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

Anonymous submission

Abstract

Large Language Models (LLMs) and code agents in software development are rapidly evolving from generating isolated code snippets to producing full-fledged software applications with graphical interfaces, interactive logic, and dynamic behaviors. However, current benchmarks fall short in evaluating such production-ready software, 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—qualities that only emerge when an application is actively used. This is the blind spot of current evaluation: *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 evaluation framework for automated end-to-end assessment of LLMs' ability to generate production-ready repositories from scratch. It features two key components: (1) **RealDevBench**, a diverse collection of 194 open-ended software engineering tasks across multiple domains, incorporating multimodal elements to reflect real-world complexity; and (2) **AppEvalPilot**, a new agent-as-a-judge evaluation system that simulates realistic, GUI-based user interactions to automatically and holistically assess software functional correctness, visual fidelity, and runtime behavior. The framework delivers fine-grained, task-specific diagnostic feedback, supporting nuanced evaluation beyond simple success/failure judgments. Empirical results show that RealDevWorld delivers effective, automatic, and human-aligned evaluations, achieving an accuracy of 0.92 and a correlation of 0.85 with expert human assessments, while significantly reducing the relianc on manual review. This enables scalable, human-aligned assessment of production-level software generated by LLMs.

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

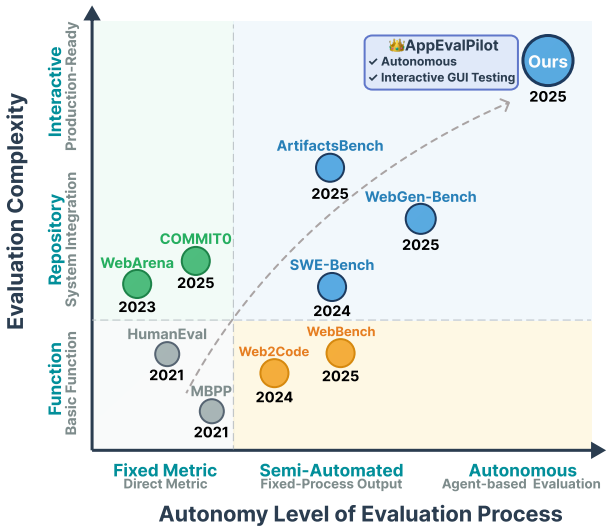

Figure 1: Software Engineering Evaluation: From Automated to Autonomous Evaluation

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 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 (Zhuo et al. 2024; Jain et al. 2025; Zhang et al. 2024) 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 (Ding et al. 2023; Li et al. 2025; Liu, Xu, and McAuley 2024; Zhang et al. 2023; Hu et al. 2025; Jimenez et al. 2024; Miserendino et al. 2025) 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 (Zhuge et al. 2024), 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 (Zhuge et al. 2024). Existing benchmarks such as BigCodeBench (Zhuo et al. 2024), LiveCodeBench (Jain et al. 2025), and NaturalCodeBench (Zhang et al. 2024) 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 (Hou et al. 2024; Jin et al. 2024). As a result, they fall short in assessing real-world development challenges such as integration, ambiguous specifications, and evolving requirements. Repository-level benchmarks (Ding et al. 2023; Li et al. 2025; Liu, Xu, and McAuley 2024; Zhang et al. 2023; Hu et al. 2025; Jimenez et al. 2024; Miserendino et al. 2025) tackle broader software tasks with interdependent components, but mainly use static metrics like similarity scores or unit tests (Fan et al. 2023; Laskar et al. 2024), which may not fully reflect functional correctness. Advanced benchmarks like rSDE-Bench (Hu et al. 2025), SWE-Bench (Jimenez et al. 2024), and SWE-Lancer (Miserendino et al. 2025) depend on pre-defined test cases, limiting their ability to evaluate adaptability to requirement changes or the creation of new modules. DEVAI (Zhuge et al. 2024) and MLE-Bench (Chan et al. 2024) 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 (Zheng et al. 2023b), 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.

Benchmark	Lang.	Level	Tasks	Eval Method	Agent Judge	Input Data	Interactive
BigCodeBench (Zhuo et al. 2024)	PY	Func.	Comp.	Unit test	✗	Text, Code	✗
LiveCodeBench (Jain et al. 2025)	PY	Func.	Gen.	Unit test	✗	Text, Code	✗
RepoBench (Liu, Xu, and McAuley 2024)	PY, Java	Repo.	Ret.	Similarity	✗	Text, Code	✗
SWE-Bench (Jimenez et al. 2024)	PY	Repo.	Maint.	Unit test	✗	Multi-modal	✗
EvoCodeBench (Li et al. 2025)	PY	Repo.	Ret.	Pass@k	✗	Text, Code	✗
SWE-Lancer (Miserendino et al. 2025)	JS, TS	Repo.	Dev.	Unit test	✗	Multi-modal	✗
FrontendBench (Zhu et al. 2025)	JS	Repo.	Gen.	Unit test	✗	Text	✓
COMMIT0 (Zhao et al. 2024)	PY	Repo.	Dev.	Unit test	✗	Multi-modal	✗
Web-Bench (Xu et al. 2025)	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.*

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 Preliminary

This section formalizes the task of end-to-end software evaluation and analyzes three mainstream evaluation paradigms—human evaluation, LLM-as-a-Judge (Zheng et al. 2023a), and Agent-as-a-Judge (Zhuge et al. 2024)-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 (iso 2022), SVCOMP (Beyer 2024)), 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 Q_{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 $\text{Trace}_{\text{static}}$.

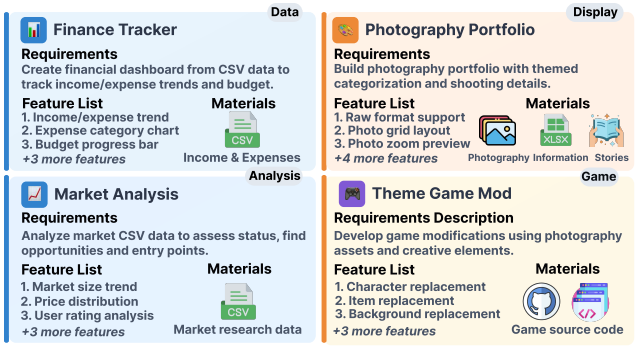

Figure 2: Representative cases from **RealDevBench** across four domains - Data, Display, Analysis, and Game - with consistent triplet structure (requirements, features, materials), reflecting real-world software engineering challenges.

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

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

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%). 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) 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 (Vichare et al. 2025) to establish 4 domain categories: **Display**, **Analysis Data**, and **Game**. We sampled requirements from SRDD (OpenBMB 2024) and expanded through web crawling freelancer platforms (Upwork[1] and Freelancer[2]) 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 (Anthropic 2024) to extract functional requirements from repository documentation and expand requirements into structured feature specifications, ensuring consistent translation of requirements into actionable development features with clear evaluation criteria.

Task Structure and Formulation. As illustrated in Figure 2, each task in **RealDevBench** is structured as a triplet to simulate realistic software development scenarios: (1) Requirements Description: A brief 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 [3] for thematic relevance and professional quality; (2) Datasets: Selected from Kaggle [4] based on topic relevance and appropriate complexity; (3) Documents: Manually created documents (resumes, business proposals, catalogs) that mirror real-world usage 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 3, 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 behaviors. This dynamic and

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

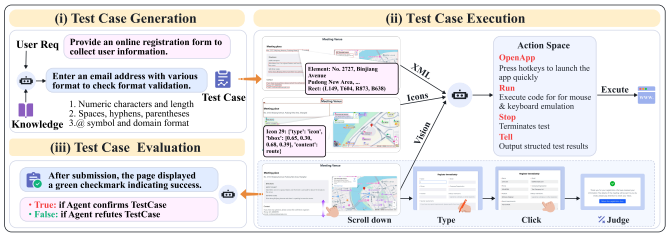

Figure 3: Overall design of AppEvalPilot showing the automated testing workflow: test case generation from user requirements, multimodal test execution through interface interaction (scrolling, typing, clicking), and binary evaluation of outcomes for objective software assessment.

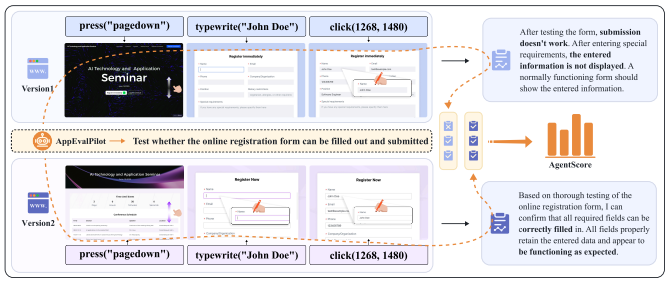

Figure 4: Evaluation pipeline of AppEvalPilot. The agent performs test sequences on two different web implementations, systematically assesses functionality through direct interaction, documents observable differences in form behavior, and generates quantitative scores based on test cases.

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 (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. 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 graphical user interfaces (GUIs), effectively simulating genuine user interactions. As shown in Figure 3, 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. 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.

These atomic actions, as shown in Figure 3, 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 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 4, 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 engineerinhg?

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 (Team 2024a) 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 (Anthropic 2024), UI-Tars (Qin et al. 2025), WebVoyager-Agent (He et al. 2024) with qwen2.5-vl-32B (Bai et al. 2025) and claude-3.5-sonnet-v2 backbones, and Browser-Use with claude-3.7-sonnet-v2 (Anthropic 2025). 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 (Müller and Žunič 2024), 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.

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 5: (1) Code Quality assessment (Zheng et al. 2023a) 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 (Lu et al. 2025). As demonstrated in Figure 5, 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

Figure 5: 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).

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.

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 (Team 2024b), MGX (BoN-3), Bolt (StackBlitz 2024), Lovable, OpenHands (Wang et al. 2025), Claude-3.5-Sonnet, Gemini-2.5-Pro (Comanici et al. 2025), Kimi-K2 (Team et al. 2025), DeepSeek-V3 (DeepSeek-AI 2024), Qwen3-Coder-480B (Team 2025), 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.

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 evaluation results across different code generation systems and evaluation methods.

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

2022. Software and systems engineering — Software testing — Part 1: General concepts.

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

Anthropic. 2025. Claude 3.7 Sonnet. https://www.anthropic.com/claude/sonnet.

Bai, S.; Chen, K.; Liu, X.; Wang, J.; Ge, W.; Song, S.; Dang, K.; Wang, P.; Wang, S.; Tang, J.; et al. 2025. Qwen2. 5-vl technical report. *arXiv preprint arXiv:2502.13923*.

Beyer, D. 2024. 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.

Chan, J. S.; Chowdhury, N.; Jaffe, O.; Aung, J.; Sherburn, D.; Mays, E.; Starace, G.; Liu, K.; Maksin, L.; Patwardhan, T.; et al. 2024. Mle-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*.

Cheng, K.; Sun, Q.; Chu, Y.; Xu, F.; Li, Y.; Zhang, J.; and Wu, Z. 2024. Seeclick: Harnessing gui grounding for advanced visual gui agents. *arXiv preprint arXiv:2401.10935*.

Comanici, G.; Bieber, E.; Schaekermann, M.; Pasupat, I.; Sachdeva, N.; Dhillon, I.; Blistein, M.; Ram, O.; Zhang, D.; Rosen, E.; et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437.

Ding, Y.; Wang, Z.; Ahmad, W.; Ding, H.; Tan, M.; Jain, N.; Ramanathan, M. K.; Nallapati, R.; Bhatia, P.; Roth, D.; et al. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36: 46701–46723.

Fan, A.; Gokkaya, B.; Harman, M.; Lyubarskiy, M.; Sengupta, S.; Yoo, S.; and Zhang, J. M. 2023. 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)*, 31–53. IEEE.

He, H.; Yao, W.; Ma, K.; Yu, W.; Dai, Y.; Zhang, H.; Lan, Z.; and Yu, D. 2024. Webvoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919*.

Hou, X.; Zhao, Y.; Liu, Y.; Yang, Z.; Wang, K.; Li, L.; Luo, X.; Lo, D.; Grundy, J.; and Wang, H. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8): 1–79.

Hu, Y.; Cai, Y.; Du, Y.; Zhu, X.; Liu, X.; Yu, Z.; Hou, Y.; Tang, S.; and Chen, S. 2025. Self-Evolving Multi-Agent Networks for Software Development. In *The Thirteenth International Conference on Learning Representations*.

Jain, N.; Han, K.; Gu, A.; Li, W.-D.; Yan, F.; Zhang, T.; Wang, S.; Solar-Lezama, A.; Sen, K.; and Stoica, I. 2025. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. In *The Thirteenth International Conference on Learning Representations*.

Jimenez, C. E.; Yang, J.; Wettig, A.; Yao, S.; Pei, K.; Press, O.; and Narasimhan, K. R. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues? In *The Twelfth International Conference on Learning Representations*.

Jin, H.; Huang, L.; Cai, H.; Yan, J.; Li, B.; and Chen, H. 2024. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*.

Laskar, M. T. R.; Alqahtani, S.; Bari, M. S.; Rahman, M.; Khan, M. A. M.; Khan, H.; Jahan, I.; Bhuiyan, A.; Tan, C. W.; Parvez, M. R.; et al. 2024. 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*, 13785–13816.

Li, J.; Li, G.; Zhang, X.; Zhao, Y.; Dong, Y.; Jin, Z.; Li, B.; Huang, F.; and Li, Y. 2025. Evocodebench: An evolving code generation benchmark with domain-specific evaluations. *Advances in Neural Information Processing Systems*, 37: 57619–57641.

Liu, T.; Xu, C.; and McAuley, J. 2024. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. In *The Twelfth International Conference on Learning Representations*.

Lu, Z.; Yang, Y.; Ren, H.; Hou, H.; Xiao, H.; Wang, K.; Shi, W.; Zhou, A.; Zhan, M.; and Li, H. 2025. WebGen-Bench: Evaluating LLMs on Generating Interactive and Functional Websites from Scratch. arXiv:2505.03733.

Miserendino, S.; Wang, M.; Patwardhan, T.; and Heidecke, J. 2025. SWE-Lancer: Can Frontier LLMs Earn $1 Million from Real-World Freelance Software Engineering? *arXiv preprint arXiv:2502.12115*.

Müller, M.; and Žunič, G. 2024. Browser Use: Enable AI to control your browser.

OpenBMB. 2024. SRDD. https://github.com/OpenBMB/ChatDev/tree/main/SRDD. Accessed: 2025-03-29.

Qin, Y.; Ye, Y.; Fang, J.; et al. 2025. UI-TARS: Pioneering Automated GUI Interaction with Native Agents. *arXiv preprint arXiv:2501.12326*.

StackBlitz. 2024. Bolt: AI-Powered Development Platform. https://bolt.new.

Team, K.; Bai, Y.; Bao, Y.; Chen, G.; Chen, J.; Chen, N.; Chen, R.; Chen, Y.; Chen, Y.; Chen, Y.; Chen, Z.; Cui, J.; Ding, H.; Dong, M.; Du, A.; Du, C.; Du, D.; Du, Y.; Fan, Y.; Feng, Y.; Fu, K.; Gao, B.; Gao, H.; Gao, P.; Gao, T.; Gu, X.; Guan, L.; Guo, H.; Guo, J.; Hu, H.; Hao, X.; He, T.; He, W.; He, W.; Hong, C.; Hu, Y.; Hu, Z.; Huang, W.; Huang, Z.; Huang, Z.; Jiang, T.; Jiang, Z.; Jin, X.; Kang, Y.; Lai, G.; Li, C.; Li, F.; Li, H.; Li, M.; Li, W.; Li, Y.; Li, Y.; Li, Z.; Li, Z.; Lin, H.; Lin, X.; Lin, Z.; Liu, C.; Liu, C.; Liu,

H.; Liu, J.; Liu, J.; Liu, L.; Liu, S.; Liu, T. Y.; Liu, T.; Liu, W.; Liu, Y.; Liu, Y.; Liu, Y.; Liu, Y.; Liu, Z.; Lu, E.; Lu, L.; Ma, S.; Ma, X.; Ma, Y.; Mao, S.; Mei, J.; Men, X.; Miao, Y.; Pan, S.; Peng, Y.; Qin, R.; Qu, B.; Shang, Z.; Shi, L.; Shi, S.; Song, F.; Su, J.; Su, Z.; Sun, X.; Sung, F.; Tang, H.; Tao, J.; Teng, Q.; Wang, C.; Wang, D.; Wang, F.; Wang, H.; Wang, J.; Wang, J.; Wang, J.; Wang, S.; Wang, S.; Wang, Y.; Wang, Y.; Wang, Y.; Wang, Y.; Wang, Y.; Wang, Z.; Wang, Z.; Wang, Z.; Wei, C.; Wei, Q.; Wu, W.; Wu, X.; Wu, Y.; Xiao, C.; Xie, X.; Xiong, W.; Xu, B.; Xu, J.; Xu, J.; Xu, L. H.; Xu, L.; Xu, S.; Xu, W.; Xu, X.; Xu, Y.; Xu, Z.; Yan, J.; Yan, Y.; Yang, X.; Yang, Y.; Yang, Z.; Yang, Z.; Yang, Z.; Yao, H.; Yao, X.; Ye, W.; Ye, Z.; Yin, B.; Yu, L.; Yuan, E.; Yuan, H.; Yuan, M.; Zhan, H.; Zhang, D.; Zhang, H.; Zhang, W.; Zhang, X.; Zhang, Y.; Zhang, Y.; Zhang, Y.; Zhang, Y.; Zhang, Y.; Zhang, Z.; Zhao, H.; Zhao, Y.; Zheng, H.; Zheng, S.; Zhou, J.; Zhou, X.; Zhou, Z.; Zhu, Z.; Zhuang, W.; and Zu, X. 2025. Kimi K2: Open Agentic Intelligence. arXiv:2507.20534.

Team, L. 2024a. Lovable: AI Development Solution. https://lovable.dev.

Team, M. 2024b. MGX: AI Software Development Platform. https://mgx.dev.

Team, Q. 2025. Qwen3 Technical Report. arXiv:2505.09388.

Vichare, A.; Angelopoulos, A. N.; Chiang, W.-L.; Tang, K.; and Manolache, L. 2025. WebDev Arena: A Live LLM Leaderboard for Web App Development.

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

Wang, X.; Li, B.; Song, Y.; Xu, F. F.; Tang, X.; Zhuge, M.; Pan, J.; Song, Y.; Li, B.; Singh, J.; Tran, H. H.; Li, F.; Ma, R.; Zheng, M.; Qian, B.; Shao, Y.; Muennighoff, N.; Zhang, Y.; Hui, B.; Lin, J.; Brennan, R.; Peng, H.; Ji, H.; and Neubig, G. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *The Thirteenth International Conference on Learning Representations*.

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

Xu, J.; Guo, K.; Gong, W.; and Shi, R. 2024. OSAgent: Copiloting Operating System with LLM-based Agent. In *2024 International Joint Conference on Neural Networks (IJCNN)*, 1–9. IEEE.

Xu, K.; Mao, Y.; Guan, X.; and Feng, Z. 2025. Web-bench: A llm code benchmark based on web standards and frameworks. *arXiv preprint arXiv:2505.07473*.

Zhang, F.; Chen, B.; Zhang, Y.; Keung, J.; Liu, J.; Zan, D.; Mao, Y.; Lou, J.-G.; and Chen, W. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2471–2484.

Zhang, S.; Zhao, H.; Liu, X.; Zheng, Q.; Qi, Z.; Gu, X.; Dong, Y.; and Tang, J. 2024. Naturalcodebench: Examining coding performance mismatch on humaneval and natural user queries. In *Findings of the Association for Computational Linguistics ACL 2024*, 7907–7928.

Zhao, W.; Jiang, N.; Lee, C.; Chiu, J. T.; Cardie, C.; Gallé, M.; and Rush, A. M. 2024. Commit0: Library generation from scratch. *arXiv preprint arXiv:2412.01769*.

Zheng, L.; Chiang, W.-L.; Sheng, Y.; Zhuang, S.; Wu, Z.; Zhuang, Y.; Lin, Z.; Li, Z.; Li, D.; Xing, E.; Zhang, H.; Gonzalez, J. E.; and Stoica, I. 2023a. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In Oh, A.; Naumann, T.; Globerson, A.; Saenko, K.; Hardt, M.; and Levine, S., eds., *Advances in Neural Information Processing Systems*, volume 36, 46595–46623. Curran Associates, Inc.

Zheng, L.; Chiang, W.-L.; Sheng, Y.; Zhuang, S.; Wu, Z.; Zhuang, Y.; Lin, Z.; Li, Z.; Li, D.; Xing, E.; et al. 2023b. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36: 46595–46623.

Zhu, H.; Zhang, Y.; Zhao, B.; Ding, J.; Liu, S.; Liu, T.; Wang, D.; Liu, Y.; and Li, Z. 2025. Frontend-Bench: A Benchmark for Evaluating LLMs on Front-End Development via Automatic Evaluation. *arXiv preprint arXiv:2506.13832*.

Zhuge, M.; Zhao, C.; Ashley, D.; Wang, W.; Khizbullin, D.; Xiong, Y.; Liu, Z.; Chang, E.; Krishnamoorthi, R.; Tian, Y.; et al. 2024. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934*.

Zhuo, T. Y.; Vu, M. C.; Chim, J.; Hu, H.; Yu, W.; Widyasari, R.; Yusuf, I. N. B.; Zhan, H.; He, J.; Paul, I.; et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

Reproducibility Checklist

Instructions for Authors:
This document outlines key aspects for assessing reproducibility. Please provide your input by editing this `.tex` file directly.

For each question (that applies), replace the "Type your response here" text with your answer.

Example: If a question appears as

```
\question{Proofs of all novel claims
are included} {(yes/partial/no)}
Type your response here
```

you would change it to:

```
\question{Proofs of all novel claims
are included} {(yes/partial/no)}
yes
```

Please make sure to:

- Replace ONLY the "Type your response here" text and nothing else.
- Use one of the options listed for that question (e.g., **yes**, **no**, **partial**, or **NA**).
- **Not** modify any other part of the `\question` command or any other lines in this document.

You can `\input` this .tex file right before `\end{document}` of your main file or compile it as a stand-alone document. Check the instructions on your conference's website to see if you will be asked to provide this checklist with your paper or separately.

1. General Paper Structure

1.1. Includes a conceptual outline and/or pseudocode description of AI methods introduced (yes/partial/no/NA) yes

1.2. Clearly delineates statements that are opinions, hypothesis, and speculation from objective facts and results (yes/no) yes

1.3. Provides well-marked pedagogical references for less-familiar readers to gain background necessary to replicate the paper (yes/no) yes

2. Theoretical Contributions

2.1. Does this paper make theoretical contributions? (yes/no) yes

If yes, please address the following points:

2.2. All assumptions and restrictions are stated clearly and formally (yes/partial/no) yes

2.3. All novel claims are stated formally (e.g., in theorem statements) (yes/partial/no) yes

2.4. Proofs of all novel claims are included (yes/partial/no) yes

2.5. Proof sketches or intuitions are given for complex and/or novel results (yes/partial/no) yes

2.6. Appropriate citations to theoretical tools used are given (yes/partial/no) yes

2.7. All theoretical claims are demonstrated empirically to hold (yes/partial/no/NA) yes

2.8. All experimental code used to eliminate or disprove claims is included (yes/no/NA) yes

3. Dataset Usage

3.1. Does this paper rely on one or more datasets? (yes/no) yes

If yes, please address the following points:

3.2. A motivation is given for why the experiments are conducted on the selected datasets (yes/partial/no/NA) yes

3.3. All novel datasets introduced in this paper are included in a data appendix (yes/partial/no/NA) yes

3.4. All novel datasets introduced in this paper will be made publicly available upon publication of the paper with a license that allows free usage for research purposes (yes/partial/no/NA) yes

3.5. All datasets drawn from the existing literature (potentially including authors' own previously published work) are accompanied by appropriate citations (yes/no/NA) yes

3.6. All datasets drawn from the existing literature (potentially including authors' own previously published work) are publicly available (yes/partial/no/NA) yes

3.7. All datasets that are not publicly available are described in detail, with explanation why publicly available alternatives are not scientifically satisficing (yes/partial/no/NA) yes

4. Computational Experiments

4.1. Does this paper include computational experiments? (yes/no) yes

If yes, please address the following points:

4.2. This paper states the number and range of values tried per (hyper-) parameter during development of the paper, along with the criterion used for selecting the final parameter setting (yes/partial/no/NA) yes

4.3. Any code required for pre-processing data is included in the appendix (yes/partial/no) yes

4.4. All source code required for conducting and analyzing the experiments is included in a code appendix (yes/partial/no) yes

4.5. All source code required for conducting and analyzing the experiments will be made publicly available upon publication of the paper with a license that allows free usage for research purposes (yes/partial/no) yes

4.6. All source code implementing new methods have comments detailing the implementation, with references to the paper where each step comes from (yes/partial/no) yes

4.7. If an algorithm depends on randomness, then the method used for setting seeds is described in a way sufficient to allow replication of results (yes/partial/no/NA) yes

4.8. This paper specifies the computing infrastructure used for running experiments (hardware and software), including GPU/CPU models; amount of memory; operating system; names and versions of relevant software libraries and frameworks (yes/partial/no) yes

4.9. This paper formally describes evaluation metrics used and explains the motivation for choosing these metrics (yes/partial/no) yes

4.10. This paper states the number of algorithm runs used to compute each reported result (yes/no) yes

4.11. Analysis of experiments goes beyond single-dimensional summaries of performance (e.g., average; median) to include measures of variation, confidence, or other distributional information (yes/no) yes

4.12. The significance of any improvement or decrease in performance is judged using appropriate statistical tests (e.g., Wilcoxon signed-rank) (yes/partial/no) yes

4.13. This paper lists all final (hyper-)parameters used for each model/algorithm in the paper's experiments (yes/partial/no/NA) yes