You Don't Know Until You Click: Automated GUI Testing for Production-Ready Software Evaluation

Yutong Bian[1], **Xianhao Lin**[2], **Yupeng Xie**[3], **Tianyang Liu**[4], **Mingchen Zhuge**[5], **Siyuan Lu**[6], **Haoming Tang**[1], **Jinlin Wang**[1], **Jiayi Zhang**[1,3], **Jiaqi Chen**[7], **Xiangru Tang**[8], **Yongxin Ni**[9], **Sirui Hong**[1], **Chenglin Wu**[1]*

[1]DeepWisdom [2]Fudan [3]HKUST(GZ) [4]UC San Diego [5]KAUST [6]Westlake [7]Stanford [8]Yale [9]NUS

alexanderwu@deepwisdom.ai

Abstract

Large Language Models (LLMs) and code agents are rapidly evolving from generating isolated snippets to building full-fledged applications with graphical interfaces, interactive logic, and dynamic behaviors. However, current benchmarks fall short as they often rely on static checks or binary pass/fail scripts, failing to capture the interactive behaviors and runtime dynamics that define real-world usability — in other words, *you don't know if an app works until you click through it, interact with it, and observe how it responds.* To bridge this gap, we introduce **RealDevWorld**, a novel framework for automated end-to-end evaluation of production-ready repositories from scratch. It features two components: (1) **RealDevBench**, a diverse set of 194 open-ended software engineering tasks across different domains and modalities; and (2) **AppEvalPilot**, an agent-as-a-judge system that simulates realistic GUI-based interactions to automatically and holistically assess functional correctness, visual fidelity, and runtime behavior. RealDevWorld delivers fine-grained, task-specific diagnostics beyond binary judgments and aligns strongly with human assessments (accuracy 0.92, correlation 0.85), while substantially reducing manual review. This enables scalable, human-aligned evaluation of LLMs' ability to generate production-level software.

1 Introduction

Remarkable advancements in LLMs for code and autonomous coding agents are driving a paradigm shift in software development. Their generative capabilities are evolving from function-level code snippets, to crafting self-contained demos, and now towards the creation of sophisticated, production-ready repositories featuring intuitive user interfaces, modular architectures, and robust runtime integration. However, this evolution poses significant challenges for evaluation. Current repository-level code generation tasks lack rigorous assessments of functional completeness, especially with respect to dynamic and interactive user-centric behaviors. For example, consider a game application generated by such a system. Its correctness and quality cannot be reliably determined by code inspection or static analysis alone. Instead, it requires user-centric validation: clicking through

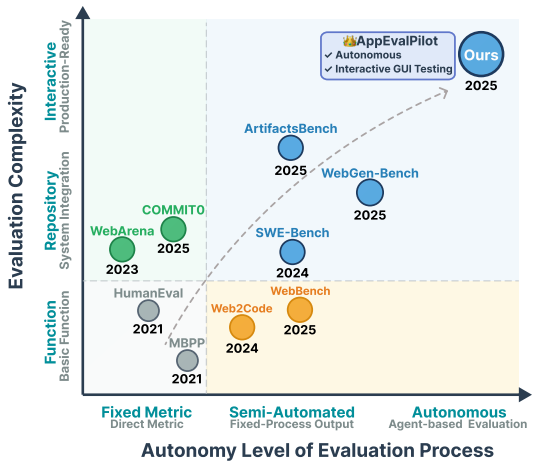


Figure 1: Software Engineering Evaluation: From Automated to Autonomous Evaluation

*Corresponding author.

Benchmark	Lang.	Level	Tasks	Eval Method	Agent Judge	Input Data	Interactive
BigCodeBench [1]	PY	Func.	Comp.	Unit test	✗	Text, Code	✗
LiveCodeBench [2]	PY	Func.	Gen.	Unit test	✗	Text, Code	✗
RepoBench [6]	PY, Java	Repo.	Ret.	Similarity	✗	Text, Code	✗
SWE-Bench [9]	PY	Repo.	Maint.	Unit test	✗	Multi-modal	✗
EvoCodeBench [5]	PY	Repo.	Ret.	Pass@k	✗	Text, Code	✗
SWE-Lancer [10]	JS, TS	Repo.	Dev.	Unit test	✗	Multi-modal	✗
FrontendBench [11]	JS	Repo.	Gen.	Unit test	✗	Text	✓
COMMIT0 [12]	PY	Repo.	Dev.	Unit test	✗	Multi-modal	✗
Web-Bench [13]	JS, TS	Repo.	Dev.	Unit test	✗	Text	✗
RealDevWorld	PY, JS, TS	Repo.	Dev.	Unit test	✓	Multi-modal	✓

Table 1: **Comparison of RealDevWorld with existing benchmarks.** It leverages AppEvalPilot 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.*

the interface, interacting with game elements, observing state transitions, and receiving feedback in real time—actions that reflect how an actual user would engage with the system. These user-centric and runtime-dependent behaviors are difficult to capture through conventional metrics and often demand the execution of complex end-to-end (E2E) test cases on the generated front-end to assess correctness, interaction quality, and behavioral robustness. However, automating such evaluations remains challenging: generated repositories frequently vary in visual layout, interaction flow, and execution paths, making static or script-based evaluations brittle and often infeasible.

Current benchmarks fall short in automatically assessing the functional completeness and real-world applicability of production-ready repositories, as illustrated in Figure 1. Function-level benchmarks [1–3] primarily focus on isolated generation tasks, such as function or class implementation, which fail to capture the complexity and dynamic interactions of real-world repository-level applications. Repository-level benchmarks [4–10] attempt to assess entire codebases, yet commonly rely on static or predefined evaluation methods, such as code similarity metrics, unit tests, or scripted integration tests, that are inherently brittle and limited. These methods struggle to reflect real-time interactions, user-driven workflows, runtime errors, or the diverse visual and structural variability of generated outputs. Real-world applications, especially those involving user interfaces, documentation, and multimodal content, exhibit dynamic, unpredictable behaviors. Evaluating them accurately demands intelligent, adaptive methods capable of systematically capturing runtime interaction fidelity and user-centric correctness, highlighting the urgent need for more comprehensive evaluation frameworks.

Recent advances in interactive agent technology offer promising directions toward this goal. Emerging paradigms, such as Agent-as-a-Judge [14], employ autonomous agents that execute end-to-end tests by emulating human behaviors, monitoring runtime states, and capturing detailed execution traces. Such agents transcend traditional static metrics, treating evaluated applications not merely as passive test subjects, but as dynamic, interactive environments that inform agent reasoning and decision-making. Building upon this paradigm, we present **ReaDevWorld**, a comprehensive evaluation framework explicitly designed to assess AI-generated, production-ready codebases through dynamic interaction and open-ended testing scenarios. As part of this framework, we introduce **RealDevBench**, a benchmark of 194 carefully curated open-ended software engineering tasks across display, analysis, data, and game domains. These tasks are sampled from the real-world programming community requirements and systematically expanded at the function level using LLMs, with a subset incorporating multimodal complexity (structured data, images, audio) to reflect real-world challenges. Table 1 highlights how RealDevBench differs from existing evaluation datasets. To operationalize this benchmark, we develop **AppEvalPilot**, a novel agent-based evaluation framework that emulates human interactive software engineering practices. Given a task description and generated code, AppEvalPilot integrates web and OS-level operations to simulate testing workflows, conducting both functional and boundary evaluations for comprehensive software development verification. This agent serves as an automated and effective testbed for production-ready software engineering. Our main contributions are:

A GUI-Interactive Agent-as-a-Judge Paradigm for Automated Evaluation. We present AppEvalPilot, a novel agent-as-a-judge evaluation paradigm for production-ready code generation in complex, dynamic interaction scenarios. By simulating realistic user behavior and performing

runtime GUI interactions, AppEvalPilot enables fine-grained diagnostics comparable to white-box testing in traditional software engineering.

An Open-ended and Scalable Benchmark Suite. RealDevBench features a diverse set of tasks derived from real-world programming needs, spanning domains like display, analysis, data, and gaming. It benchmarks the ability of code intelligence models to build repository-level software from scratch, with tasks incorporating multimodal inputs—such as images, audio, text, and structured data—to increase reasoning difficulty and scenario realism.

Human Alignment and Cost-Effective Validation. Our framework achieves strong alignment with expert human assessments, reaching an accuracy of 0.92 and a correlation of 0.85, substantially outperforming existing automated evaluators. By narrowing the gap between model-based and human evaluation, it enables more reliable and cost-effective validation of generated code.

2 Related Work

2.1 Benchmarks for Software Engineering

Evaluating repository-level code generation in LLM-based agents remains challenging due to the complexity of end-to-end software development, including system integration, dependency management, and dynamic interactions [14]. Existing benchmarks such as BigCodeBench [1], LiveCodeBench [2], and NaturalCodeBench [3] focus on function- or class-level code completion and rely primarily on static test cases, failing to capture dynamic behaviors like web interfaces or gameplay [15, 16]. As a result, they fall short in assessing real-world development challenges such as integration, ambiguous specifications, and evolving requirements. Repository-level benchmarks [4–10] tackle broader software tasks with interdependent components, but mainly use static metrics like similarity scores or unit tests [17, 18], which may not fully reflect functional correctness. Advanced benchmarks like rSDE-Bench [8], SWE-Bench [9], and SWE-Lancer [10] depend on pre-defined test cases, limiting their ability to evaluate adaptability to requirement changes or the creation of new modules. DEVAI [14] and MLE-Bench [19] introduce automated development tasks for agent evaluation but rely on public datasets, which may be seen during model training. In contrast, our proposed benchmark supports adaptive module development and dynamic interaction testing, simulating human-like evaluation processes to more comprehensively assess software development capabilities.

2.2 Advanced Judgement Approaches

Recent evaluation techniques have established new paradigms, starting with LLM-as-a-Judge [20], 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 [14] builds on this by introducing a dynamic agent-based approach, leveraging multi-dimensional scoring and iterative feedback loops [21]. 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 [22, 23], to mirror human testing processes for a more dynamic and comprehensive evaluation. We summarized the comparisons in Table 1.

3 Preliminary

This section formalizes the task of end-to-end software evaluation and analyzes three mainstream evaluation paradigms—human evaluation, LLM-as-a-Judge [24], and Agent-as-a-Judge [14]-in terms of their coverage across software quality dimensions, laying the foundation for subsequent experiments and theoretical analysis.

3.1 End-to-End Software Evaluation

As previously discussed in the introduction, end-to-end testing is essential for assessing production-ready software development. Formally, a generator $\mathcal{A}$ (e.g., a human developer or an AI system) receives a requirement instance $Q = (D, F, M)$, where D is the requirement description, F is the list

of desired features, and M represents any supplementary materials. Given this input, the generator is expected to produce a complete software repository R. The goal of end-to-end evaluation is to design an effective method to measure the quality of R. Unlike unit testing that focuses on individual components, end-to-end evaluation validates user workflows across all system layers, ensuring the entire software system functions correctly in realistic usage scenarios. This challenge is particularly significant for complex software in real-world, open scenarios, where code structure and interaction are often unpredictable.

3.2 Formalization and Evolution of Evaluation Workflows

According to software engineering standards and validation research (ISO/IEC/IEEE 29119 [25], SV-COMP [26]), production-grade software must undergo comprehensive validation at three levels: **unit level** (individual code components), **system level** (architecture and integration), and **acceptance level** (user interactions and dynamic behaviors). Only by satisfactorily meeting all three levels can software be deemed production-ready. We model the end-to-end evaluation process as a unified pipeline that transforms the general evaluation workflow into concrete implementations:

$$(Q, R) \xrightarrow{\text{Identify}} C \xrightarrow{\text{Execute}} \mathrm{T} \xrightarrow{\text{Judge}} S \tag{1}$$

where from task description Q and repository R, test cases C are identified. These test cases are executed to collect execution traces **T**, and **Judge** analyzes these traces to produce the final software quality score S. The key differences between evaluation paradigms lie in how test cases C are identified given Q and R, how these C are executed to collect traces T, and how Judge analyzes these traces to produce S. The three mainstream evaluation paradigms are as follows.

Human evaluation workflow: Human experts participate in the entire process, covering unit, system, and acceptance levels. In this paradigm, experts manually analyze requirement Q and repository R, design test cases C based on features F. The test cases are executed manually to generate comprehensive $\mathrm{T}_{\text{manual}}$ that covers all validation levels such as unit testing, system testing, and acceptance testing. Subsequently, $\text{Judge}_{\text{human}}$ analyzes the manual traces to produce quality score S_{manual}, e.g. test coverage and pass rates. The advantage is comprehensiveness, but the disadvantage is high cost and low efficiency due to the manual nature of the entire process.

LLM-as-a-Judge workflow: A typical implementation is automatic scoring based on static code analysis (e.g., ArtifactsBench). In this approach, $\text{Execute}_{\text{static}}$ extracts code fragments via fixed scripts or paths, generating limited test cases C only from static code inspection rather than from the original feature list F. This produces $\text{Trace}_{\text{static}}$ consisting of static text representations, which $\text{Judge}_{\text{LLM}}$ analyzes through text-based reasoning to generate S_{static}. This method only covers the unit and part of the system level, cannot detect runtime or interaction issues, and has limited reliability due to the static nature of both $\text{Execute}_{\text{static}}$ and $\text{Trace}_{\text{static}}$.

Interactive agent-as-a-judge workflow: The agent can automatically understand requirements and decompose features from F to generate comprehensive test cases C. During evaluation, $\text{Execute}_{\text{agent}}$ executes these C through GUI interactions with R, dynamically collecting execution results to form $\text{Trace}_{\text{agent}}$ that captures real-time behaviors and user interactions. $\text{Judge}_{\text{agent}}$ then analyzes these dynamic traces to produce S_{agent}. This method can automatically cover all three dimensions—unit, system, and acceptance levels—combining depth and scalability, making it ideal for production-grade evaluation. This framework provides the theoretical foundation for our RealDevBench benchmark and AppEvalPilot evaluation system, which we detail in the following sections.

4 RealDevBench: Open-Ended SE Benchmark

4.1 Dataset Overview

To comprehensively evaluate AI systems across these dimensions, we introduce **RealDevBench**, a benchmark specifically designed to assess end-to-end software engineering capabilities in a realistic and practical context. **RealDevBench** comprises 194 requirements spanning four practical domains—*Analysis, Display, Data*, and *Game*, that reflect core engineering needs. The distribution of tasks is as follows: Display (50.0%), Data (14.4%), Analysis (18.6%), and Game (17.0%), as illustrated in Figure 5. This allocation mirrors the prevalence of web-centric and data-intensive applications in real-world software development. The dataset is defined by three key attributes: (1)

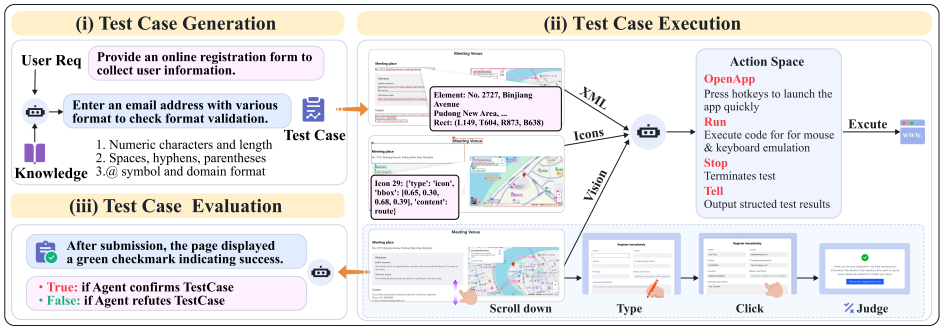


Figure 2: Overall design of AppEvalPilot showing the automated testing workflow: test case generation from user requirements, multimodal test execution through interface interaction (scrolling, typing, clicking), and binary evaluation of outcomes for objective software assessment.

Open-ended repository construction, where systems must build software from scratch rather than fill in predefined templates; (2) Multimodal complexity, incorporating diverse inputs such as text, images, audio, and tabular data to test integrative and cross-modal capabilities; (3) Functional diversity, encompassing a wide spectrum of software functionalities across varying levels of complexity.

4.2 Dataset Construction

Domain and Requirement. We examined WebDev Arena [27] to establish 4 domain categories: **Display**, **Analysis Data**, and **Game**. We sampled requirements from SRDD [28] and expanded through web crawling freelancer platforms (Upwork[2] and Freelancer[3]) to capture real client demands.

Feature Construction. To construct detailed feature lists that extend requirements from development and functional perspectives, we learned from open-source projects and performed systematic feature extraction. We crawled GitHub projects meeting strict selection criteria: comprehensive documentation (README, API docs), production-ready quality (1000+ stars, active development), and clear feature specifications. We employed Claude-3.5-Sonnet [29] to extract functional requirements from repository documentation and expand requirements into structured feature specifications, ensuring consistent translation of requirements into actionable features with clear evaluation criteria.

Task Structure and Formulation. As shown in Figure 5, each task in **RealDevBench** is structured as a triplet to simulate realistic software development scenarios: (1) Requirements Description: A textual summary outlining the project's purpose and setting; (2) Feature List: A detailed and structured list of functional goals that define the success criteria; (3) Supplementary Materials: Task-specific resources such as images, audio, or datasets that introduce real-world complexity. To further enhance the realism of each task, we incorporated carefully curated materials from multiple sources: (1) Images: Sourced from Unsplash [4] for thematic relevance and professional quality; (2) Datasets: Selected from Kaggle [5] based on topic relevance and appropriate complexity; (3) Documents: Manually created documents (resumes, business proposals, catalogs) that mirror real-world scenarios.

5 AppEvalPilot: Autonomous Evaluation

As discussed previously, the rise of AI-driven software development demands scalable, automated, and adaptive evaluation methods. To achieve this, we introduce AppEvalPilot, an Agent-as-a-Judge evaluation paradigm designed for automated end-to-end interaction-based software project testing. Unlike static analysis or rigid test suites, AppEvalPilot actively engages with software interfaces, executing real-time user interactions to assess functional correctness and adaptability. As illustrated in Figure 2, the evaluation framework follows a three-stage pipeline: (1) generate test cases based on requirements and domain knowledge; (2) simulate real-world user interactions via textual and visual inputs; (3) assess correctness and completeness by comparing actual outcomes with expected

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

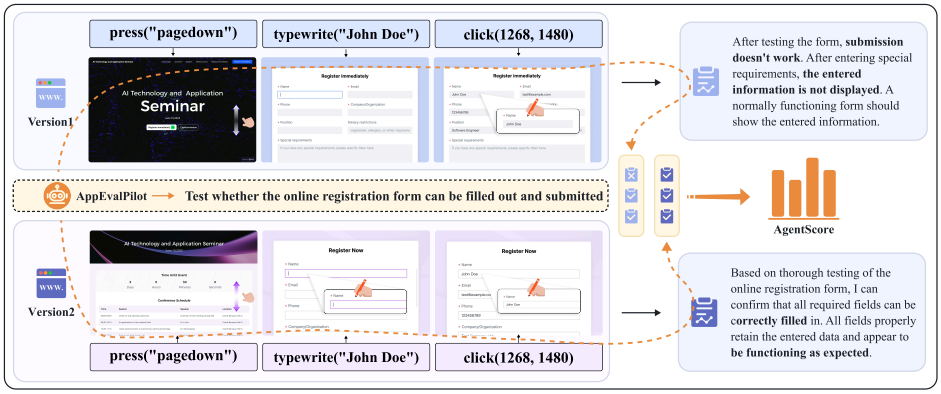


Figure 3: Evaluation pipeline of AppEvalPilot. The agent performs test sequences on two different web implementations, systematically assesses functionality through direct interaction, documents observable differences in form behavior, and generates quantitative scores based on test cases.

behaviors. This dynamic and automated approach aligns with **RealDevBench**'s focus on practical software evaluation, enabling scalable and rigorous assessment of AI-generated systems.

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 [30] to infer requirement-to-test mappings from a small set of manually curated examples, allowing it to generalize efficiently across diverse software requirements. 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 standardize generation, the agent uses a structured prompt that simulates the behavior of a professional test engineer. The number of cases is capped (e.g., 15–20) to ensure evaluation tractability.

Test Case Execution. AppEvalPilot next autonomously executes the generated test cases by directly interacting with software applications through their GUIs, 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 (see Appendix B.1. These atomic actions, as shown in 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 [31] 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 compares actual interaction outcomes against the expected success criteria defined in **RealDevBench**. The agent autonomously executes interaction workflows across different application implementations, adapting its actions based on each interface while maintaining consistent testing objectives. Specifically, after each test execution, AppEvalPilot generates a structured report that documents both the performed actions (e.g., entering user information, submitting a form) and the resulting behaviors (e.g., form submission success, data persistence). Based on observed outcomes, AppEvalPilot classifies each test case into one of three categories: **Pass** (expected behavior is met), **Fail** (expected behavior is violated), or **Uncertain** (outcome is inconclusive or partially observed). These classifications feed into an aggregated score on test case or feature levels, offering a quantitative assessment of the software quality.

As illustrated in Figure 3, the agent runs similar interaction sequences across different implementations and determines test case satisfaction by comparing observed execution results against specified

requirements. This autonomous execution approach enables the agent to make informed judgments about requirement satisfaction by directly observing how different implementations respond to similar user interactions. This process not only surfaces hidden behavioral issues but also ensures that the evaluation remains scalable, interpretable, and grounded in observable user-level feedback.

6 Experiments

We conduct comprehensive experiments to validate AppEvalPilot's evaluation capabilities and its effectiveness in benchmarking software development systems. Our experimental design addresses two critical research questions: (1) How effectively does AppEvalPilot evaluate software quality compared to existing evaluation approaches? and (2) Can AppEvalPilot serve as a reliable automated judge for benchmarking LLM-based software engineering?

6.1 AppEvalPilot Capability Validation

Dataset. We construct our evaluation dataset by selecting 49 tasks (25%) from RealDevBench, ensuring coverage across all domains. We first fix the generated software projects using Lovable [32] and establish reliable human ground truth labels through a rigorous two-level evaluation process: (1) *Test case-level*: For test cases c_i generated by AppEvalPilot, we invite 3 QA specialists (1-3 years experience) to execute each test case and evaluate Pass/Failed/Uncertain outcomes; (2) *Feature-level*: Each project also receives independent scoring from 3 QA specialists who manually test generated software projects against feature lists, providing granular scores for each feature $f_i \in \{0, 1\}$ (Failed/Pass), with final validation by a senior expert. Therefore, each project quality is recorded as human_quality $= \frac{1}{n}\sum_{i=1}^{n} f_i$ where n represents the total number of features.

Baselines. We compare against state-of-the-art GUI systems: Claude-3.5-Sonnet-v2 [29], UI-Tars [33], WebVoyager-Agent [34] with qwen2.5-vl-32B [35] and claude-3.5-sonnet-v2 backbones, and Browser-Use with claude-3.7-sonnet-v2 [36]. Framework protocols provide high-level requirements for autonomous test strategy decomposition, while model protocols provide pre-generated test cases aligned with their input paradigms.

Metrics. Given test case set $C = \{c_1, c_2, ..., c_N\}$ or feature list $F = \{f_1, f_2, ..., f_M\}$, each item is classified as true, false, or uncertain by human evaluators or agents. We define binary scores as:

$$\text{score}_i = \begin{cases} 1 & \text{if class}_i = \text{true} \\ 0 & \text{if class}_i \in \{\text{false}, \text{uncertain}\} \end{cases}$$

We use **accuracy** to measure judgement correctness and **quality alignment** using Pearson correlation at *test case-level* and *feature-level*, where test case-level represents averaged performance across all test cases in each project, and feature-level measures correlation between agent and human feature scores across all software projects.

Results & Analysis. AppEvalPilot demonstrates superior performance across all evaluation metrics. Our framework achieves an accuracy of 0.92 in test case classification and a quality alignment correlation of 0.81 with human evaluators, representing a 47% improvement over WebVoyager (Claude-3.5-Sonnet) which achieved 0.55 accuracy alignment. Compared to baseline GUI testing approaches like Browser-Use [37], AppEvalPilot reduces evaluation time by 33% (from 13.50 to 9.00 minutes per app) while achieving 77% cost reduction through its interactive-driven paradigm. At the feature level, AppEvalPilot maintains the highest alignment with human assessments, achieving 0.85 correlation across diverse application domains compared to Browser-Use's 0.58, representing a 47% improvement and validating its effectiveness in end-to-end automated evaluation. End-to-end automated software testing presents significant challenges for existing GUI models and agents, requiring sophisticated planning capabilities and execution accuracy, where traditional GUI tasks primarily focus on fine-grained operational requirements similar to individual test case granularity. When utilizing test cases provided by AppEvalPilot, all baseline models showed an average improvement of 0.17, demonstrating the value of our test case generation approach. Our observations reveal that detailed test cases not only improve GUI agent testing success rates but also enhance testing robustness, since each feature is decomposed into multiple supporting test cases where incorrect judgment on one test case does not affect the results of other test cases, thereby improving the robustness and reliability of the overall testing process.

Method	Feature-level		Test Case-level			Efficiency	
	Quality	Align.	Quality	Align.	Acc.	Time	Cost
Human	0.74	–	0.65	–	–	–	–
GUI Model							
Claude-3.5-Sonnet	0.27	0.23	0.46	0.49	0.68	9.20	1.01
UI-Tars	0.49	0.29	0.63	0.59	0.75	8.65	0.17
GUI Agent Framework							
WebVoyager (Qwen2.5)	0.29	0.25	0.35	0.44	0.6	2.16	**0.04**
WebVoyager (Claude)	0.64	0.43	0.6	0.55	0.74	**1.60**	0.10
Browser-Use (Claude)	0.67	0.58	0.63	0.61	0.76	13.50	1.13
AppEvalPilot(Claude)	0.73	**0.85**	0.74	**0.81**	**0.92**	9.0	0.26

Table 2: Performance comparison on RealDevBench benchmark. Human Quality (GT) represents ground truth project quality scores from human evaluation. Quality Alignment measures correlation with human assessments.

Figure 4: Comparative analysis of evaluation methods versus human quality. (Left) AppEvalPilot's autonomous evaluation, (Middle) Static LLM code scoring, (Right) Visual aesthetic scoring. Each point represents one project, with radial distance indicating quality scores (0-1 scale).

Comparative Evaluation Analysis. To comprehensively validate AppEvalPilot's evaluation effectiveness, we conduct systematic comparative analysis across multiple evaluation methodologies using the same 49 Lovable-generated projects. Our comparison encompasses static evaluation methods as illustrated in Figure 4: (1) Code Quality assessment [24] employing integrated Claude-3.5-Sonnet scoring of source files, and (2) Visual Quality evaluation utilizing Claude-3.5-Sonnet aesthetic scoring with WebGen-Bench prompts [38]. As demonstrated in Figure 4, both Code Quality and Visual Quality fail to effectively capture the nuances of software quality, in contrast to Agent Quality, which shows a strong alignment with human assessments. Our analysis reveals critical shortcomings in existing LLM-as-a-judge and MLLM-as-a-judge approaches. First, static evaluation cannot capture dynamic interaction issues that define software quality—the deviation means for Code Quality and Visual Quality are 2.79× and 3.34× higher than AppEvalPilot's Agent Quality, respectively, demonstrating substantial gaps between static assessment and actual user experience. Second, evaluation distributions exhibit pronounced misalignment with human judgment: AppEvalPilot achieves a distribution overlap rate of 0.96 with human scores, while Code Quality and Visual Quality achieve mere 0.75 and 0.55 overlap rates, indicating fundamental divergence from natural evaluation patterns. These findings underscore the superiority of our agent-based evaluation framework in capturing multifaceted software quality aspects that traditional static methods systematically overlook. AppEvalPilot's dynamic interaction capabilities enable accurate quality assessment that closely mirrors human evaluation standards while providing actionable feedback for developers, demonstrating clear advantages over existing static evaluation paradigms.

System	Agent Quality	Code Quality	Visual Quality
Large Language Models			
Claude-3.7-Sonnet	0.31	0.41	0.18
Gemini-2.5-Pro	0.29	0.45	0.26
Kimi-K2	0.39	0.41	0.29
DeepSeek-V3	0.29	0.18	0.21
Qwen3-Coder-480B	0.53	0.41	0.32
Qwen3-235B-Instruct	0.33	0.42	0.20
Agent Systems			
OpenHands	0.50	0.38	0.33
Lovable	0.74	0.58	0.47
Bolt	0.54	0.69	**0.50**
MGX	0.60	0.68	0.41
MGX (BoN-3)	0.78	0.72	0.41

Table 3: Comparative results across different code generation systems and evaluation methods.

6.2 Performance of LLMs on RealDevBench

Experimental Setting. Considering validation costs, we conduct experiments on 54 tasks from RealDevBench-test. The evaluated generation frameworks include MGX [39], MGX (BoN-3), Bolt [40], Lovable, OpenHands [41], Claude-3.5-Sonnet, Gemini-2.5-Pro [42], Kimi-K2 [43], DeepSeek-V3 [44], Qwen3-Coder-480B [45], and Qwen3-235B-Instruct. After code generation, we execute deployment through automated scripts and LLM-generated deployment commands. For MGX, Bolt, and Lovable, we directly utilize their pre-deployed project URLs for testing. We employ three evaluation approaches: AppEvalPilot's interactive assessment, static code quality evaluation, and visual aesthetic scoring through screenshot analysis.

Performance Analysis. RealDevBench presents significant challenges for LLMs, with even state-of-the-art models like Kimi-K2 achieving only 0.39 in software quality for generated projects. Current LLM performance on **RealDevBench** is substantially lower than their performance on traditional coding benchmarks, revealing significant defects and bugs in complete interactive functionality development and validation. Visual and static code assessment alone cannot adequately quantify these limitations and shortcomings. For agent frameworks, generation quality shows significantly higher average scores in Agent Quality, with an improvement of approximately 0.27 compared to direct LLM generation. This improvement stems from two key factors: First, these frameworks adopt standard software engineering development processes through design, development, and basic deployment verification, significantly enhancing code usability. Second, for complex interactive functionality design, agent-generated projects contain multiple files and components, providing more complete functional implementation compared to single-script solutions produced by LLMs. As shown in Table 3, static assessment methods fail to capture runtime behaviors, user interaction flows, and integration issues that are critical for real-world software functionality. This validates AppEvalPilot's interactive evaluation paradigm as essential for comprehensive software quality assessment.

7 Conclusion

In this paper, we introduce **RealDevWorld**, a novel framework for evaluating AI systems that generate code repositories from scratch. It comprises **RealDevBench**, an open-ended and scalable dataset of 194 diverse tasks with multimodal elements, and **AppEvalPilot**, a GUI-based Agent-as-a-Judge evaluation paradigm. **AppEvalPilot** performs automated, end-to-end validation of software functionality, including dynamic behaviors and interaction logic, while providing fine-grained, task-specific diagnostic feedback.

Extensive experiments show that our framework closely aligns with expert human judgments while significantly reducing evaluation time and cost. On the **RealDevBench** benchmark, **AppEvalPilot** substantially outperforms existing GUI frameworks, achieving an accuracy of up to 87%. Overall, **RealDevWorld** offers a scalable and automated solution for reliable software evaluation, paving the way for future advancements in production-ready code generation.

References

[1] 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.

[2] 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.

[3] 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*, pages 7907–7928, 2024.

[4] 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.

[5] 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.

[6] 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.

[7] 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*, pages 2471–2484, 2023.

[8] 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.

[9] 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.

[10] 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.

[11] Hongda Zhu, Yiwen Zhang, Bing Zhao, Jingzhe Ding, Siyao Liu, Tong Liu, Dandan Wang, Yanan Liu, and Zhaojian Li. Frontendbench: A benchmark for evaluating llms on front-end development via automatic evaluation. *arXiv preprint arXiv:2506.13832*, 2025.

[12] Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch. *arXiv preprint arXiv:2412.01769*, 2024.

[13] Kai Xu, YiWei Mao, XinYi Guan, and ZiLong Feng. Web-bench: A llm code benchmark based on web standards and frameworks. *arXiv preprint arXiv:2505.07473*, 2025.

[14] 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.

[15] 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.

[16] 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.

[17] 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)*, pages 31–53. IEEE, 2023.

[18] 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*, pages 13785–13816, 2024.

[19] 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.

[20] 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.

[21] Yupeng Xie, Yuyu Luo, Guoliang Li, and Nan Tang. Haichart: Human and AI paired visualization system. *Proc. VLDB Endow.*, 17(11):3178–3191, 2024. doi: 10.14778/3681954.3681992. URL https://www.vldb.org/pvldb/vol17/p3178-luo.pdf.

[22] 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)*, pages 1–9. IEEE, 2024.

[23] 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.

[24] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 46595–46623. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/91f18a1287b398d378ef22505bf41832-Paper-Datasets_and_Benchmarks.pdf.

[25] Software and systems engineering — software testing — part 1: General concepts, 2022. URL https://www.iso.org/standard/81291.html.

[26] Dirk Beyer. State of the art in software verification and witness validation: Sv-comp 2024. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science. Springer, 2024. URL https://link.springer.com/chapter/10.1007/978-3-031-57256-2_15.

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

[28] OpenBMB. Srdd. https://github.com/OpenBMB/ChatDev/tree/main/SRDD, 2024. Accessed: 2025-03-29.

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

[30] 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.

[31] 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.

[32] Lovable Team. Lovable: Ai development solution. https://lovable.dev, 2024.

[33] Y. Qin, Y. Ye, J. Fang, et al. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

[34] Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919*, 2024.

[35] Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, et al. Qwen2. 5-vl technical report. *arXiv preprint arXiv:2502.13923*, 2025.

[36] Anthropic. Claude 3.7 Sonnet. https://www.anthropic.com/claude/sonnet, 2025.

[37] Magnus Müller and Gregor Žunič. Browser use: Enable ai to control your browser, 2024. URL https://github.com/browser-use/browser-use.

[38] Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. Webgen-bench: Evaluating llms on generating interactive and functional websites from scratch, 2025. URL https://arxiv.org/abs/2505.03733.

[39] MetaGPT Team. Mgx: Ai software development platform. https://mgx.dev, 2024.

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

[41] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

[42] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

[43] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Hao Hu, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Chao Hong, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Lijun Lu, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Xinjie Sun, Flood Sung, Heyi Tang, Jiawen Tao, Qifeng Teng, Chensi Wang, Dinglu Wang, Feng Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Xiaofei Yang, Ying Yang, Zhen Yang,

Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yangkun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Huabin Zheng, Shaojie Zheng, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi k2: Open agentic intelligence, 2025. URL https://arxiv.org/abs/2507.20534.

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

[45] Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Appendix A Benchmark Samples Analysis

RealDevBench comprises 194 requirements spanning four practical domains: Analysis, Display, Data, and Game, that reflect core engineering needs. The distribution of tasks is as follows: Display (50.0%), Data (14.4%), Analysis (18.6%), and Game (17.0%), as illustrated in Figure 5. This allocation mirrors the prevalence of web-centric and data-intensive applications in real-world software development.

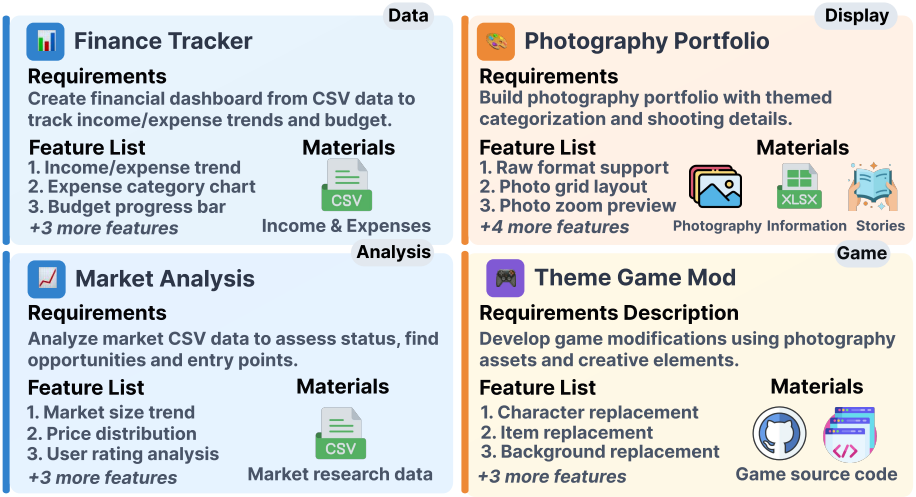


Figure 5: Representative cases from **RealDevBench** across four domains with consistent triplet structure (requirements, features, materials).

A.1 Display Domain Examples

The Display domain focuses primarily on web development and content presentation applications requiring front-end development expertise, involving interactive display functionalities such as personal blogs, portfolios, corporate websites, e-commerce storefronts, documentation sites, and multimedia galleries.

Display Task 1: Professional Portfolio Website

Requirement Description:

Create a **professional personal portfolio website** that showcases expertise and project experience. The system will process provided materials to generate a comprehensive, responsive web presence with privacy-conscious content filtering.

Feature List:

1. **Navigation System:** Fixed header with smooth scrolling navigation links
2. **Hero Section:** Professional profile photograph integration with dynamic introduction
3. **Project Showcase:** Interactive card-based layout with hover effects
4. **Skills Visualization:** Dynamic skill tag cloud with proficiency indicators
5. **Social Integration:** Elegant social media link collection with animations
6. **Resume Access:** Secure PDF download with privacy filtering
7. **Responsive Design:** Adaptive layout for all device types

Supplementary Materials: Resume document (PDF format), Professional profile photograph (JPG)

Display Task 2: Social Link Tree

Requirement 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

A.2 Analysis Domain Examples

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

Analysis Task 1: Blog Traffic Analysis

Requirement Description: Please design and implement the data analysis based on the following requirements: 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.

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

Requirement Description: Please design and implement the data analysis based on the following requirements. 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.

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

A.3 Data Domain Examples

The Data domain focuses on information processing and visualization systems:

Data Task 1: Finance Tracker

Requirement Description: Please design and implement the dashboard based on the following requirements: 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.

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

Requirement Description: Please design and implement the dashboard based on the following requirements: 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.

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

A.4 Game Domain Examples

The Game domain challenges test interactive entertainment application development:

Game Task 1: Mini Card Game

Requirement 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: TurboRally Game

Requirement Description: Turbo Rally is a racing game software that combines off-road driving with intense rally racing. Players can choose from a variety of rugged vehicles and compete in thrilling rally races on challenging off-road tracks. The objective is to navigate through rough terrain,dodge obstacles,and reach the finish line in the shortest time possible. The game features realistic physics,dynamic weather conditions,and stunning graphics to provide an immersive rally racing experience. Please design and implement it based on the following requirements:

Feature List:

1. Implement a vehicle selection interface displaying a minimum of 5 different off-road vehicles with distinct specifications (speed, handling, acceleration) and visual previews
2. Create a physics engine that simulates realistic vehicle behavior including suspension, terrain interaction, and collision detection with obstacles
3. Develop a dynamic weather system that affects vehicle handling and track conditions (rain reduces traction, mud affects speed, etc.)
4. Design a race tracking system that records lap times, checkpoint times, and maintains a leaderboard for each track
5. Create at least 3 distinct off-road tracks with varying terrain types (mud, gravel, sand) and obstacles (rocks, logs, water crossings)
6. Implement a real-time performance dashboard showing current speed, lap time, position, and track progress during races

Supplementary Materials: None

Appendix B AppEvalPilot Details

B.1 Hierarchical Action Space

The action space A of AppEvalPilot strikes a balance between expressiveness and operational efficiency, comprising four core actions that enable comprehensive automated testing capabilities, as shown in Table 4.

Specifically, the agent operates within a structured action space consisting of four core commands, serving as the foundational components for complex interactions. The action space includes:

- Open (app): Launches the target application via shortcut keys to enable quick context switching.
- Run (code): Uses PyAutoGUI to simulate mouse and keyboard input for complex interaction sequences.
- Tell (answer): Outputs test results to support validation and downstream metrics like *AgentScore*.
- **Stop**: Ends the current test episode, managing execution boundaries.

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

Table 4: **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.

B.2 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 at our case study website[6]. Analysis of these cases will illustrate AppEvalPilot's dynamic testing capabilities.

B.3 Prompt for Test Case Generation

[6]https://appevalpilot.realdev.world

Test Case Examples

1. **Navigation Verification:** Persistent top navigation bar positioning during scrolling
2. **Link Validation:** Intra-page navigation link accuracy ("Home", "Projects", etc.)
3. **Image Quality:** Avatar image rendering quality and aspect ratio preservation
4. **Content Integrity:** Biographical text completeness and typographic consistency
5. **Layout Testing:** Project card list formatting and content integrity
6. **Privacy Compliance:** Verify absence of compensation data in project disclosures
7. **Responsive Design:** Skill tag cloud layout responsiveness across devices
8. **Interactive Elements:** Test hover effects on skill tags and buttons
9. **External Links:** Social media link destination accuracy verification
10. **Download Function:** PDF resume download functionality testing
11. **File Integrity:** Validate 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.

`[Test Case Examples]`

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.

B.4 Prompt for Test Result Judgment

Test Judgement 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 "Uncertain":

Test Case Description: `[task_desc]`

Model Result: `[model_output]`

Only answer with "Yes", "No", or "Uncertain"

B.5 Prompt for Test Execution

Test Execution 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).

Test Execution Report Prompt

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.

B.6 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.

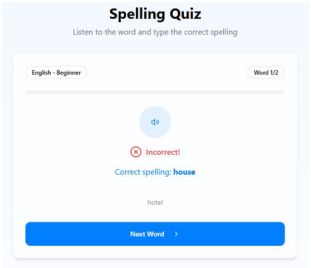
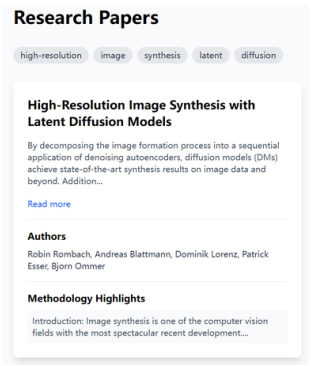
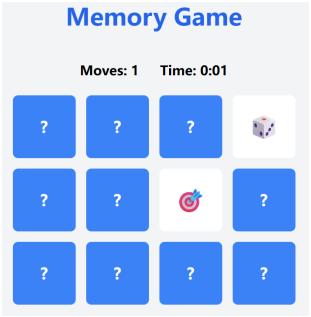
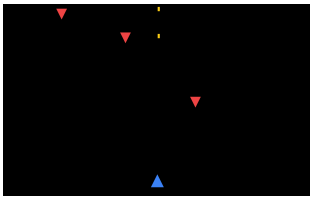
Project	Test Case	Failure Reason	Analysis	Screenshot
Language Spelling Bee	Verify that audio playback or definition display for quiz words functions correctly.	Missing Necessary Information	1.Lack of audio information makes the evidence insufficient. 2. The agent hallucinates a conclusion despite the 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.	
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.	

Table 5 – continued from previous page

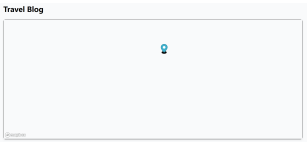
Project	Test Case	Failure Reason	Analysis	Screenshot
Travel Blog	Check if the webpage displays a map module.	Differences in Test Standards Understanding	The agent interpreted the test case literally, passing it as long as a map module was visible. However, the actual requirement implied that the map module must also be fully functional.	

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

Appendix C Evaluation of Software Quality

C.1 Manual Evaluation Process

In our evaluation process, we invited a total of 12 individuals, comprising 9 QA specialists (1-3 years experience) and 3 senior testing experts (5+ years experience), all of whom are professionals in the field of computer science and experienced software testing engineers. For each project evaluation, we assign 3 QA specialists to conduct independent assessments, with 1 senior expert overseeing the quality assurance process. This team conducted comprehensive assessments of each collected task and the generated software projects.

We first fix the generated software projects using Lovable [32] and establish reliable human ground truth labels through a rigorous two-level evaluation process: (1) *Test case-level*: For test cases c_i generated by AppEvalPilot, we invite 3 QA specialists (1-3 years experience) to execute each test case and evaluate Pass/Failed/Uncertain outcomes; (2) *Feature-level*: Each project also receives independent scoring from 3 QA specialists who manually test generated software projects against feature lists, providing granular scores for each feature $f_i \in \{0, 1\}$ (Failed/Pass), with final validation by a senior expert. Therefore, each project quality is recorded as human_quality $= \frac{1}{n}\sum_{i=1}^{n} f_i$ where n represents the total number of features.

Feature-level Human Annotation Process. In the feature-level annotation phase, human annotators are required to independently design verification procedures and test cases based on the provided feature list and the implemented application. For each feature f_i, annotators create approximately 3-5 test case groups that comprehensively cover different aspects and edge cases of the feature implementation. These test cases are designed to systematically validate whether each feature meets the specified requirements through practical execution scenarios. Based on the execution results of these custom-designed test cases, annotators provide feature implementation labels with three possible outcomes: *true* (feature correctly implemented), *false* (feature failed or incorrectly implemented), and *uncertain* (ambiguous or partially implemented feature requiring further evaluation).

Test Case Annotation Process. In the test case annotation phase, human annotators are tasked with executing the test cases generated by AppEvalPilot on the implemented applications. For each test case c_i, annotators manually perform the specified actions step-by-step on the live application interface. This includes clicking buttons, filling forms, navigating between pages, and triggering interactive elements as described in the test case instructions. During execution, annotators carefully document the complete execution trajectory, recording each action performed, the system's response, and any intermediate states encountered. For each test case, annotators provide comprehensive assessment results including: (1) the complete execution trace documenting each step performed and the corresponding system responses, (2) screenshots or screen recordings of key execution moments, (3) detailed descriptions of any deviations from expected behavior, and (4) a final evaluation label categorized as *true* (test case passed - application behavior matches expectations), *false* (test case failed with clear deviation from expected outcomes), or *uncertain* (ambiguous results requiring expert review, partial functionality or unclear expectations).

Quality Assurance and Validation. Throughout both annotation phases, all human annotators work independently to ensure unbiased evaluation results. To maintain annotation quality and consistency, the assigned senior expert performs comprehensive secondary review of all annotation results for their designated project. This validation process involves identifying cases or features with significant assessment discrepancies across the three independent evaluations, conducting trajectory review and re-execution verification for disputed results, and ensuring the reliability and trustworthiness of the final ground truth labels.

C.2 Code Quality

We adopt the Code Quality assessment methodology from [24], employing integrated Claude-3.5-Sonnet for automated scoring of source files. For supported file extensions including `.py`, `.html`, `.css`, `.js`, `.ts`, `.tsx`, and `.jsx`, we scan the corresponding code files and concatenate their content, then utilize the LLM to evaluate and score the overall code quality. The core prompt is shown as follows.

The evaluation process generates individual scores for each feature in the feature list. We apply a threshold of 75: features scoring above 75 are marked as passed (1), while those scoring 75 or below are marked as failed (0). The final LLM score represents the pass rate across all features: $\text{LLM\_score} = \frac{\text{number of passed features}}{\text{total number of features}}$.

Code Quality Prompt

```
To perform a comprehensive evaluation of the provided code, focus on a meticulous and
     step-by-step assessment using the established Software Evaluation Framework, aiming
     to yield minimal assessment scores based on rigorous real-world high standards.

# Software Evaluation Framework
## Evaluation Criteria
1. Implementation
    - Modularity: Code should be organized into logical, reusable components
    - Architecture: Clear separation of concerns and appropriate design patterns
    - Reusability: Components should be designed for potential reuse
2. Functionality
    - Core Features: All specified features must be fully implemented
    - Interactivity: Dynamic user interactions vs static implementations
    - User Experience: Intuitive and responsive interface
    - Error Handling: Comprehensive error management
    - State Management: Proper handling of application state
3. Logical Flow
    - Control Flow: Clear and efficient program execution paths
    - Data Flow: Proper data transformation and management
    - Event Handling: Appropriate response to system and user events
    - Asynchronous Operations: Proper handling of async processes
    - State Transitions: Clear and predictable state changes
4. Edge Cases
    - Input Validation: Handling of invalid or unexpected inputs
    - Boundary Conditions: Managing edge values and limits
    - Resource Management: Handling resource exhaustion scenarios
5. Requirement Dependencies
    - Feature Dependencies: Proper implementation of dependent features
    - External Services: Correct integration with external services
    - Database Schema: Proper database relationships and constraints

## Quality Metrics and Weightings
### Core Quality Dimensions (Total: 100 points)

1. Functional Correctness (25 points)
2. User Experience (25 points)
3. Maintainability (20 points)
4. Reliability \& Stability (20 points)
5. Security \& Data Protection (10 points)

# Query
{query}

# Requirements
{features}

# Code
{codes}

# Output Format
Output the evaluation results as a list of Boolean values and corresponding scores in
     JSON format.
'''
[
    {{
        "requirement_id": "Task Id",
        "satisfied": boolean, true or false, satisfies the requirement or not.
        "score": int, 0 ~ 100, the minimal evaluation score based on the high standards.
        "reason": "string, the detailed explanation of the evaluation in 3~5 sentences."
    }},
]
'''
## Examples
{example}

# Output
"""
```