 2025), a real-world benchmark focused on LLM-based website development. The requirement scenario categories identified in WebDev Arena closely align with SRDD (OpenBMB)'s main scenario categories, providing validation for our approach. Since we could not access the complete WebDev Arena requirement description data, we chose SRDD as our foundation.

Category Consolidation: We consolidated SRDD's subcategories into 4 general scenarios based on implementation functionality and purpose, using insights from multiple sources.

A.2 Open-Source Project Integration

Project Selection Criteria: We crawled projects from GitHub with: Comprehensive documentation (README, API docs) Production-ready quality (1000+ stars, active development) Clear feature specifications in documentation **Selected Open-Source Projects**: We crawled the following open-source projects, all containing complete development requirements and feature descriptions that meet real deployment and application scenario needs.

A.3 From Requirements to Features

We use Claude-3.5-Sonnet (Anthropic, 2024) to expand SRDD (OpenBMB) requirements into detailed features. The model generates structured JSON tasks with IDs, prerequisites, and success criteria across functional categories (UI/UX, Data Management, Business Logic, Reporting). We will include prompt templates and feature examples in the revised manuscript.

Supplementary Materials Integration: We added practical complexity through carefully curated supplementary materials: Images: Sourced from Unsplash [3] for thematic relevance and professional quality Datasets: Selected from Kaggle [4] based on topic relevance and appropriate complexity Documents: Manually created documents (resumes, business proposals, catalogs) that mirror real-world usage scenarios. **Construction Process:** Document creation involves approximately 1-2 hours per category using LLM-generated content with human review. Human verification focuses on checking consistency of information such as dates, locations, and other factual details.

Examples include:

- *Resumes*: Multi-page CVs with employment history, education, and skills
- *Business proposals*: Project proposals with budgets, timelines, and deliverables
- *Product catalogs*: Technical specifications, pricing, and product descriptions
- *Financial reports*: Quarterly statements with charts and analysis
- *Legal contracts*: Service agreements with terms and conditions

[1] https://www.upwork.com
[2] https://www.freelancer.com
[3] https://unsplash.com/
[4] https://www.kaggle.com/

Table 4: Curated Supplementary Materials for Practical Complexity

Material Type	Source	Selection Criteria & Description
Images	Unsplash[5]	Sourced for thematic relevance and professional quality to enhance visual appeal and contextual appropriateness of projects
Datasets	Kaggle [6]	Selected based on topic relevance and appropriate complexity level to provide realistic data processing challenges
Documents	Manually Created	Custom-developed documents that mirror real-world usage scenarios and professional contexts.

B Benchmark Samples Analysis

This section presents representative examples from the **RealDevBench** dataset across four domains: Display, Analysis, Data, and Game. These examples illustrate the diverse software engineering challenges evaluated by our benchmark.

B.1 Display Domain Examples

The Display domain focuses on UI-centric applications requiring front-end development expertise:

Display Task 1: Professional Portfolio

Software Requirements Description:
Please help me create a professional personal portfolio website. I will provide a PDF resume that includes my work experience, project experience, and skills list, as well as a professional profile picture. The website needs to highlight my project experience and display my profile picture in appropriate places, while ensuring that sensitive salary information in the resume is hidden. The website will include the following features:
Feature List:

1. A fixed navigation bar with links to pages such as Home, Projects, Skills, and Contact.
2. A profile picture and personal introduction section to showcase my background and professional field.
3. A project experience card list to showcase the important projects I have worked on.
4. A skill tag cloud to visually present the skills and expertise I have mastered.
5. A social media link list, making it easy for visitors to quickly access my social media platforms.
6. A PDF resume download button, allowing visitors to download my resume.
7. A responsive design, ensuring the website displays well on both desktop (greater than 1024px) and mobile devices (less than 768px).

Supplementary Materials: Resume (PDF), profile photograph

Display Task 2: Social Link Tree

Software Requirements Description:
I have a set of social media links and creative platform homepage links. These materials need to be used to create a link navigation page that conveniently displays all my links on a single page. Please design and implement a social link navigation page based on the following requirements:
Feature List:

1. Display a personal avatar and profile text.
2. Display all links as a list of buttons.
3. Links can be filtered by category tags.
4. Add a theme toggle button to support both light and dark modes.
5. Generate a QR code for the page to make it easy for others to scan and access.

Supplementary Materials: Link.md containing social media platform URLs

B.2 Analysis Domain Examples

The Analysis domain challenges involve transforming raw data into actionable insights:

Analysis Task 1: Blog Traffic Analysis

Software Requirements Description:
I have a blog visit data CSV with PV, UV, visit duration, source page, etc. and want to analyze the visit pattern and give optimization suggestions. Please design and implement the data analysis based on the following requirements:
Feature List:

1. Draw a daily access trend graph to show the trend of blog access.
2. Provide a ranking of popular articles to show the most visited articles.
3. Plot the average dwell time graph to analyze how long readers stay on the page.
4. Provide visit source percentage to help me understand the source channels of visitors.
5. Provide page bounce rate table to analyze which pages have higher bounce rate.
6. Provide popular search terms cloud to show the keywords searched by users.

Supplementary Materials: Blog visit data.csv

Analysis Task 2: Product Review Analysis

Software Requirements Description:
I have a CSV of user review data for a product on an e-commerce platform containing ratings, review text, date of purchase, etc., and would like to analyze these reviews and summarize the product benefits and issues. Please design and implement the data analysis based on the following requirements:
Feature List:

1. Draw a rating distribution chart to show the distribution of ratings for the product.
2. Provide a keyword extraction table to analyze the keywords appearing in user reviews.
3. Plot monthly rating trends and analyze changes in ratings over time.
4. Provide advantages and problems classification, summarize the advantages and disadvantages of the product.
5. Provide the rate of favorable and unfavorable charts, showing the proportion of favorable and unfavorable reviews.
6. Provide an excerpt of popular reviews, showing what users are saying in key reviews.

Supplementary Materials: User comment data.csv

B.3 Data Domain Examples

The Data domain focuses on information processing and visualization systems:

Data Task 1: Finance Tracker

Software Requirements Description:
I have a CSV of a year's worth of personal income and expense details, including dates, categories, amounts, notes, and other information. Based on this data, create a personal finance analytics Kanban board that can show income and expenditure trends and track budget execution.
Please design and implement the dashboard based on the following requirements:
Feature List:

1. Display a monthly income and expenditure trend chart.
2. Provide a pie chart of expenditure categories.
3. Display a budget execution progress bar.
4. Provide an income and expenditure breakdown grid.
5. Show a curve of balance changes.
6. Provides a monthly report out function.

Supplementary Materials: Personal income and expenditure details.csv

Data Task 2: Stock Data View

Software Requirements Description:
I have a CSV file with historical stock data, including date, opening price, closing price, trading volume, and related news headlines. Based on this data, I would like to create a dashboard to display the market trends of the stock and help me analyze its movement.
Please design and implement the dashboard based on the following requirements:
Feature List:

1. Candlestick Chart (K-Line Chart): Display a candlestick chart to visualize the stock's opening, closing, high, and low prices over time.
2. Trading Volume Bar Chart: Show a bar chart that represents the trading volume on different days.
3. Technical Indicators Chart: Provide a chart with technical indicators like Moving Averages (MA), Relative Strength Index (RSI), or Bollinger Bands.
4. News Sentiment Analysis Chart: Display a sentiment analysis chart showing the positive, negative, and neutral sentiment of the related news headlines.
5. Correlation Heatmap: Provide a heatmap that shows the correlation between the stock price and other related data (such as volume, technical indicators, etc.).
6. Data Export Feature: Provide a function that allows users to export the analyzed data in a format such as CSV or Excel.

Supplementary Materials: Stock historical data.csv

B.4 Game Domain Examples

The Game domain challenges test interactive entertainment application development:

Game Task 1: Mini Card Game

Software Requirements Description:
Please develop a card battle game based on the following requirements, where players can play turn-based battles against the computer:
Feature List:

1. Create a card display interface.
2. Implement a basic matchmaking system.
3. Add a simple AI opponent.
4. Implement a turn counter.
5. Judge the winners and losers and display the results.
6. Add a replay button.

Supplementary Materials: None

Game Task 2: Two Player Game

Software Requirements Description:
I have the python source code for a single player tetris game and would like to transform it into a two player game, allowing two players to play on the same computer. Please make changes based on the following requirements:
Feature List:

1. Add two player control settings.
2. Implement a split screen display function.
3. Create a scoreboard to show the score of each player.
4. Display the winner and loser results at the end of the game.
5. Add a restart button to allow restarting the game.
6. Provide hints to help players understand the rules of the game.

Supplementary Materials: Tetris_game.py, MONACO.TTF, game_over.gif

C AppEvalPilot Details

Test Case Generation

Test Case Examples:

1. Verify persistent top navigation bar positioning during scrolling
2. Validate intra-page navigation link accuracy (”Home”, ”Projects”, etc.)
3. Confirm avatar image rendering quality and aspect ratio preservation
4. Audit biographical text completeness and typographic consistency
5. Check project card list formatting and content integrity
6. Verify absence of compensation data in project disclosures
7. Validate skill tag cloud layout responsiveness
8. Test interactive hover effects on skill tags
9. Confirm social media link destination accuracy
10. Validate PDF resume download functionality
11. Verify PDF file integrity and readability

Case Generation Prompt:

```
You are a professional test engineer. Please generate a series of specific test
    cases based on the following user requirements for the webpage.
Requirements:
1. Test cases must be generated entirely around user requirements, absolutely not
    missing any user requirements
2. Please return all test cases in Python list format
3. When generating test cases, consider both whether the corresponding module is
    displayed on the webpage and whether the corresponding function is working
    properly. You need to generate methods to verify webpage functionality based on
    your knowledge.
4. Please do not implement test cases that require other device assistance for
    verification.
5. Please control the number of test cases to 15~20, focusing only on the main
    functionalities mentioned in the user requirements. Do not generate test cases
    that are not directly related to the user requirements.
6. When generating test cases, focus on functional testing, not UI testing.
User Requirements: {demand}
Please return the test case list in List(str) format, without any additional
    characters, as the result will be converted using the eval function.
```

Test Execution Agent

Execution Agent Prompt:

```
You are a professional and responsible web testing engineer (with real operation
    capabilities). I will provide you with a test task list, and you need to provide
    test results for all test tasks. If you fail to complete the test tasks, it may
    cause significant losses to the client. Please maintain the test tasks and their
    results in a task list. For test cases of a project, you must conduct thorough
    testing with at least five steps or more - the more tests, the more reliable the
    results.

[IMPORTANT]: You must test ALL test cases before providing your final report! Do not
    skip any test cases or fabricate results without actual testing! Failing to
    complete the entire task list will result in invalid test results and
    significant client losses.

Task Tips:
Standard Operating Procedure (SOP):
1. Determine test plan based on tasks and screenshots
2. Execute test plan for each test case systematically - verify each case in the
    task list one by one
3. After completing each test case, you can use Tell action to report that
    individual test case result
4. After completing ALL test case evaluations, use Tell action to report the
    COMPLETE results in the specified format

Reporting Language: Answer in natural English using structured format (like
    dictionaries). Tell me your judgment basis and results. You need to report the
    completion status of each condition in the task and your basis for determining
    whether it's complete.

Note that you're seeing only part of the app(or webpage) on screen. If you can't
    find modules mentioned in the task (especially when the right scroll bar shows
    you're at the top), try using pagedown to view the complete app(or webpage).

Inspection Standards:
1. Test cases are considered Pass if implemented on any page (not necessarily
    homepage). Please patiently review all pages (including scrolling down, clicking
    buttons to explore) before ending testing. You must understand relationships
    between pages - the first page you see is the target app's homepage.

2. If images in tested app(or webpage) modules aren't displaying correctly, that
    test case fails.

3. You may switch to other pages on the app(or webpage) during testing. On these
    pages, just confirm the test case result - don't mark other pages-passed cases
    as Fail if subpages lack features. Return to homepage after judging each case.

4. Trust your operations completely. If expected results don't appear after an
    operation, that function isn't implemented - report judgment as False.

5. If target module isn't found after complete app(or webpage) browsing, test case
    result is negative, citing "target module not found on any page" as basis.

6. Don't judge functionality solely by element attributes (clickable etc.) or text
    ("Filter by category" etc.). You must perform corresponding tests before
    outputting case results.

7. When tasks require operations for judgment, you must execute those operations.
    Final results can't have cases with unknown results due to lack of operations
    (clicks, inputs etc.).

8. For similar test cases (e.g., checking different social media links), if you
    verify one link works, you can assume others work normally.
```

```
For each individual test case completion, you can use Tell action to report just
    that result:
Tell ({"case_number": {"result": "Pass/Fail/Uncertain", "evidence": "Your evidence
    here"}})

Even in these failure cases, you must perform sufficient testing steps to prove your
    judgment before using the Tell action to report all results.

[VERIFICATION REQUIRED]: Before submitting your final report, verify that:
1. You have tested EVERY test case in the task list
2. Each test case has an explicit result (Pass/Fail/Uncertain)
3. Each result has supporting evidence based on your actual testing

Final Result Format (must include ALL test cases):
{{
    "0": {{"result": "Pass", "evidence": "The thumbnail click functionality is
        working correctly. When clicking on "Digital Artwork 1" thumbnail, it
        successfully redirects to a properly formatted detail page containing the
        artwork's title, image, description, creation process, sharing options, and
        comments section."}},
    "1": {{"result": "Uncertain", "evidence": "Cannot verify price calculation
        accuracy as no pricing information is displayed"}},
    "2": {{"result": "Fail", "evidence": "After fully browsing and exploring the web
        page, I did not find the message board appearing on the homepage or any
        subpage."}}
}}
**Return only the result string. Do not include any additional text, markdown
    formatting, or code blocks.**
```

Result Evaluation

Assessment Prompt:

```
The model results are labeled as ground truth. Please judge whether the described
    test case has been successfully implemented based on the facts. If there is
    evidence that it has been implemented, just output "Yes", otherwise output "No".
    If the model results indicate that the outcome cannot be determined, output
    "Indeterminate":
Test Case Description: {task_desc}
Model Result: {model_output}

Only answer with "Yes", "No", or "Indeterminate"
```

C.1 Agent Execution Case Study

This section presents case studies designed to demonstrate the agent's ability to evaluate applications across a range of scenarios. Each case study includes the original software design requirements, the corresponding automated test cases, and the agent's evaluation results. For enhanced clarity, screenshots of the agent's operations and historical data are provided and can be accessed at our website. Analysis of these cases will illustrate **AppEvalPilot**'s dynamic testing capabilities.

D Hierarchical Action Space

AppEvalPilot's action space A strikes a balance between expressiveness and operational efficiency. The key actions include:

Table 5: **Action Space of AppEvalPilot.** *OpenApp* is designed to facilitate the rapid initialization of the testing environment for **AppEvalPilot**. The *Run action* constitutes the primary operational module of **AppEvalPilot**, enabling flexible execution of testing procedures via Python code blocks. The *Tell action* allows **AppEvalPilot** to output evaluation results. The Stop action terminates the testing process.

Action	Implementation	Purpose
Open (app)	Using shortcut keys to quickly launch the application (e.g., Win + [Search] + Enter)	Facilitates rapid context switching
Run (code)	Executes Python scripts via PyAutoGUI for mouse and keyboard emulation	Enables complex interaction sequences
Tell (answer)	Outputs test results	Provides reporting and validation
Stop	Terminates the test episode	Controls episode termination

The **Run** action facilitates compositional tasks (e.g., automated form submission), surpassing predefined action libraries in adaptability and error resilience.

E Manual Evaluation of Software Quality

Manual Evaluation Process In our evaluation process, we invited a total of 12 individuals, all of whom are professionals in the field of computer science and experienced software testing engineers. This team conducted comprehensive assessments of each collected task and the generated software projects.

For each generated application, two evaluators independently created test cases and assigned scores. The evaluators were required to focus on generating test cases that addressed page responsiveness, page functionality, interaction testing, and boundary testing. Additionally, they were also tasked with assessing executability; for projects that failed to deploy or displayed a blank page, they recorded a score of 0. Furthermore, two other evaluators performed a cross-review of the execution results based on the actual evaluation outcomes. Consequently, each application was evaluated and reviewed by four evaluators, with the entire evaluation process taking approximately 20 minutes per application.

F Accuracy Analysis

To validate the effectiveness of AppEvalPilot, we evaluated its assessment accuracy against human-provided ground truth. Across 845 manually annotated test cases, AppEvalPilot achieves a mean accuracy of **87.0%**. We define accuracy as the proportion of cases where the automated assessment (A_i) aligns with the manual evaluation (M_i):

$$\text{Accuracy} = \frac{|\{i : A_i = M_i\}|}{N}$$

where N is the total number of test cases.

To further investigate this performance, we explored the relationship between accuracy and the human-evaluated quality of the software, as visualized in Figure 5. The analysis reveals a slight negative correlation. This indicates that the agreement between AppEvalPilot and human evaluators is exceptionally high for lower-quality software, and remains robust even as the software quality increases.

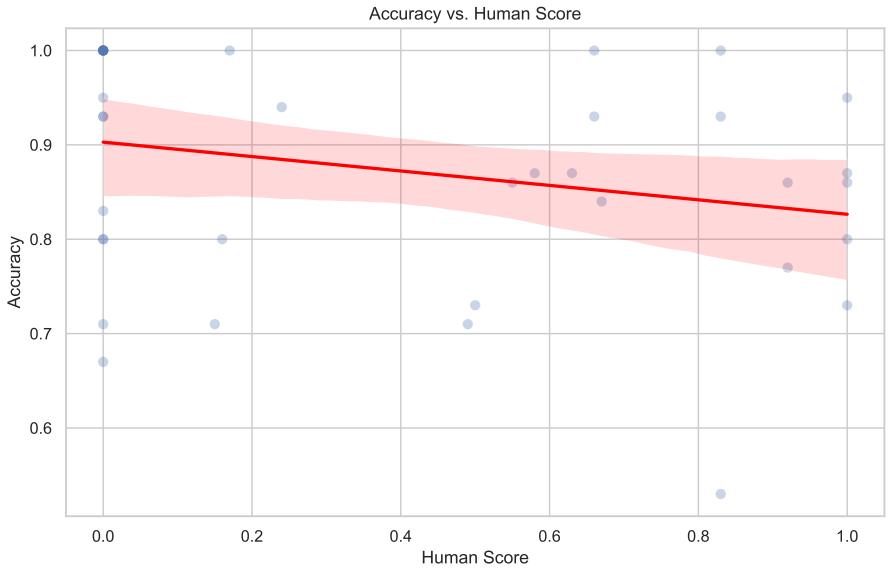
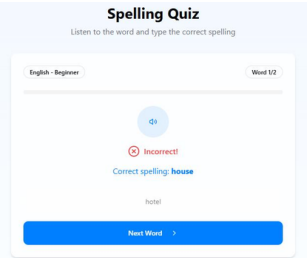


Figure 5: Accuracy of AppEvalPilot relative to human-rated software quality. The overall accuracy is 87.0%. The plot shows a slight negative correlation, suggesting that our method's agreement with human evaluators is highest on lower-quality software.

G Qualitative Analysis of Failure Modes

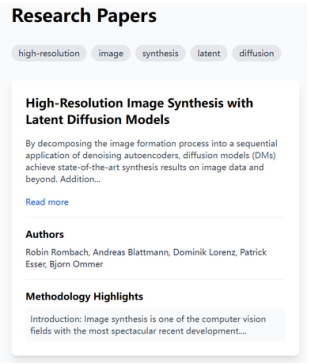
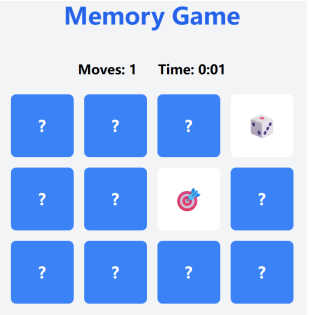
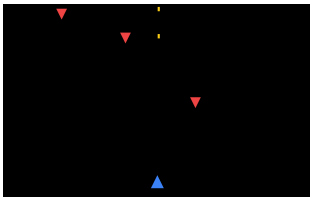
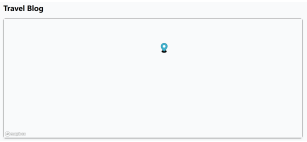
To provide a deeper understanding of our agent's behavior, we conducted a qualitative analysis of identified failure cases. This analysis reveals the characteristic limitations of our current approach and provides a roadmap for future improvements in agentic testing. Below, we present a table summarizing common failure modes with specific examples.

Table 6: Examples of Common Failure Modes in Agentic Testing.

Project	Test Case	Failure Reason	Analysis	Screenshot
Language Spelling Bee	Verify that audio playback or definition display for quiz words functions correctly.	Missing Necessary Information	1.Lack of audio information makes the evidence insufficient. 2. The agent hallucinates a conclusion despite the insufficient evidence.	
MoodMaker	Test if the generated playlist contains between 10-15 songs.	Model Hallucination	The LLM hallucinates. It correctly identifies that there are 3 songs but fails to recognize that 3 is not within the 10-15 range.	

Continued on next page

Table 6 – continued from previous page

Project	Test Case	Failure Reason	Analysis	Screenshot
Research Paper Gallery	Click on a paper title to check if it navigates to the paper's details page.	Low-quality Test Cases	The generated test case was not aligned with the actual implementation. The "details page" was accessible by clicking "read more," not the title, but the test case was marked as failed for not adhering to the overly specific instruction.	
Memory Match	Flip all paired cards to verify if the game correctly identifies the completed state.	Need for Advanced Reasoning Ability	The task requires the agent to possess strong logical thinking and memory skills to track and match pairs.	
Space Shooter	Press the spacebar or the designated shoot key to check if the spaceship fires a projectile.	Need for Real-time Feedback	A significant time lag exists between the agent's observation and its action. By the time the agent decides to act, the environment has already changed.	
Travel Blog	Check if the webpage displays a map module.	Differences in Test Standards Understanding	The agent interpreted the test case literally, passing it as long as a map module was visible. However, the actual requirement implied that the map module must also be fully functional.	

H Metrics

For **effectiveness**, we assess the quality and functionality acorss metrics (as shown table 2):

- Software Quality: it is quantified through the systematic evaluation of discrete feature test outcomes, where each feature is assessed on a dichotomous scale (0 denoting failure, 1 indicating success). The composite SQ score for each project is derived as the arithmetic mean of all feature-specific test pass rates, expressed as:

$$SQ = \frac{1}{n} \sum_{i=1}^{n} f_i$$

where n represents the total number of features and f_i denotes the test outcome of the i-th feature.

- Executability: measures whether the generated project deploys and executes without errors. Projects are tested using a standard launch script, with successful execution scored as 1 and failures as 0.

For **efficiency**, we measure resource utilization through:

- Cost: Tracks the number of tokens used during project generation.
- Time: Measures the total elapsed time for generating the project, from requirement input to final output.

To assess generated project complexity quantitatively, we analyze three key technical metrics as **statistics** :

- Total Files: counts all project files through recursive directory traversal, including code, configuration, and documentation files.
- Code Files: counts only source code files, identified by extensions (e.g., `.java`, `.py`, `.cpp`).
- Lines of Code: quantifies the total code lines as a measure of project complexity and development effort.

I Agent-as-a-Judge

Our evaluation framework implements three complementary metrics that collectively evaluate AI-generated software quality:

I.1 LLM Scorer for Completeness

We employ an LLM-as-a-Judge (Zheng et al., 2023) and Agent-as-a-Judge (Zhuge et al., 2024) to quantitatively evaluate implementation completeness against specified requirements:

- **Methodology:** Systematic analysis of project code against requirements specification and feature list
- **Assessment Approach:** Per-feature evaluation of implementation completeness with code organization analysis
- **Scoring Scale:** Normalized 0-100 scale with feature-weighted scoring

I.2 LLM-score

The composite metric combines implementation completeness with executability:

$$\text{LLM-score} = \text{Completeness} \times \text{Executability} \tag{1}$$

Where Executability serves as a binary multiplier (1 or 0). This formulation enforces a critical principle in software evaluation: high-quality software must both implement required features comprehensively and function correctly in practice. The approach effectively penalizes implementations with strong theoretical completeness but practical execution failures, aligning our metric with real-world software quality standards.

I.3 LLM Scorer Prompt Template

LLM Scorer Prompt Template

```
# Evaluation criteria
When evaluating the code, the following must be strictly followed:
1. Verify each task separately
- Task splitting: According to the task sequence, verify the completion and quality
    of each task separately
- Functional check: Verify whether the function has been implemented in the project
    code, and strictly verify whether the logic of the implementation code is correct
- Intrinsic code quality: Ignore the length or number of lines of code. Focus on
    evaluating the efficiency, readability and maintainability of the code.
    Prioritize best practices, optimal performance, clear and logical structure, and
    ease of future modification, so as to focus on the intrinsic advantages of the
    code rather than its size. Don't judge a code as more accurate just because it
    is longer
- Analysis-type tasks: For the implementation of graphs, tables, etc. mentioned in
    the task, it is necessary to analyze whether it has the corresponding graph or
    table code implementation from the perspective of code implementation
- Effect display: For the effect display task, the elements of the display code and
    the logic of the world display need to be strictly verified in the project code
    to see if they meet the task requirements
- Data verification: If an attachment file is provided in the user requirements, it
    is necessary to strictly judge whether the content of the attachment file (or
    the reference of the attachment file) is obtained in the project code
- Website display: Regarding the tasks of web page buttons and page display, it is
    necessary to globally evaluate whether the corresponding display code and
    button-related codes are legal and logically correct in the entire project;
    logically evaluate whether an error will be reported after running the code.The
    dataset file will only give one data segment, including the corresponding column
    names

2. Library, package and non-manufactured element verification
- Compliance: Check whether the use of libraries/packages meets the requirements of
    the project and follows best practices
- Library/package authenticity: Ensure that the code does not import or introduce
    non-existent libraries or packages. Flag any fabricated or false information.
- Compliance with real-world standards: Confirm that all components used in the code
    are real, supported, and meet real-world standards and practices.

Combining each of the two major points of the above evaluation criteria, each
    evaluation point needs to be strictly and objectively evaluated, especially when
    it comes to page image display, button interaction, and website data (links), it
    is necessary to strictly verify whether the global logic of the code is correct;
    each task is strictly evaluated and scored. Once the relevant code of a task is
    found to have logical or global call errors, etc., then the task will be
    directly marked as failed (when evaluating the logic and correctness of the
    code, ignore the comment information in the code directly, because it is likely
    to be an error; the code needs to be strictly evaluated directly)

# Output Format
Output the evaluation results as a JSON object containing one main sections:
    `evaluations`.
{
  "task_id": "string, unique identifier for the `Tasks`.",
  "direct_judgment": boolean, // True if correctly implemented without logical errors
  "score": integer, // 0-100, based on strict verification against evaluation criteria
  "satisfied": boolean, // True if score > 75 and meets requirements
  "reason": "string, 3-5 sentence explanation with step-by-step reasoning."
}
```