PlayCoder: Making LLM-Generated GUI Code Playable

ZHIYUAN PENG*, Shanghai Jiao Tong University, China
WEI TAO*, LIGHTSPEED, China
XIN YIN, Zhejiang University, China
CHENHAO YING, Shanghai Jiao Tong University, China
YUAN LUO†, Shanghai Jiao Tong University, China
YIWEN GUO†, Independent Researcher, China

Large language models (LLMs) have transformed code generation, but their ability to generate code for applications with graphical user interfaces (GUIs), particularly games, remains underexplored. Prior code-generation benchmarks assess correctness using test cases, but this is insufficient for GUI applications. These applications are interactive and event-driven, and their correctness depends on stateful behavior over sequences of user actions. Consequently, evaluation should account for interaction flows and UI state transitions rather than relying solely on pass or fail test outcomes. To explore the performance of LLMs on GUI applications, we construct PlayEval, a repository-aware evaluation dataset from 43 multilingual (Python, TypeScript, and JavaScript) GUI applications. Different from existing GUI benchmarks which are difficult to transplant to Desktop platform, PlayEval consists of 6 major categories of GUI applications and directly facilitates evaluation on code generation tasks. To enable more reliable assessment beyond simple execution and unit tests, we propose Play@k, which measures whether at least one of k generated candidates yields an application that can be played end-to-end without logical errors. We further develop an LLM-based agent, PlayTester, that automates interactive evaluation by driving the GUI through task-oriented playthroughs and checking for logic violations. Through systematic evaluation, we demonstrate that 10 state-of-the-art code LLMs struggle to generate logically correct GUI applications, achieving near-zero Play@3 scores despite high compilation rates. To address these, we introduce PlayCoder, a multi-agent, repository-aware framework that writes, evaluates and refines GUI application code via closed-loop control. PlayCoder substantially improves functional correctness and semantic alignment for both open-source and closed-source models, achieving up to 38.1% Exec@3 and 20.3% Play@3. Case studies show that it detects silent logic flaws missed by traditional metrics and repairs them through targeted edits. These results indicate that coupling an end-to-end GUI testing agent with repository-aware automated program repair is an effective path towards reliable GUI code generation. Our implementation is publicly available at https://github.com/Tencent/PlayCoder.

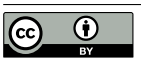
CCS Concepts: • **Software and its engineering**;

Additional Key Words and Phrases: Large Language Model, Code Generation, Multi-Agent, GUI Applications

ACM Reference Format:
Zhiyuan Peng, Wei Tao, Xin Yin, Chenhao Ying, Yuan Luo, and Yiwen Guo. 2026. PlayCoder: Making LLM-Generated GUI Code Playable. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE090 (July 2026), 24 pages. https://doi.org/10.1145/3808097

*Zhiyuan Peng and Wei Tao contributed equally to this research.
†Yuan Luo and Yiwen Guo are the corresponding authors.

Authors' Contact Information: Zhiyuan Peng, pzy2000@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Wei Tao, wtao@ieee.org, LIGHTSPEED, Shenzhen, China; Xin Yin, xyin@zju.edu.cn, Zhejiang University, Hangzhou, China; Chenhao Ying, yingchenhao@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Yuan Luo, yuanluo@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Yiwen Guo, guoyiwen89@gmail.com, Independent Researcher, China.



ACM 2994-970X/2026/7-ARTFSE090
https://doi.org/10.1145/3808097

1 Introduction

Large language models (LLMs) have revolutionized software engineering tasks (e.g., code generation and bug fixing), achieving impressive results on established benchmarks like HumanEval [10], CoderEval [76], and SWE-Bench [26]. These established benchmarks primarily target well-specified, self-contained programming tasks amenable to unit test verification. Consequently, they inadequately represent the complexities of open-ended environments that require capabilities beyond single-shot function synthesis (e.g., multi-step interaction with live systems, sustained stateful execution, and external tool integration). Such capabilities are fundamental to interactive graphical user interface (GUI) applications. GUI code generation presents distinct evaluation challenges: models must handle event-driven control flow, persistent and evolving application state, and complex user interaction patterns. While algorithmic tasks permit assessment through input-output validation, GUI systems necessitate interactive verification procedures that existing evaluation paradigms cannot accommodate. This evaluation limitation becomes particularly acute because GUI applications frequently manifest silent behavioral failures, where syntactically correct and executable code violates fundamental application logic.

Current evaluation frameworks inadequately assess the behavioral requirements of GUI applications. While SWE-Bench [26] advances repository-aware evaluation, it relies primarily on unit tests and static analysis. These approaches prove insufficient for GUI applications, where behavioral correctness cannot be captured through traditional test cases [7]. While web-based GUI testing frameworks (e.g., Selenium, Playwright) automate interaction through DOM manipulation, they fundamentally rely on accessible structural representations that many GUI applications lack. Canvas-based applications, desktop games, and native GUI programs render content directly to pixels without exposing DOM trees or accessibility APIs, making structure-based testing infeasible. GUI applications may pass unit tests yet exhibit interactive failures that manifest only during runtime execution. As shown in Fig. 1, consider a Flappy Bird game that compiles without errors but allows the bird to pass through obstacles, violating core game mechanics while producing no exceptions. Such failures remain undetectable through unit tests because obstacles are randomly generated with non-deterministic coordinates, making comprehensive test case coverage impractical. Consequently, developers typically rely on human testers to identify and report behavioral bugs, a process that is both time-consuming and costly. This gap demonstrates that current approaches fundamentally struggle with interactive GUI applications.

To address these, we establish evaluation benchmark, metrics, and method for GUI-based code generation. We develop a GUI-testing methodology that capture behavioral correctness through automated user interaction simulation. These methodologies are complemented by PlayEval, a curated benchmark of 43 diverse multilingual (Python, TypeScript, and JavaScript) GUI applications spanning six major categories (e.g., classic games, MMORPG games, productivity tools) with verifiable GUI behaviors. We focus on interactive GUI applications. Games are selected as examples to represent the challenge of interactivity: requiring frequent state updates, event handling, and operating solely based on visual feedback (unlike web-GUIs with accessible DOM structures). The benchmark includes general-purpose desktop-applications because it can reflect common use cases and evaluate the generality of such less complex, yet challenging applications. We propose Play@k, a behavioral correctness metric that measures whether generated code can be interactively executed end-to-end without logical errors. Since Play@k requires code first to pass unit tests before GUI validation, it provides a more stringent assessment than traditional Pass@k metrics.

Building upon these evaluation foundations, we propose PlayCoder, a multi-agent framework that leverages our evaluation methods for robust GUI-based code generation. The framework employs two specialized agents: a repository-aware coding agent (PlayDeveloper) for initial code generation and an automated program repair agent (PlayRefiner) that iteratively refines code based on evaluation feedback from PlayTester. PlayTester serves as the behavioral testing framework that verifies correctness across programming languages and platforms (Windows, macOS, and X11-based Linux distributions). As shown in Figure 4, PlayDeveloper generates initial code, then PlayTester detects behavioral deviations, enabling PlayRefiner to autonomously debug and modify code through successive test-repair cycles. This iterative loop helps produce code that is syntactically valid and better aligned with the requirements. Iterative refinement [28, 45, 59, 70] is prevalent. However, PlayCoder differs in *what-drives-the-loop*: 1) visual vs. textual feedback: Prior works rely on **textual signals**. PlayCoder closes the loop using **visual feedback** (i.e., screenshots) and **dynamic interaction** (e.g., mouse or keyboard operation), enabling it to fix "silent failures" (e.g., invisible text, unresponsive buttons) that text-based oracles miss. 2) active exploration vs. passive testing: Standard loops use pre-defined test suite. Employing PlayTester dynamically explores the UI to discover bugs. Our evaluation demonstrates PlayCoder's effectiveness across multilingual GUI applications. Using GPT-5-mini, our framework achieves 26.8% Exec@3 and 9.8% Play@3, compared to the best baseline (DeepCode) with 17.9% Exec@3 and 6.4% Play@3. With Claude-Sonnet-4, PlayCoder reaches 36.8% Exec@3 and 20.3% Play@3, demonstrating model-agnostic benefits and establishing a new paradigm for interactive GUI application code generation. Our main contributions are summarized as follows:

- We present a comprehensive benchmark for GUI application code generation, consisting of the PlayEval dataset and the Play@k metric. Covering 43 multilingual applications across 6 domains (e.g., MMORPGs), our experiments uncover a severe performance gap in behavioral correctness. With the top-performing model achieving only 9.9% Play@3 (18.6% Exec@3) and the weakest baseline scoring <1%, our findings highlight critical challenges that current methods fail to solve.
- We propose PlayTester, a GUI Testing Agent that serves dual purposes: As an evaluator, PlayTester detects subtle behavioral failures (e.g., collision detection errors, event handling inconsistencies) that traditional unit tests overlook. As a feedback provider, PlayTester provides precise behavioral diagnostics to guide iterative code improvement, addressing repository hallucination where models generate syntactically correct but behaviorally incorrect code.
- We propose PlayCoder, a novel multi-agent framework that integrates repository-aware code generation, automated GUI behavioral testing, and iterative program repair. The framework employs two specialized agents (PlayDeveloper and PlayRefiner) that collaborate through structured test&repair cycles, using visual feedback from the behavioral evaluation framework. PlayCoder achieves up to 20.2% higher Exec@3 and 11.0% higher Play@3 compared to baselines, with consistent effectiveness across diverse LLM architectures and superior cost-effectiveness.

2 Motivation

2.1 A Motivating Example

Fig. 1 illustrates an example in *Flappy Bird* generated by GPT-4o-mini and a human programmer. The program compiles and runs, but it allows the bird to pass through obstacles, so the game never ends (as shown in the bottom-right of Fig. 1). In the correct behavior, a collision should kill the bird and terminate the game (as shown in the top-right of Fig. 1). Such failures do not raise exceptions or cause crashes, allowing them to slip past evaluations that only verify compilation or test cases.

Challenge 1: Testing Dilemma for GUI Application Code Generation. Traditional evaluation of code emphasizes compilation success and unit test, which are inadequate for GUI applications. Unlike function-level code validated by input–output pairs, GUI programs demand interactive, stateful, and temporal validation that current paradigms miss [7]. In the *Flappy Bird* example, the code runs without runtime errors, yet critical behavioral flaws (e.g., the bird penetrating pipes) remain undetected. Metrics (e.g., Pass@k) cannot distinguish a GUI application from one with broken logic, calling for behavioral testing via interactive execution.

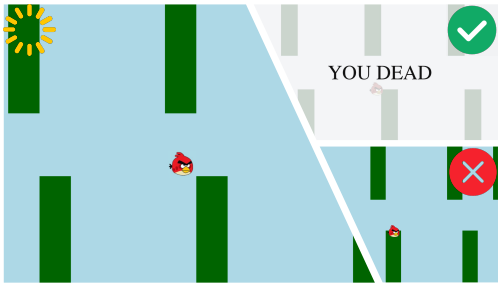


Fig. 1. *Flappy Bird* generated by GPT-4o-mini and human programmer. Top-right: code written by a human programmer, where collision correctly kills the bird and ends the game. Bottom-right: code generated by GPT-4o-mini, where the bird can pass through the pipe, which is a critical logic flaw.

Challenge 2: Insufficient Benchmarks for GUI Application Code Generation. Existing code generation benchmarks systematically underrepresent GUI application generation (including GUI-based games), despite its prevalence in practice. HumanEval [10] and CoderEval [76] focus on algorithmic or function-level tasks; while SWE-Bench [26] advances repository-aware evaluation, it fundamentally relies on unit test passage and static analysis, making it insufficient for GUI applications where behavioral correctness requires interactive validation that traditional test cases cannot capture. Preliminary experiment from PlayEval (i.e., Table 2) shows a sharp degradation from executability to behavioral validity: the best model (Claude-Sonnet-4) achieves 18.6% execution correctness but only 9.9% behavior correctness on Python; GPT-5 drops from 17.5% to 6.9% (refer to Section 3.6 for more details) on Python. This gap yields a false sense of competence: models that score highly on traditional benchmarks perform poorly on high-complexity GUI tasks involving event handling, state updates, and physics-based animation. **Challenge 3: Difficulty in Repository-Aware GUI Code Generation.** While recent repository-aware code generation methods (e.g., MetaGPT [22], DeepCode [29]) have made progress in incorporating repository context, they still face significant challenges when applied to GUI application code generation. These methods excel at retrieving relevant code snippets and API documentation, yet they struggle with GUI-specific behavioral correctness because traditional repository analysis focuses on syntactic patterns rather than interactive semantics. Observable behavioral failures, such as the bird in *Flappy Bird* passing through pipes without collision detection, often occur despite syntactically correct API usage and proper imports. GUI applications require understanding of event loops, state transitions, and temporal properties that cannot be captured through static repository analysis alone. Even with comprehensive repository grounding, existing agents fail to distinguish between code that compiles correctly and code that behaves correctly during interactive execution, leading to systematic behavioral hallucinations in GUI contexts. In addition to this, existing tools (e.g., Selenium) and recent LLM-based testers (e.g., GPTDroid [41], LLMDroid [61], VETL [62]) focus on **Web or Mobile** platforms. They rely on structured DOMs or accessibility trees that are missing or non-standard in desktop Python GUIs (especially PyGame, which renders to a canvas). Transplanting them to support desktop GUI applications is impractical.

2.2 Key Ideas

Based on the above challenges, we present three complementary ideas to enable and evaluate repository-aware GUI application code generation with behavioral guarantees.

Key Idea 1: Overlooked GUI Generation Challenges. Our preliminary experiments on PlayEval reveal that GUI application code generation poses significantly greater challenges than previously recognized, with systematic failures overlooked by existing evaluation paradigms. Pass@k, which relies solely on unit tests, fails to capture GUI-specific errors such as incorrect collision detection, broken event handling, or randomly generated map inconsistencies that appear functionally correct but render applications unusable. Current benchmarks inadequately represent GUI application complexity, while existing methods lack proper assessment frameworks for interactive behavioral correctness. Preliminary studies in Section 3.6 demonstrate substantial performance degradation across state-of-the-art models: Claude-Sonnet-4 drops from 18.6% Exec@3 to 9.9% Play@3, while GPT-5 plummets from 17.5% Exec@3 to merely 6.9% Play@3, revealing a critical gap between syntactic correctness and behavioral validity that traditional metrics systematically miss.

Key Idea 2: Novel GUI Benchmark with Hierarchical Behavioral Testing. We contribute PlayEval, a comprehensive repository-aware benchmark that advances GUI application code generation evaluation through three key innovations: (1) a curated dataset of diverse GUI applications (e.g., GUI-based games, productivity tools) with repository scaffolds and behavioral specifications; (2) a novel evaluation methodology that introduces hierarchical testing with our PlayTester; and (3) a rigorous metric framework that detects subtle behavioral failures overlooked by traditional approaches. Our Play@k metric represents a significant methodological advancement: it exclusively evaluates code that first passes unit tests (Pass@k), then subjects it to automated GUI behavioral testing via our specialized PlayTester, making it substantially more stringent than existing metrics. This approach ensures Play@k captures hidden defects (e.g., collision detection failures, event handling inconsistencies, temporal property violations) that appear correct under unit testing but manifest as critical behavioral flaws during interactive execution. PlayEval emphasizes event handling, state management, and physics/animation with standardized interaction scripts, enabling fair and reproducible model comparison under identical behavioral testing conditions.

Key Idea 3: Multi-Agent Framework with PlayTester for Hallucination Mitigation. We design a multi-agent framework in which PlayTester serves *both* as a rigorous behavioral evaluator *and* as a hallucination-reduction feedback source for GUI code generation. Beyond its evaluation role, the PlayTester provides behavioral feedback that enables collaborative agents to systematically address repository hallucination, a pervasive issue where models generate syntactically correct but behaviorally incorrect code. The framework integrates retrieval-augmented generation, GUI testing, and automated program repair (APR) in a closed-loop control via a multi-agent system where PlayTester acts as a behavioral oracle. PlayDeveloper generates repository-aware code while invoking tools to retrieve context from the codebase. PlayTester executes applications and provides precise behavioral diagnostics (e.g., collision detection failures, event handling inconsistencies, state transition violations) that guide subsequent generation attempts. PlayRefiner performs targeted repairs using these behavioral insights: (1) compilation errors trigger `Validator` invocations; (2) API misuse and undefined symbols prompt context-guided adaptations via `ContextSearchTool`; (3) behavioral failures (e.g., incorrect physics) trigger logic adjustments based on PlayTester feedback. This approach transforms the PlayTester from a passive evaluator into an active participant that reduces hallucination by providing actionable behavioral signals throughout the generation process.

3 Benchmark and Evaluation

In this section, we introduce the benchmark construction, baselines, evaluation method, and preliminary studies. For all experiments, we generate $n = 3$ samples per problem and compute the

unbiased estimator with $k \leq 3$. All baselines are evaluated with their official configuration. We mitigate the influence of environmental factors (e.g., hardware differences) by providing standardized configurations and conducting all experiments with 5 repetitions.

3.1 Benchmark Construction

Following established best practices for code generation benchmarks [27, 33, 52, 69, 76], we present PlayEval, a repository-aware benchmark for GUI application code generation in Python, TypeScript, and JavaScript.

3.1.1 Selection Criteria. We curated repositories using the following criteria: (1) *Historically Active Development*: repositories with commits within the past 12 months at the time of initial selection, or demonstrated sustained development history (≥6 months of active maintenance) and achieved feature completeness before archival; (2) *Community validation*: most projects with >100 GitHub stars (We also include projects with ≤ 100 stars but have excellent deployability and representative of a certain category); (3) *Functional completeness*: applications that demonstrate complete GUI workflows rather than isolated snippets; (4) *Framework diversity*: coverage of major Python GUI frameworks, including PyQt, PySide, Tkinter, and Pygame; (5) *Exemplary value*: projects with clear structure and documentation suitable for code generation evaluation. We selected non-trivial functions central to the app's logic (e.g., game-loop, event-handlers) rather than utility helpers. We apply a filter to focus extraction on behavior-rich code: after excluding docstrings and decorator lines, we use a default threshold min_lines = 28 because we empirically find that functions with fewer than 28 lines are more likely to be utility helpers, simple accessors and rarely implement core interactive behaviors (e.g., game-loops or event-handlers).

Table 1. Comprehensive code complexity statistics of PlayEval across categories. CC = Cyclomatic Complexity (avg per file), ND = Nesting Depth (avg levels), CF/kLOC = Control Flow structures per 1000 LOC. LOC reports total lines of code across all projects.

Category	Projects	Files	LOC	Functions	Classes	CC↑	ND↑	CF/kLOC↑	LOC/Func↑	Test Cases↑
Game Emulation	1	89	26,699	1,350	289	8.2	11.4	24.1	19.7	86
Classic Games	6	34	2,605	72	19	6.7	8.8	19.4	36.2	24
Game Engine	1	43	12,484	661	87	9.5	13.2	28.3	18.8	27
Standalone Applications	24	387	123,442	2,069	172	10.8	10.8	32.1	55.7	1,539
Desktop Widgets	9	67	21,420	314	24	11.8	12.8	31.4	68.2	396
MMORPG Games	2	17	1,782	31	4	8.8	3.4	21.9	57.5	32
Total	**43**	**637**	**188,432**	**4,497**	**595**	**10.2**	**11.0**	**30.4**	**40.0**	**2,104**

3.1.2 Dataset Composition. As shown in Table 1, PlayEval comprises 43 diverse GUI applications including GUI-based games, productivity tools, multimedia applications, etc. The benchmark covers three programming languages: Python, TypeScript, and JavaScript. We selected Python as it is popular in AI/ML research and has rich GUI-bindings. Furthermore, most baselines (except OpenManus) are optimized only for Python, making it a "common supported language". We included TypeScript and JavaScript because, according to GitHub's 2025 Octoverse report[1], TypeScript became the most widely used programming language on GitHub and now serves as the default scaffold language for most mainstream frontend frameworks. PlayEval encompasses six major categories: (1) *Game Emulation* comprising a complete Game Boy emulator (i.e., PyBoy) with sophisticated hardware simulation capabilities; (2) *Classic Games* including traditional arcade-style games (i.e., 2048, Snake, Flappy Bird, Sudoku, Chrome Dragon) and strategy-based games like

[1] https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1

Solitaire and Chess variants; (3) *MMORPG Games* featuring two high-star TypeScript games (both with >1000 stars), CyberCodeOnline and biomes-game, to evaluate cross-language capability; (4) *Game Engine* featuring the Jupylet framework for educational game development; (5) *Standalone Applications* encompassing general-purpose applications implemented in Python, JavaScript, and TypeScript that can be further categorized into productivity tools (e.g., text editors, file managers), multimedia applications (e.g., media players), and web-based applications (e.g., Spotify client, Windows 11 simulator). This category also includes small-scale applications like calculators. (6) *Desktop Widgets* comprising interactive components (e.g., color pickers, range sliders). The classification distinction relies on window resizability: Standalone Applications typically support dynamic window resizing, whereas Desktop Widgets are often fixed-size components of a larger user interface;

3.1.3 Benchmark Structure. As shown in Fig. 2, PlayEval uses three evaluation metrics. The benchmark aims at repository-aware code generation, where each evaluation instance comprises: (1) **Function Signature** extracted from the original codebase, providing the exact method declaration with parameter types and return specifications; (2) **Requirement** automatically generated from the original function body using LLM-based docstring generation, which analyzes implementation logic to produce concise natural language descriptions of the function's purpose, behavior, parameters, and expected outcomes; (3) **Repository Context** containing relevant import statements, class definitions, and related function bodies from the same codebase to enable repository-aware code generation. The context for PlayCoder is the same as for human developers. To achieve it, we revert the repository to a certain state by `` `git checkout' ``. Requirements were generated by GPT-4o-mini and manually verified on a subset ($\approx$ 10%) by 3 developer experts. Through a voting strategy, over 95.6% of the requirements were deemed high-quality. The original repository's unit tests serve as ground truth (line coverage: 47.2%, branch coverage: 32.1%), reflecting the inadequate test coverage of real-world projects. Therefore, Play@k is critical for interactive validation.

3.1.4 Evaluation Workflow. The evaluation pipeline proceeds through three stages:

Compilation and Execution Stage: Generated functions undergo Python compilation testing to measure *Exec@k*, the percentage of problems for which at least one solution among k samples executes successfully without runtime errors, syntax errors, or import failures. This metric evaluates basic code correctness and syntactic validity.

Unit Testing Stage: Functions that pass compilation are evaluated against comprehensive test suites to measure *Pass@k*, the percentage of problems for which at least one solution among k samples passes all provided unit tests. Due to the lack of sufficient test cases in the original projects, we automatically generate a more robust suite. These test cases are created using an LLM-based analysis of the original function implementations, covering unit tests, integration tests, functional tests, and edge cases with proper mocking and isolation strategies.

Behavioral GUI Testing Stage: For GUI applications, our specialized GUI Behavioral Testing performs interactive validation to measure *Play@k*, the percentage of problems for which at least one solution among k samples demonstrates correct behavioral semantics in live application environments. This testing is conducted using PlayTester (detailed in Section 3.3), which performs automated GUI interaction and validation. For games with explicit objectives (e.g., winning conditions), the testing strategy focuses on achieving game completion through strategic gameplay. For general GUI applications, the testing approach emphasizes comprehensive feature coverage through carefully curated interaction sequences. *Note that Exec@k and Pass@k are deterministic metrics, which are not affected by the reliability of LLM backbones.*

3.1.5 Advanced Complexity Analysis. The benchmark exhibits a fine-grained complexity profile that stresses modern code generation models along multiple dimensions:

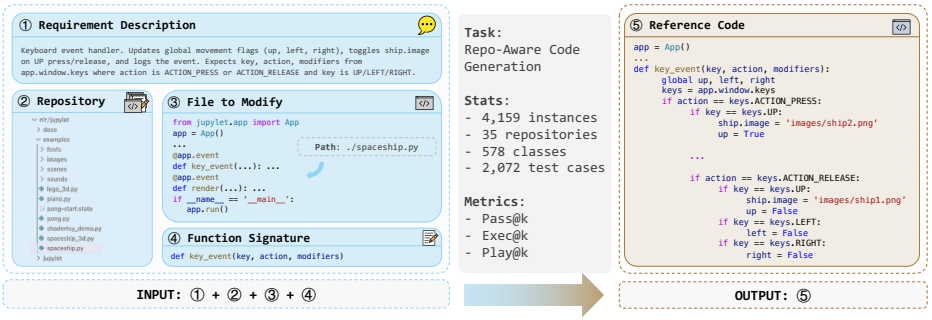


Fig. 2. The structure of PlayEval Data.

Cyclomatic Complexity: The dataset shows an average cyclomatic complexity of 10.2 per file, with GUI applications exhibiting the highest complexity (12.6) due to extensive event handling and user interaction logic. Game-related projects maintain moderate complexity (i.e., 6.5 to 9.5), reflecting focused algorithmic implementations, while the overall distribution ranges from simple utilities to sophisticated emulation systems.

Nesting and Control Flow: The benchmark exhibits an average nesting depth of 11.0 levels, with 107 files whose *per-file maximum* nesting depth exceeds 20 levels, creating substantial structural complexity. Control-flow analysis reveals 4,480 conditional statements (e.g., if statements), 814 loops (e.g., for statements), and 220 exception-handling blocks, yielding a control-flow density of 30.4 structures per 1000 lines of code, which is substantially higher than typical code generation benchmarks.

3.2 Baselines

We evaluate PlayCoder against: (i) state-of-the-art LLMs and (ii) advanced LLM-based approaches. Our baseline selection follows two key principles: (1) breadth of model capabilities, spanning general-purpose and coding-specialized LLMs from both open-source and closed-source families, and (2) coverage of representative LLM-based enhancement strategies, including widely-recognized prompt-based and agentic approaches proven effective for code generation tasks.

3.2.1 Basic LLMs. We consider ten state-of-the-art (SOTA) LLMs spanning diverse architectures, model families, and parameter scales. This selection encompasses both general-purpose LLMs (GPT-5, GPT-5-mini, GPT-4o, GPT-4o-mini, Claude-Sonnet-4, Claude-Sonnet-3.7, Grok-3-mini, GLM-4.5, DeepSeek-V3) and coding-specialized models (Qwen3-Coder), ensuring comprehensive coverage of different optimization objectives. The suite ranges from compact, efficient models to large-scale systems and includes both closed- and open-source variants with varying degrees of code specialization. For closed-source models, we choose GPT-5, GPT-5-mini, GPT-4o, GPT-4o-mini, Claude-Sonnet-4, Claude-Sonnet-3.7, and Grok-3-mini. For open-source models, we include Qwen3-Coder (480B), GLM-4.5 (355B), and DeepSeek-V3 (671B).

3.2.2 LLM-based Approaches. Beyond basic LLMs, we evaluate state-of-the-art LLM-based approaches that incorporate repository-aware retrieval, structured reasoning, or specialized code-generation capabilities. Our selection prioritizes methods with demonstrated effectiveness in code generation research and practical applicability to GUI application development:

- **SCoT** [31]: a widely-cited prompt-based approach using structured chain-of-thought prompting with an 8-step reasoning pipeline for code generation;
- **HCPCoder** [80]: Hierarchical context pruning with repository-aware prompt generation, retrieving code examples and import patterns via semantic similarity;

- **MetaGPT** [22]: a popular agentic framework (2051 citations) for code generation that simulates software development processes through specialized roles; the original paper [22] demonstrates its suitability for generating games (e.g., 2048, Snake), making it particularly relevant to our benchmark;
- **OpenManus** [38]: General-purpose multi-agent framework for complex task execution;
- **DeepCode** [29]: AI-powered development platform that automates code generation and implementation; the multi-agent system translates requirements into functional code.

3.3 PlayTester

PlayTester implements multi-modal testing capabilities through three specialized components that collaborate to validate GUI application behavior, of which the prompts are specified in [51].

3.3.1 Visual Observer Module. The `VisualObserver` captures application state via screenshots using `pyautogui` and `PIL`. It supports region-specific capture and performs window detection using platform-specific APIs (e.g., AppleScript on macOS, Win32 on Windows). The module caches recent frames (specifically the last three screenshots to differentiate animations from static states) and provides image comparison for state change detection. Screenshots are captured after each action execution (i.e., one second after the action is completed). The `VisualObserver` supports Windows, macOS, and X11-based Linux distributions. Wayland-based systems are excluded because Wayland's security architecture prevents cross-window screenshot capture and input injection via standard APIs. And support for Wayland-based systems is still under PR for pyautogui[2].

3.3.2 Action Executor Module. The `ActionExecutor` translates test strategies into specific GUI operations: `click(x, y)`, `type(text)`, `hotkey(keys)`, `press(key)`, `scroll(x, y, direction)`, `wait(duration)`, and `finish(success/failure)`. The module includes safety mechanisms (e.g., coordinate boundary checks and failsafe cursors) and maintains execution history for debugging. Actions are parsed from structured LLM output using the `ActionParser`.

3.3.3 Test Manager. The *Test Manager* integrates vision-language models to plan tests. It processes screenshots and textual context to generate strategies using specialized prompts for GUI analysis, test strategy generation, and action decision-making. The agents in all phases except for test strategy generation have one pre-defined prompt-template. We use two test-strategy prompt templates because GUI applications exhibit two distinct interaction regimes. Games typically have explicit objectives and terminal conditions (e.g., win/lose states, scores), so effective testing is goal-driven and focuses on reaching completion-critical states. In contrast, non-game applications often lack a natural terminal state, where effective testing is coverage-driven and emphasizes traversing UI workflows (e.g., menus) to maximize feature coverage. The behavioral testing phase is fully automated during verification. Tests are reusable across consistent screen resolutions.

3.3.4 Case Study. We illustrate PlayTester's evaluation approach using a representative 2048 implementation, focusing on how it perceives visual state, plans interactions, and verifies behavioral properties while maintaining coherent gameplay progression. As shown in Fig. 3, the tester simultaneously validates game functionality and maintains strategic progression in an early-game state (i.e., play and test simultaneously). Three key capabilities are highlighted:

- **Perception and State Extraction.** The `VisualObserver` captures the current screen and extracts structured state (e.g., r3c1=2, r3c4=2, r4c4=4; score=8), recognizing a sparse early-game configuration.

[2] https://github.com/asweigart/pyautogui/pull/936

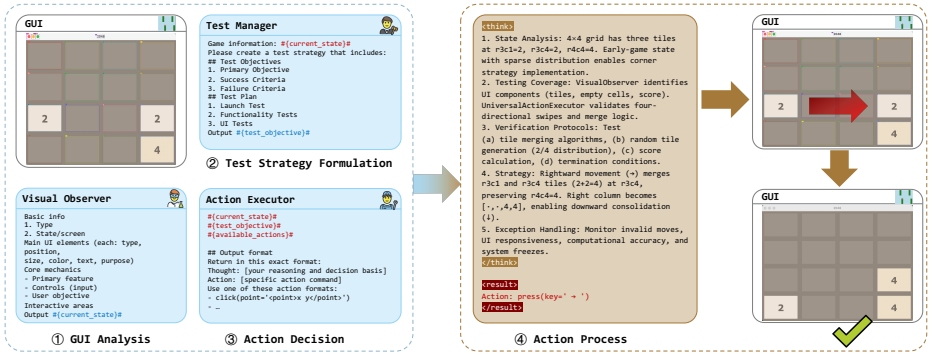


Fig. 3. Testing a 2048 implementation. Left: the rendered 4 × 4 grid shows tiles at (3,1), (3,4), and (4,4) with values 2, 2, and 4. Right: the agent's structured reasoning process including state analysis, coverage assessment, verification protocols, strategy selection, and exception-aware considerations that lead to a recommended rightward swipe.

- **Adaptive Strategy with Dual Objectives.** Based on real-time analysis, the agent selects a rightward swipe (→) that advances strategy (merging two 2-tiles into a 4 at r3c4) and validates key mechanics: swipe responsiveness, merge algorithm correctness, and score updates. This achieves coverage while preserving plausible gameplay.
- **Exception-Aware Validation.** The agent proactively checks invalid moves, UI freezes, numerical inconsistencies, and termination conditions. This approach reveals behavioral faults that unit test-based evaluation would miss.

3.4 Evaluation Metrics

Inspired by Pass@k [10], we propose Exec@k and Play@k, which estimate the probability that at least one of the top-k samples satisfies a task-specific criterion (i.e., successful execution or logically correct gameplay). For each problem, we draw n samples and let c_{succeed} denote the number of samples meeting the criterion; we then compute the unbiased estimator:

$$[\text{Exec, Pass, Play}]@k := \mathop{\mathbb{E}}_{\text{Problems}}\left[1 - \frac{\binom{n-c_{\text{succeed}}}{k}}{\binom{n}{k}}\right] \tag{1}$$

We evaluate four dimensions of code generation under this framework. *Exec@k*: at least one of the top-k samples executes successfully without syntax, or compile errors. We employ `Exec@k` and `Pass@k` following prior works (HumanEval [10], CODERANKER [23]). As in Section 3.1.4, our evaluation follows the pipeline (`Exec@k` → `Pass@k` → `Play@k`). A sample is evaluated for the next stage *only if* it passes the previous one. If the program failed the compilation before running, it is considered failed (Exec@k); *Pass@k*: at least one of the top-k samples passes all provided test cases; *Play@k*: at least one of the top-k samples exhibits correct GUI behavior under interactive testing by our GUI Behavioral Testing. `Play@k` is further enforced by constraints (e.g., "Did the game crash?") verified by logs and final-state screenshots. To answer "Are we maximizing successful results with minimal token usage?", we propose Efficiency@k, which measures the effectiveness-to-cost ratio:

$$\text{Efficiency@}k := \frac{\text{Play@}k}{AT_k}, \qquad AT_k := \frac{\text{Token}_k}{N \cdot 10^3} \tag{2}$$

where AT_k denotes the average number of tokens consumed per problem (in thousands), Token_k is the total token usage over all N problems when generating k samples per problem, and the 10^3 factor keeps the numerator and denominator on comparable scales (matching the caption of

Table 4). For $k = 1$, Efficiency@k simplifies through the combinatorial formula in Equation 1:

$$\text{Efficiency@1} = \frac{\text{Play@1}}{AT_1} = \frac{\mathbb{E}[c_{\text{succeed}}/n]}{AT_1} = \frac{\text{Succeed}_1/N}{\text{Token}_1/N} = \frac{\text{Succeed}_1}{\text{Token}_1}, \tag{3}$$

where c_{succeed} is the number of successful samples per problem and n is the number of samples per problem.

3.5 Preliminary Study 1: PlayTester Effectiveness

To establish PlayTester as a reliable evaluation framework, we conducted validation through human evaluation and statistical analysis before using it to evaluate code generation methods. We manually verified 100 successes and 100 failures identified by PlayTester, all randomly selected. To mitigate order bias, these samples were presented to evaluators in a randomized order. We recruited 3 software engineers with >5 years of experience in GUI development to serve as independent evaluators, treating human judgment as the ground truth. The evaluation reveals a 16% false-negative rate and a 5% false-positive rate. To quantify the consistency between PlayTester and human evaluators, we calculated Krippendorff's Alpha ($\alpha = 0.790$) and Kendall's Tau-b ($\tau_b = 0.795$). These statistical measures confirm substantial agreement and establish PlayTester as a reliable evaluation tool for GUI behavioral correctness. This demonstrates that PlayTester can serve as an automated evaluation framework before being incorporated into any code generation system.

3.6 Preliminary Study 2: Method Performance

Having established PlayTester as a reliable evaluation framework in Section 3.5, we now employ it to assess the performance of state-of-the-art LLMs and LLM-based methods.

3.6.1 Experiment Setup. We evaluate 10 state-of-the-art LLMs and 5 representative LLM-based enhanced methods on PlayEval. For base LLMs, we use standard few-shot prompting techniques to generate complete, repository-aware code implementations. For enhanced methods, we evaluate five representative approaches. *SCoT* employs structured chain-of-thought reasoning. *HCPCoder* uses hierarchical context pruning for repository awareness. *MetaGPT* applies multi-agent software development workflows. *OpenManus* utilizes collaborative agent coordination. *DeepCode* leverages specialized code understanding and generation capabilities. Each generated solution undergoes evaluation across three progressive criteria using PlayTester as the evaluation framework. *Exec@k* measures basic executability without runtime crashes. *Pass@k* evaluates correctness against provided unit tests. *Play@k* assesses semantic correctness through interactive GUI testing.

Table 2. Performance (%) of LLMs and LLM-based enhanced methods on PlayEval. We report mean values over 5 independent runs with 95% confidence intervals calculated using the Student's t-distribution.

LLMs / Methods	Size	Python						JavaScript						TypeScript					
		Exec@1	Exec@3	Pass@1	Pass@3	Play@1	Play@3	Exec@1	Exec@3	Pass@1	Pass@3	Play@1	Play@3	Exec@1	Exec@3	Pass@1	Pass@3	Play@1	Play@3
Base LLMs																			
GPT-4o-mini	-	10.3 ± 1.1	12.7 ± 1.4	2.9 ± 0.3	5.2 ± 0.5	2.1 ± 0.2	2.6 ± 0.4	11.1 ± 1.3	13.4 ± 1.2	3.1 ± 0.4	5.5 ± 0.6	2.3 ± 0.3	2.9 ± 0.5	8.7 ± 0.9	10.2 ± 1.1	2.2 ± 0.2	3.9 ± 0.4	1.6 ± 0.2	2.0 ± 0.3
GPT-5-mini	-	12.4 ± 2.0	13.7 ± 1.6	6.4 ± 0.3	6.7 ± 0.5	4.3 ± 0.5	5.2 ± 0.6	13.2 ± 1.8	14.5 ± 1.7	6.7 ± 0.4	7.1 ± 0.6	4.5 ± 0.6	5.5 ± 0.7	10.3 ± 1.6	11.4 ± 1.3	5.1 ± 0.3	5.4 ± 0.4	3.4 ± 0.4	4.1 ± 0.5
Grok-3-mini	-	13.9 ± 2.0	16.6 ± 1.8	7.0 ± 0.6	8.1 ± 0.9	4.6 ± 1.0	5.8 ± 0.6	14.7 ± 2.1	17.3 ± 1.9	7.4 ± 0.7	8.5 ± 1.0	4.8 ± 1.1	6.1 ± 0.7	11.5 ± 1.7	13.8 ± 1.5	5.6 ± 0.5	6.5 ± 0.7	3.7 ± 0.8	4.6 ± 0.5
Claude-Sonnet-3.7	-	10.8 ± 1.7	13.1 ± 1.1	6.1 ± 1.1	9.6 ± 1.8	4.7 ± 0.4	7.5 ± 1.0	11.5 ± 1.8	13.9 ± 1.2	6.5 ± 1.2	10.1 ± 1.9	5.0 ± 0.5	7.9 ± 1.1	9.0 ± 1.4	10.9 ± 0.9	4.9 ± 0.9	7.7 ± 1.4	3.8 ± 0.3	6.0 ± 0.8
Claude-Sonnet-4	-	**17.9 ± 2.1**	**18.6 ± 3.5**	**10.1 ± 0.7**	**13.0 ± 2.2**	6.4 ± 0.6	**9.9 ± 0.8**	**19.3 ± 2.3**	19.7 ± 3.7	**10.6 ± 0.8**	**13.7 ± 2.3**	6.7 ± 0.7	**10.4 ± 0.9**	**14.5 ± 1.7**	**16.7 ± 2.9**	**8.1 ± 0.6**	**10.4 ± 1.8**	5.1 ± 0.5	**7.9 ± 0.6**
GPT-4o	-	13.7 ± 3.4	13.8 ± 1.6	8.1 ± 0.5	8.9 ± 0.8	3.8 ± 0.4	6.7 ± 1.1	14.6 ± 3.2	14.6 ± 1.7	8.5 ± 0.6	9.4 ± 0.9	4.0 ± 0.5	7.1 ± 1.2	11.3 ± 2.8	11.5 ± 1.3	6.5 ± 0.4	7.1 ± 0.6	3.0 ± 0.3	5.4 ± 0.9
GPT-5	-	17.4 ± 1.9	17.5 ± 2.2	8.6 ± 1.1	10.2 ± 0.8	**6.6 ± 1.6**	6.9 ± 1.5	18.5 ± 2.0	18.6 ± 2.3	9.1 ± 1.2	10.7 ± 0.9	**8.0 ± 1.7**	8.9 ± 1.6	14.4 ± 1.6	14.5 ± 1.8	6.9 ± 0.9	8.2 ± 0.6	**6.1 ± 1.3**	5.2 ± 1.2
Qwen3-Coder	480B	14.0 ± 1.8	**18.8 ± 4.7**	5.2 ± 0.8	5.9 ± 1.1	4.9 ± 0.8	6.1 ± 0.4	14.8 ± 1.9	**19.8 ± 4.9**	5.5 ± 0.9	6.2 ± 1.2	5.2 ± 0.9	6.4 ± 0.5	11.6 ± 1.5	15.6 ± 3.9	4.2 ± 0.6	4.7 ± 0.9	3.9 ± 0.6	4.9 ± 0.3
GLM-4.5	355B	7.6 ± 2.1	**17.8 ± 0.9**	7.3 ± 1.1	7.1 ± 0.9	5.9 ± 1.7	6.3 ± 0.7	8.1 ± 2.2	18.7 ± 1.0	7.7 ± 1.2	8.0 ± 1.0	6.2 ± 1.8	6.6 ± 0.8	6.3 ± 1.7	14.7 ± 0.7	5.8 ± 0.9	6.1 ± 0.7	4.7 ± 1.4	5.0 ± 0.6
DeepSeek-V3	671B	11.7 ± 2.2	15.1 ± 1.1	5.8 ± 0.9	7.1 ± 0.5	5.0 ± 0.3	7.2 ± 1.1	12.5 ± 2.3	16.0 ± 1.2	6.1 ± 1.0	7.5 ± 0.6	5.3 ± 0.4	7.6 ± 1.2	9.7 ± 1.8	12.5 ± 0.9	4.6 ± 0.7	5.7 ± 0.4	4.0 ± 0.2	5.8 ± 0.9
LLM-based Enhanced Methods (GPT-5-mini as backbone LLM)																			
SCoT [31]	-	13.8 ± 1.3	15.2 ± 1.3	4.7 ± 0.5	7.0 ± 0.6	4.8 ± 0.3	6.0 ± 0.5	14.6 ± 1.5	16.2 ± 1.2	5.9 ± 0.7	7.6 ± 1.2	5.2 ± 0.6	6.1 ± 1.0	11.3 ± 1.2	12.7 ± 0.9	4.5 ± 0.5	5.8 ± 0.9	3.9 ± 0.4	4.6 ± 0.7
HCPCoder [80]	-	12.3 ± 2.1	12.8 ± 2.9	1.7 ± 0.1	3.5 ± 0.5	0.3 ± 0.0	0.3 ± 0.1	14.1 ± 0.6	14.6 ± 3.1	1.7 ± 0.5	3.5 ± 0.3	0.3 ± 0.1	0.3 ± 0.1	10.9 ± 0.4	11.4 ± 2.5	1.3 ± 0.3	2.6 ± 0.2	0.2 ± 0.1	0.2 ± 0.1
MetaGPT [22]	-	12.6 ± 0.7	13.0 ± 1.8	6.6 ± 0.5	10.3 ± 1.1	4.0 ± 0.7	4.4 ± 0.6	13.3 ± 1.9	13.6 ± 2.5	7.5 ± 1.9	9.9 ± 1.6	4.5 ± 0.7	4.9 ± 0.5	10.3 ± 1.5	10.6 ± 2.0	5.7 ± 1.4	7.5 ± 1.2	3.4 ± 0.5	3.7 ± 0.3
OpenManus [38]	-	12.3 ± 0.9	15.4 ± 2.6	8.1 ± 2.0	12.2 ± 1.2	5.3 ± 1.2	5.6 ± 0.9	13.1 ± 1.0	16.3 ± 2.7	8.5 ± 2.1	12.8 ± 1.3	5.6 ± 1.3	5.9 ± 1.0	10.2 ± 0.7	12.8 ± 2.2	6.5 ± 1.6	9.8 ± 1.0	4.2 ± 1.0	4.5 ± 0.7
DeepCode [29]	-	**17.1 ± 3.8**	**17.9 ± 6.4**	**10.5 ± 1.3**	**14.2 ± 2.2**	**6.0 ± 1.0**	**6.4 ± 1.0**	**18.2 ± 4.0**	**19.0 ± 6.7**	**11.0 ± 1.4**	**14.9 ± 2.3**	**5.9 ± 1.6**	**6.7 ± 1.1**	**14.1 ± 3.1**	**14.8 ± 5.3**	**8.4 ± 1.0**	**11.4 ± 1.8**	**5.3 ± 0.8**	**6.0 ± 0.4**

3.6.2 Results. Table 2 presents the performance of 10 state-of-the-art LLMs and 5 representative LLM-based enhanced methods on PlayEval across three programming languages and all evaluation metrics. The results reveal striking performance degradation as evaluation criteria become more stringent across all baselines, with notable cross-language performance variations.

Base LLM Performance: Among base LLMs, Claude-Sonnet-4 demonstrates the strongest performance across all three languages, maintaining its lead in both execution and behavioral validation metrics. However, even top-performing models achieve relatively modest behavioral correctness rates, with the best Play@3 scores remaining in the single digits for TypeScript implementations. Most models exhibit significant performance drops from execution to behavioral validation across all languages. Notably, the execution-to-behavior gap widens progressively from JavaScript to Python to TypeScript, suggesting that syntactic correctness does not reliably predict behavioral validity in statically-typed GUI applications.

LLM-based Enhanced Methods: Evaluation of state-of-the-art LLM-based enhancement methods reveals limited and inconsistent improvements over base models across all three languages. Prompting based methods (e.g., SCoT) show marginal improvements in Python and JavaScript but fail to bridge the performance gap for TypeScript implementations. Repository-aware approaches (e.g., HCPCoder) demonstrate catastrophic results across all languages, particularly in behavioral validation, indicating that context retrieval alone cannot address the semantic complexity of GUI applications. Multi-agent frameworks exhibit mixed results, with some methods (OpenManus) showing modest improvements in Python and JavaScript while others (MetaGPT) experience performance degradation relative to base models. DeepCode, despite specialized code understanding capabilities, achieves strong execution success but demonstrates limited behavioral validation improvements, with this pattern consistent across all three languages. Critically, no enhanced method successfully narrows the cross-language performance gap, suggesting that current enhancement strategies do not adequately address language-specific challenges in GUI code generation.

Answer to Preliminary Study: *Evaluation across state-of-the-art LLMs and methods reveals their limitations on PlayEval. The consistent failures across existing approaches indicate that repository-level GUI code generation remains challenging, even with prompting and agentic approaches.*

4 Our Approach: PlayCoder

To address the critical challenge of repository-aware GUI application code generation, we propose PlayCoder, a novel multi-agent framework that leverages two specialized agents as shown in Fig. 4: (1) PlayDeveloper: a repository-aware agent for code generation, and (2) PlayRefiner: an automated program repair (APR) agent to iteratively refine code based on behavioral testing feedback.

4.1 Multi-Agent Collaboration Workflow

We detail the collaboration workflow between the two agents in PlayCoder and then summarize the shared processes and telemetry that support this workflow.

4.1.1 Workflow Phases. The two agents collaborate through a structured test & repair cycle to achieve both syntactic correctness and behavioral alignment:

(1) **Context-Aware Generation.** PlayDeveloper receives specifications and generates repository-aware GUI application code using retrieved patterns and module structures.
(2) **Behavioral Testing.** The generated application is evaluated through automated behavioral testing (Section 3.3): the Visual Observer Module captures application state, the Test Manager

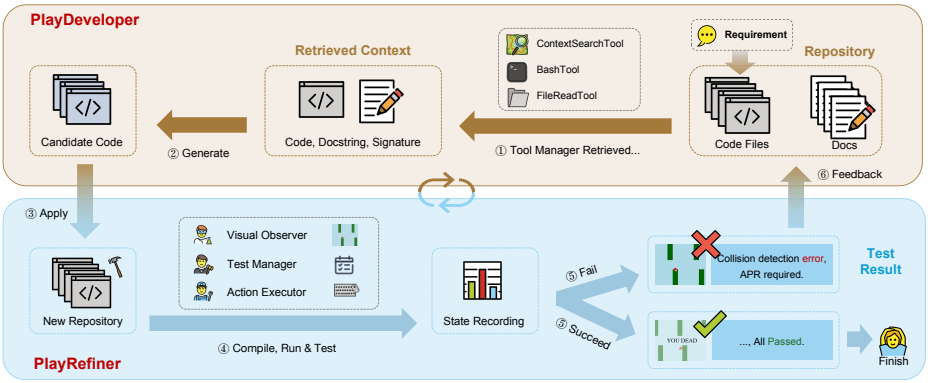


Fig. 4. The overview of PlayCoder.

plans interaction sequences using vision-language analysis, and the Action Executor Module executes tests to collect behavioral signals.

(3) **Diagnosis & Repair.** PlayRefiner analyzes execution traces and testing feedback from the behavioral testing modules, synthesizes patches with repository context, and applies fixes with compilation/runtime checks.

(4) **Iterative Feedback.** The updated application is re-evaluated through automated behavioral testing, checking behavior against specifications, including interactive semantics (e.g., collision handling, event responses, and state transitions). Each cycle refines code quality and testing strategy based on accumulated feedback. This process continues until the specified behavioral criteria are met or the iteration limit is reached. Concretely, the generate-refine loop terminates early once the application passes all per-sample behavioral checks enforced by PlayTester, and otherwise runs up to a maximum of $T = 6$ iterations.

4.1.2 Trajectory Recording. The multi-agent framework maintains comprehensive execution logs through the `AgentTrajectory` tool. This tool logs LLM interactions, tool usage, token consumption, screenshots, actions, and decision points. This comprehensive logging enables diagnosis, ablation studies, reproducibility, and APR prioritization.

4.1.3 Sandboxing & Determinism. Applications execute in sandboxed environments with deterministic seeding and controlled timing. The framework emits standardized logs to ensure fair, reproducible comparisons and to provide precise failure signals to the Testing and APR agents.

4.2 PlayDeveloper

PlayDeveloper implements a tool-based architecture dedicated to repository-aware code generation. The agent supports multiple LLM providers (e.g., OpenAI, Anthropic) and employs a modular tool ecosystem for context collection and code generation. The key components of `ToolManager` include: (1) `ContextSearchTool` for retrieving relevant code examples and import patterns from the repository context using grep-based searching, (2) `FileReadTool` for file operations, (3) `BashTool` for executing shell commands, and (4) `ConversationTool` for maintaining dialogue sessions. We use few-shot prompting with standard requirement-code examples.

4.3 PlayRefiner

This agent performs APR driven by behavioral feedback and execution traces. The agent coordinates validation tooling and orchestrates iterative fixes until behavioral criteria are satisfied.

4.3.1 Repair Tooling. PlayRefiner coordinates three core tools: (1) `ContextSearcher` for retrieving repository-aware APIs and import patterns during code repair, (2) `Validator` for syntax/AST checks and fast compile gating, and (3) `Executor` for executing the target program in a sandbox to capture runtime and behavioral signals.

4.3.2 Repair Workflow. The APR loop proceeds in five phases: (1) *Diagnosis* aggregates compiler output, runtime logs, and behavioral testing reports (screenshots, actions, and unexpected behaviors) into actionable failure summaries, (2) *Patch Generation* proposes minimal edits guided by retrieved context, (3) *Patch Application* writes changes atomically to the repository, (4) *Build & Runtime Validation* compiles and executes the application, followed by behavioral re-evaluation through the testing modules (Section 3.3), and (5) *Iterative Refinement* repeats up to a fixed budget or until behavioral criteria are met.

5 Results

To evaluate the effectiveness and efficiency of PlayCoder, we formulate three research questions:

- **RQ-1: Effectiveness Study.** *How does the performance of PlayCoder compare with the baselines?*
- **RQ-2: Efficiency Study.** *How does the efficiency of PlayCoder compare with the baselines?*
- **RQ-3: Ablation Study.** *How do different components and models affect PlayCoder's effectiveness?*

Statistical Reporting. We report the mean of each metric over 5 independent runs. Each run uses a distinct predefined random seed assigned to a unique integer, while keeping the LLM sampling temperature fixed at $T = 0.3$, prompts identical without dynamic elements, and all external tools and database states reset before each run. We quantify uncertainty using 95% confidence intervals for the mean, computed with the Student's t-distribution, which is suitable for small sample sizes. We observe that many improvements are stable with relatively small intervals, while some close comparisons show overlapping intervals and should be interpreted as comparable rather than decisively better. The variance mainly comes from stochastic LLM sampling and multi-agent interaction trajectories, and it does not change the overall ranking trends reported in this section.

5.1 RQ-1: How does PlayCoder perform compared to baselines?

Objective. This research question evaluates the effectiveness of PlayCoder against state-of-the-art repository-aware code generation baselines. We investigate whether our multi-agent approach, which combines automated program repair with dynamic GUI testing, provides substantial improvements over existing methods that focus solely on enhanced prompting strategies. Specifically, we compare PlayCoder against five representative baselines: SCoT [31], HCPCoder [80], MetaGPT [22], OpenManus [38], and DeepCode [29] evaluating performance across Exec@k, Pass@k, and Play@k to assess both syntactic correctness and behavioral validity.

Experimental Design. We conduct experiments using three different LLMs to ensure fair comparison. For SCoT, we implement structured chain-of-thought prompting following an 8-step construction pipeline that guides LLMs through systematic reasoning before code generation. The approach uses fixed demonstration examples covering sequential, branch, and loop programming structures, embedding step-by-step reasoning as line comments within the prompt template. The method first provides high-level instructions, then presents demonstration examples with structured reasoning patterns, and finally requests the model to follow the same reasoning approach for the target task. HCPCoder implements hierarchical context pruning with repository-aware prompt generation. Both SCoT and HCPCoder operate as single-shot generation methods without iterative refinement capabilities. The five baselines are evaluated across three backbone LLMs.

Results. Table 3 presents cross-language performance across three LLMs (GPT-5-mini, Claude-Sonnet-3.7, Qwen3-Coder) evaluating Python, JavaScript, and TypeScript. JavaScript demonstrates

comparable or superior performance to Python across most methods (e.g., PlayCoder: 30.9% vs 26.8% Exec@3), while TypeScript exhibits systematically lower performance, since there are complex MMORPG games in our benchmark implemented in TypeScript. With GPT-5-mini, PlayCoder substantially outperforms all baselines across languages: compared to SCoT, improvements include 11.6pp in Exec@3 and 12.7pp in Pass@3; compared to HCPCoder, 14.0pp and 16.2pp respectively. Notably, all baselines struggle with behavioral validation (Play@k). HCPCoder achieves near-zero Play@k across all languages despite sophisticated repository retrieval and token consumption. Prompting methods (SCoT) and multi-agent frameworks (MetaGPT, OpenManus) show limited improvements, failing to capture GUI requirements regardless of language. Cross-language patterns remain stable across LLM backbones, confirming language-specific trends reflect code generation difficulty rather than model biases. To address evaluator bias concerns [83], we validated with Claude-Sonnet-3.7, observing minimal ranking impact (Play@3 variations 0.9%-3.7%).

Table 3. Performance (%) of PlayCoder compared to baselines across different LLMs. We report mean values over 5 independent runs with 95% confidence intervals calculated using the Student's t-distribution.

Methods	LLMs	Python						JavaScript						TypeScript					
		Exec@1	Exec@3	Pass@1	Pass@3	Play@1	Play@3	Exec@1	Exec@3	Pass@1	Pass@3	Play@1	Play@3	Exec@1	Exec@3	Pass@1	Pass@3	Play@1	Play@3
HCPCoder [80]	GPT-5-mini	12.3 ± 2.1	12.8 ± 2.9	1.7 ± 0.1	3.5 ± 0.5	0.3 ± 0.0	0.3 ± 0.1	14.1 ± 0.6	14.6 ± 3.1	1.7 ± 0.5	3.5 ± 0.3	0.3 ± 0.1	0.3 ± 0.1	10.9 ± 0.4	11.4 ± 2.5	1.3 ± 0.3	2.6 ± 0.2	0.2 ± 0.1	0.2 ± 0.1
	Claude-Sonnet-3.7	16.1 ± 2.6	18.3 ± 2.2	4.9 ± 0.7	7.0 ± 1.4	5.0 ± 0.5	6.0 ± 0.7	18.5 ± 2.9	21.3 ± 2.6	5.8 ± 1.0	8.3 ± 1.6	5.4 ± 0.7	6.5 ± 0.9	13.2 ± 2.1	15.0 ± 1.8	3.9 ± 0.6	5.6 ± 1.1	4.0 ± 0.4	4.8 ± 0.6
	Qwen3-Coder	15.0 ± 1.9	18.1 ± 3.2	6.7 ± 1.2	12.0 ± 2.0	4.9 ± 1.0	6.5 ± 1.8	17.3 ± 2.3	20.7 ± 3.6	7.9 ± 1.5	13.8 ± 2.4	5.6 ± 1.2	7.4 ± 2.0	12.2 ± 1.5	14.8 ± 2.6	5.3 ± 0.9	9.5 ± 1.6	3.8 ± 0.8	5.2 ± 1.4
SCoT [31]	GPT-5-mini	13.8 ± 1.3	15.2 ± 1.3	4.7 ± 0.5	7.0 ± 0.6	4.8 ± 0.3	6.0 ± 0.5	14.6 ± 1.5	16.2 ± 1.2	5.9 ± 0.7	7.6 ± 1.2	5.2 ± 0.6	6.1 ± 1.0	11.3 ± 1.2	12.7 ± 0.9	4.5 ± 0.5	5.8 ± 0.9	3.9 ± 0.4	4.6 ± 0.7
	Claude-Sonnet-3.7	15.2 ± 3.8	19.0 ± 2.7	9.2 ± 2.1	12.9 ± 1.5	6.0 ± 1.2	7.8 ± 0.6	17.6 ± 4.2	22.1 ± 3.1	10.9 ± 2.5	15.3 ± 1.8	7.1 ± 1.4	9.2 ± 0.7	12.4 ± 3.1	15.5 ± 2.2	7.3 ± 1.7	10.3 ± 1.2	4.8 ± 0.9	6.2 ± 0.5
	Qwen3-Coder	23.5 ± 6.1	24.4 ± 5.3	10.2 ± 1.5	15.5 ± 0.8	6.2 ± 0.7	8.9 ± 0.5	26.9 ± 6.8	27.9 ± 5.9	12.1 ± 1.8	18.3 ± 0.9	6.8 ± 0.9	10.5 ± 0.6	19.1 ± 5.0	19.9 ± 4.3	8.1 ± 1.2	12.4 ± 0.6	6.3 ± 0.6	7.1 ± 0.4
OpenManus [38]	GPT-5-mini	12.3 ± 0.9	15.4 ± 2.6	8.1 ± 2.0	12.2 ± 1.2	5.3 ± 1.2	5.6 ± 0.9	13.1 ± 1.0	16.3 ± 2.7	8.5 ± 2.1	12.8 ± 1.3	5.6 ± 1.3	5.9 ± 1.0	10.2 ± 0.7	12.8 ± 2.2	6.5 ± 1.6	9.8 ± 1.0	4.2 ± 1.0	4.5 ± 0.7
	Claude-Sonnet-3.7	19.3 ± 0.3	28.2 ± 4.8	10.5 ± 1.7	15.4 ± 1.0	5.8 ± 0.4	8.1 ± 1.2	22.4 ± 0.4	32.8 ± 5.6	12.4 ± 2.0	18.2 ± 1.2	6.8 ± 0.5	9.6 ± 1.4	15.7 ± 0.2	23.0 ± 3.9	8.4 ± 1.4	12.3 ± 0.8	4.6 ± 0.3	6.5 ± 0.9
	Qwen3-Coder	23.2 ± 3.9	25.5 ± 1.7	13.0 ± 1.8	17.8 ± 2.7	6.7 ± 0.9	9.3 ± 0.8	26.7 ± 4.5	29.3 ± 1.9	15.4 ± 2.1	21.1 ± 3.2	7.9 ± 1.0	11.0 ± 0.9	18.9 ± 3.2	20.8 ± 1.4	10.4 ± 1.4	14.2 ± 2.2	5.3 ± 0.7	7.4 ± 0.6
MetaGPT [22]	GPT-5-mini	12.6 ± 0.7	13.0 ± 1.8	6.6 ± 0.5	10.3 ± 1.1	4.0 ± 0.7	4.4 ± 0.6	13.3 ± 1.9	13.6 ± 2.5	7.5 ± 1.9	9.9 ± 1.6	4.5 ± 0.7	4.9 ± 0.5	10.3 ± 1.5	10.6 ± 2.0	5.7 ± 1.4	7.5 ± 1.2	3.4 ± 0.5	3.7 ± 0.3
	Claude-Sonnet-3.7	13.5 ± 3.4	16.9 ± 0.6	7.5 ± 1.9	12.4 ± 1.0	3.9 ± 0.2	4.6 ± 0.5	15.6 ± 3.9	19.5 ± 0.7	8.9 ± 2.2	14.7 ± 1.2	4.6 ± 0.2	5.4 ± 0.6	11.0 ± 2.8	13.8 ± 0.5	6.0 ± 1.5	9.9 ± 0.8	3.1 ± 0.2	3.7 ± 0.4
	Qwen3-Coder	14.3 ± 2.0	22.2 ± 2.2	9.9 ± 0.8	11.6 ± 1.3	7.4 ± 0.6	8.8 ± 1.0	16.5 ± 2.3	25.6 ± 2.5	11.7 ± 0.9	13.7 ± 1.5	8.7 ± 0.7	10.4 ± 1.2	11.6 ± 1.6	18.1 ± 1.8	7.9 ± 0.6	9.3 ± 1.0	5.9 ± 0.5	7.0 ± 0.8
DeepCode [29]	GPT-5-mini	17.1 ± 3.8	17.9 ± 6.4	10.5 ± 1.3	14.2 ± 2.2	6.0 ± 1.0	6.4 ± 1.0	18.2 ± 4.0	19.0 ± 6.7	11.0 ± 1.4	14.9 ± 2.3	5.9 ± 1.6	6.7 ± 1.1	14.1 ± 3.1	14.8 ± 5.3	8.4 ± 1.0	11.4 ± 1.8	5.3 ± 0.8	6.0 ± 0.4
	Claude-Sonnet-3.7	29.4 ± 3.8	32.3 ± 4.6	15.6 ± 1.4	17.2 ± 2.6	5.3 ± 0.5	10.1 ± 1.1	34.1 ± 4.4	37.5 ± 5.3	18.4 ± 1.6	20.3 ± 3.0	6.2 ± 0.6	11.9 ± 1.3	24.0 ± 3.1	26.4 ± 3.8	12.5 ± 1.1	13.8 ± 2.1	4.2 ± 0.4	8.1 ± 0.9
	Qwen3-Coder	27.4 ± 6.3	37.6 ± 6.3	13.3 ± 1.2	19.4 ± 2.9	4.9 ± 1.1	9.8 ± 1.4	31.6 ± 7.3	43.4 ± 7.3	15.8 ± 1.4	23.0 ± 3.4	5.8 ± 1.3	11.6 ± 1.6	22.4 ± 5.1	30.7 ± 5.1	10.6 ± 0.9	15.5 ± 2.3	3.9 ± 0.9	7.8 ± 1.1
PlayCoder	GPT-5-mini	23.0 ± 1.4	26.8 ± 2.4	12.7 ± 1.1	19.7 ± 2.1	8.3 ± 1.3	9.8 ± 2.1	26.5 ± 1.6	30.9 ± 2.8	15.0 ± 1.3	23.3 ± 2.5	9.8 ± 1.5	11.6 ± 2.5	18.7 ± 1.1	21.8 ± 2.0	10.2 ± 0.9	15.8 ± 1.7	6.6 ± 1.0	7.8 ± 1.7
	Claude-Sonnet-3.7	29.5 ± 2.7	35.6 ± 4.4	15.5 ± 0.6	19.4 ± 2.9	13.9 ± 0.9	17.4 ± 1.3	34.1 ± 3.1	41.2 ± 5.1	18.3 ± 0.7	23.0 ± 3.4	16.4 ± 1.1	20.6 ± 1.5	24.0 ± 2.2	29.0 ± 3.6	12.4 ± 0.5	15.5 ± 2.3	11.1 ± 0.7	13.9 ± 1.0
	Qwen3-Coder	**32.4 ± 5.8**	**38.1 ± 4.9**	**17.6 ± 2.1**	**22.0 ± 3.2**	**16.0 ± 1.1**	**18.9 ± 4.0**	**37.4 ± 6.7**	**44.0 ± 5.7**	**20.8 ± 2.5**	**26.0 ± 3.8**	**18.9 ± 1.3**	**22.4 ± 4.7**	**26.4 ± 4.7**	**31.0 ± 4.0**	**14.1 ± 1.7**	**17.6 ± 2.6**	**12.8 ± 0.9**	**15.1 ± 3.2**

Answer to RQ-1: *PlayCoder outperforms all baseline methods across LLMs and languages. PlayCoder also shows clear advantages over other multi-agent baselines. These results demonstrate that existing prompt-engineering strategies and multi-agent approaches remain limited for GUI application generation. The performance differences highlight the necessity of combining dynamic GUI testing with iterative repair capabilities from visual signals and dynamic interaction.*

5.2 RQ-2: How does the efficiency of PlayCoder compare with the baselines?

Objective. This research question evaluates the computational efficiency and resource consumption of PlayCoder compared to baseline methods. Beyond effectiveness, computational efficiency considers practical deployment of GUI code generation systems. We investigate token consumption, processing time, and cost-effectiveness ratios across different methods and models to assess the trade-offs between enhancement sophistication and computational overhead.

Experimental Design. We conduct efficiency analysis across representative methods and pure LLM to ensure statistical reliability. For each method, we measure: (1) *Total token consumption* from API calls during code generation, (2) *Processing time* from task initiation to completion, and (3) *Per-function metrics* to normalize for workload differences.

Results. Table 4 presents efficiency metrics across methods and models, revealing significant trade-offs between enhancement sophistication and computational overhead. GPT-5-mini consumes 128K tokens with 4,267 tokens per function, achieving 4.3% Play@1 and 1.01 Efficiency@1. The base model shows moderate token consumption with reasonable behavioral validation performance. Enhancement methods exhibit varying efficiency characteristics compared to GPT-5-mini. SCoT increases token usage by 44% (184K vs 128K tokens) but achieves marginal improvement in

Table 4. Computational efficiency comparison across baselines and PlayCoder (GPT-5-mini as backbone LLM). Efficiency@k calculated as Play@k divided by tokens per function ($\times 10^3$). We report mean values over 5 independent runs with 95% confidence intervals calculated using the Student's t-distribution.

LLMs / Methods	Tokens ↓	Tokens / Func ↓	Play@1↑	Play@3↑	Efficiency@1↑	Efficiency@3↑
GPT-5-mini	**128K ± 7K**	**4267 ± 239**	4.3 ± 0.5	5.2 ± 0.6	1.01 ± 0.13	1.22 ± 0.16
SCoT	184K ± 23K	6135 ± 764	4.8 ± 0.3	6.0 ± 0.5	0.78 ± 0.11	0.98 ± 0.15
HCPCoder	373K ± 63K	12422 ± 2105	0.3 ± 0.0	0.3 ± 0.1	0.02 ± 0.00	0.02 ± 0.01
MetaGPT	148K ± 12K	4931 ± 416	4.0 ± 0.7	4.4 ± 0.6	0.81 ± 0.16	0.89 ± 0.14
OpenManus	176K ± 23K	5869 ± 771	5.3 ± 0.8	5.8 ± 0.3	0.90 ± 0.19	0.99 ± 0.14
DeepCode	252K ± 28K	8406 ± 939	6.0 ± 1.0	6.4 ± 1.0	0.72 ± 0.15	0.76 ± 0.15
PlayCoder	164K ± 21K	5480 ± 686	**8.3 ± 1.3**	**9.8 ± 2.1**	**1.51 ± 0.30**	**1.79 ± 0.44**

Play@1 (4.8% vs 4.3%), resulting in reduced efficiency (0.98 vs 1.22). HCPCoder presents the most concerning efficiency profile, consuming 191% more tokens (373K vs 128K) while achieving catastrophically poor behavioral validation (0.3% Play@1), resulting in extremely low Efficiency@1 (0.02), attributed to its poor repository context management. MetaGPT demonstrates moderate token efficiency with 148K tokens, maintaining 4.0% Play@1 performance with 0.89 efficiency. OpenManus requires substantial computational resources (176K tokens) but achieves competitive Play@1 performance (5.3%), resulting in moderate Efficiency@1 of 0.90. DeepCode shows the best behavioral validation among baselines (6.0% Play@1) but high token consumption (252K tokens) limits its Efficiency@1 to 0.72. PlayCoder demonstrates the 3rd best cost-effectiveness characteristics among all evaluated agentic methods. The framework consumes 164K tokens for 30 functions, achieving 5,480 tokens per function with 8.3% Play@1 performance. Our preliminary analysis on 164K tokens (Table 4) indicates that the Testing Phase accounts for ~22% consumption, while the Generation Phase (coding, iterative patching) accounts for ~78%. PlayCoder delivers significantly better performance per token consumed compared to all baseline methods.

Answer to RQ-2*: PlayCoder demonstrates the 3rd best cost-effectiveness (Token consumption) among all evaluated methods, achieving the highest Efficiency@k, and Play@k performance. These results demonstrate that PlayCoder provides acceptable resource utilization and behavioral validation performance for practical deployment scenarios.*

5.3 RQ-3: How do different components and models affect PlayCoder's effectiveness?

Objective. This research question examines the contribution of each component within the PlayCoder framework and evaluates its model-agnostic effectiveness across diverse LLM architectures. We investigate how GUI feedback, APR, and their integration impact performance across Exec@k, Pass@k, and Play@k. Additionally, we assess whether PlayCoder provides consistent benefits across different LLMs with varying capabilities, architectural designs, and parameter scales.

Table 5. The performance (%) of PlayCoder under different configurations, 5 runs + 95% CI (t-distribution).

LLMs	APR	GUI	Context	Python						JavaScript						TypeScript					
				Exec@1	Exec@3	Pass@1	Pass@3	Play@1	Play@3	Exec@1	Exec@3	Pass@1	Pass@3	Play@1	Play@3	Exec@1	Exec@3	Pass@1	Pass@3	Play@1	Play@3
GPT-5-mini	✓	✓	✓	23.0 ± 1.4	26.8 ± 2.4	12.7 ± 1.1	19.7 ± 2.1	8.3 ± 1.3	9.8 ± 2.1	26.5 ± 1.6	30.9 ± 2.8	15.0 ± 1.3	23.3 ± 2.5	9.8 ± 1.5	11.6 ± 2.5	18.7 ± 1.1	21.8 ± 2.0	10.2 ± 0.9	15.8 ± 1.7	6.6 ± 1.0	7.8 ± 1.7
	✗	✓	✓	15.5 ± 1.1	18.3 ± 2.3	6.8 ± 0.4	9.2 ± 0.5	4.9 ± 0.3	7.9 ± 0.8	17.9 ± 1.3	21.1 ± 2.7	7.7 ± 0.5	13.5 ± 0.6	5.8 ± 0.4	9.3 ± 0.9	12.6 ± 0.9	14.9 ± 1.9	4.8 ± 0.3	7.8 ± 0.4	3.9 ± 0.2	6.3 ± 0.6
	✓	✗	✓	20.5 ± 4.1	24.1 ± 5.9	10.4 ± 1.2	15.4 ± 1.0	3.0 ± 0.4	5.3 ± 1.0	23.6 ± 4.7	27.9 ± 6.8	12.3 ± 1.4	18.2 ± 1.2	3.6 ± 0.5	6.3 ± 1.2	16.7 ± 3.3	19.6 ± 4.8	8.3 ± 1.0	12.3 ± 0.8	2.4 ± 0.3	4.2 ± 0.8
	✗	✗	✓	12.8 ± 2.8	14.3 ± 2.6	5.8 ± 1.0	7.0 ± 1.0	4.5 ± 0.5	4.9 ± 0.3	15.1 ± 3.2	16.5 ± 3.0	6.8 ± 1.2	8.3 ± 1.2	5.3 ± 0.6	5.8 ± 0.4	11.0 ± 2.3	11.6 ± 2.1	4.6 ± 0.8	5.6 ± 0.8	3.6 ± 0.4	3.9 ± 0.2
	✗	✗	✗	12.6 ± 1.6	14.3 ± 1.0	5.4 ± 0.9	7.1 ± 0.7	4.2 ± 0.4	4.9 ± 0.4	14.5 ± 1.8	16.5 ± 1.2	6.4 ± 1.1	8.4 ± 0.8	5.0 ± 0.5	5.3 ± 0.6	10.2 ± 1.3	11.6 ± 0.8	4.3 ± 0.7	5.7 ± 0.6	3.4 ± 0.3	3.9 ± 0.3
Claude-3.7	✓	✓	✓	29.5 ± 2.7	35.6 ± 4.4	15.5 ± 0.6	19.4 ± 2.9	13.9 ± 0.9	17.4 ± 1.3	34.1 ± 3.1	41.2 ± 5.1	18.3 ± 0.7	23.0 ± 3.4	16.4 ± 1.1	20.6 ± 1.5	24.0 ± 2.2	29.0 ± 3.6	12.4 ± 0.5	15.5 ± 2.3	11.1 ± 0.7	13.9 ± 1.0
	✗	✓	✓	19.8 ± 2.3	21.5 ± 3.1	8.2 ± 0.7	12.4 ± 1.0	8.7 ± 2.0	9.7 ± 0.7	22.9 ± 2.7	24.8 ± 3.6	9.7 ± 0.8	14.7 ± 1.2	8.3 ± 2.4	11.5 ± 0.8	16.1 ± 1.9	17.5 ± 2.5	8.6 ± 0.6	9.9 ± 0.8	6.9 ± 1.6	7.8 ± 0.6
	✓	✗	✓	29.1 ± 5.6	34.1 ± 4.1	13.0 ± 2.3	16.5 ± 4.1	5.3 ± 0.7	6.9 ± 0.9	33.7 ± 6.5	39.4 ± 4.7	15.4 ± 2.7	19.5 ± 4.8	6.3 ± 0.8	8.2 ± 1.1	23.7 ± 4.6	27.7 ± 3.3	10.4 ± 1.8	13.2 ± 3.3	4.2 ± 0.6	5.5 ± 0.7
	✗	✗	✓	16.4 ± 2.5	17.2 ± 2.2	5.3 ± 0.5	7.0 ± 0.7	5.0 ± 0.2	5.9 ± 0.9	19.0 ± 2.9	19.9 ± 2.5	6.3 ± 0.6	8.3 ± 0.8	5.9 ± 0.2	7.0 ± 1.1	13.3 ± 2.0	14.0 ± 1.8	4.2 ± 0.4	5.6 ± 0.6	4.0 ± 0.2	4.7 ± 0.7
	✗	✗	✗	12.4 ± 1.0	13.7 ± 2.1	5.8 ± 0.7	9.0 ± 1.2	4.1 ± 0.8	7.6 ± 0.9	14.3 ± 1.2	15.8 ± 2.4	6.9 ± 0.8	10.7 ± 1.4	4.9 ± 0.9	9.0 ± 1.1	10.1 ± 0.8	11.1 ± 1.7	4.6 ± 0.6	7.2 ± 1.0	3.3 ± 0.6	6.1 ± 0.7
Qwen3-Coder	✓	✓	✓	**32.4 ± 5.8**	**38.1 ± 4.9**	**17.6 ± 2.1**	**22.0 ± 3.2**	**16.0 ± 1.1**	**18.9 ± 4.0**	**37.4 ± 6.7**	**44.0 ± 5.7**	**20.8 ± 2.5**	**26.0 ± 3.8**	**18.9 ± 1.3**	**22.4 ± 4.7**	**26.4 ± 4.7**	**31.0 ± 4.0**	**14.1 ± 1.7**	**17.6 ± 2.6**	**12.8 ± 0.9**	**15.1 ± 3.2**
	✗	✓	✓	20.8 ± 3.0	24.1 ± 3.5	10.4 ± 0.9	15.1 ± 1.6	7.7 ± 1.6	10.9 ± 2.0	24.0 ± 3.5	27.9 ± 4.0	12.3 ± 1.1	17.8 ± 1.9	9.1 ± 1.9	14.6 ± 2.4	16.9 ± 2.4	19.6 ± 2.8	8.3 ± 0.7	12.1 ± 1.3	6.2 ± 1.3	9.8 ± 1.6
	✓	✗	✓	31.3 ± 5.3	34.2 ± 3.0	9.7 ± 2.5	14.8 ± 0.8	7.5 ± 1.0	9.3 ± 3.0	36.2 ± 6.1	39.5 ± 3.5	11.5 ± 2.9	17.5 ± 0.9	8.9 ± 1.2	10.5 ± 3.5	25.5 ± 4.3	27.8 ± 2.4	7.8 ± 2.0	11.8 ± 0.6	6.0 ± 0.8	7.1 ± 2.4
	✗	✗	✓	18.0 ± 4.2	20.9 ± 5.2	9.1 ± 2.1	14.4 ± 3.0	4.9 ± 0.6	5.9 ± 0.9	20.8 ± 4.9	24.1 ± 6.0	10.8 ± 2.5	17.0 ± 3.5	5.8 ± 0.7	7.0 ± 1.1	14.7 ± 3.4	17.0 ± 4.2	7.3 ± 1.7	11.5 ± 2.4	3.9 ± 0.5	4.7 ± 0.7
	✗	✗	✗	14.9 ± 1.1	19.5 ± 1.4	5.6 ± 0.9	7.3 ± 0.4	5.0 ± 0.8	6.5 ± 0.6	17.2 ± 1.3	22.5 ± 1.6	6.6 ± 1.1	10.2 ± 0.5	5.9 ± 0.9	7.7 ± 0.7	12.1 ± 0.9	15.8 ± 1.1	4.5 ± 0.7	5.0 ± 0.3	4.0 ± 0.6	5.2 ± 0.5

Experimental Design. We conduct ablation studies across two dimensions: *component analysis* and *model robustness evaluation*. We evaluate configurations removing the APR (PlayCoder-no-APR), the GUI feedback (PlayCoder-no-gui), both agents (PlayCoder-no-apr-no-gui), the ContextSearchTool (PlayCoder-no-context), and all agentic components (PlayCoder-no-agent).

Results. Tables 6 and 5 present cross-language performance across Python, JavaScript, and TypeScript. JavaScript consistently outperforms Python, with leading models achieving 44.0% Exec@3 (Qwen3-Coder) and 44.5% Exec@3 (Claude-Sonnet-4) compared to 38.1% and 36.8% respectively in Python. TypeScript exhibits systematic degradation, with Qwen3-Coder reaching 31.0% Exec@3 and Claude-Sonnet-4 achieving 30.1% Exec@3, approximately 20% lower than Python baselines. These cross-language patterns remain stable across model tiers: intermediate models (GPT-5, GPT-4o, GPT-5-mini) demonstrate 14-16% JavaScript improvements and 15-25% TypeScript reductions, while smaller models (GPT-4o-mini, Grok-3-mini) maintain similar relative performance gaps. Ablation studies in Table 5 reveal consistent component criticality across languages. For GPT-5-mini, APR removal causes 8.5pp Exec@3 degradation in Python (26.8% to 18.3%), 9.8pp in JavaScript (30.9% to 21.1%), and 6.9pp in TypeScript (21.8% to 14.9%). GUI feedback elimination reduces Play@3 by 4.5pp in Python, 4.7pp in JavaScript, and 4.2pp in TypeScript, confirming its universal importance for behavioral validation. Higher-capability models exhibit amplified degradation: Claude-Sonnet-3.7 loses 14.1pp Python Exec@3 without APR, with proportional losses across JavaScript (16.4pp) and TypeScript (11.5pp). Complete component removal (no APR, GUI feedback, context) causes catastrophic performance collapse, validating the synergistic architecture of PlayCoder.

Table 6. Performance (%) of PlayCoder across different LLMs, 5 runs + 95% CI (t-distribution).

LLMs	Python						JavaScript						TypeScript					
	Exec@1	Exec@3	Pass@1	Pass@3	Play@1	Play@3	Exec@1	Exec@3	Pass@1	Pass@3	Play@1	Play@3	Exec@1	Exec@3	Pass@1	Pass@3	Play@1	Play@3
Qwen3-Coder	**32.4 ± 5.8**	**38.1 ± 4.9**	17.6 ± 2.1	22.0 ± 3.2	16.0 ± 1.1	18.9 ± 4.0	**37.4 ± 6.7**	44.0 ± 5.7	20.8 ± 2.5	26.0 ± 3.8	18.9 ± 1.3	22.4 ± 4.7	**26.4 ± 4.7**	**31.0 ± 4.0**	14.1 ± 1.7	17.6 ± 2.6	12.8 ± 0.9	15.1 ± 3.2
GPT-5	25.4 ± 5.9	29.5 ± 4.0	14.5 ± 2.5	19.1 ± 1.0	10.9 ± 0.7	12.4 ± 1.8	29.0 ± 5.9	33.7 ± 4.0	16.5 ± 2.5	21.8 ± 1.0	12.4 ± 0.7	14.2 ± 1.8	19.6 ± 4.6	22.8 ± 3.1	11.2 ± 1.9	14.8 ± 0.8	8.4 ± 0.5	9.6 ± 1.4
Claude-Sonnet-4	30.4 ± 1.5	36.8 ± 7.1	**18.0 ± 1.5**	**22.9 ± 2.1**	**17.1 ± 1.8**	**20.3 ± 2.5**	36.8 ± 1.8	**44.5 ± 3.6**	**21.8 ± 1.8**	**27.7 ± 2.5**	**20.7 ± 2.2**	**24.6 ± 3.0**	24.9 ± 1.2	30.1 ± 5.8	**14.7 ± 1.2**	**18.7 ± 1.7**	**14.0 ± 1.5**	**16.6 ± 2.0**
Claude-Sonnet-3.7	29.5 ± 2.7	35.6 ± 4.4	15.5 ± 0.6	19.4 ± 2.9	13.9 ± 0.9	17.4 ± 1.3	36.4 ± 3.3	43.9 ± 5.3	19.1 ± 0.7	23.9 ± 3.5	17.2 ± 1.1	21.5 ± 1.6	22.4 ± 2.0	27.0 ± 3.3	11.8 ± 0.5	14.7 ± 2.2	10.5 ± 0.7	13.2 ± 1.0
GPT-4o	23.6 ± 3.2	27.1 ± 2.8	14.1 ± 0.3	18.2 ± 1.6	9.4 ± 0.8	13.6 ± 2.8	27.5 ± 3.7	31.5 ± 3.3	16.4 ± 0.4	21.2 ± 1.9	10.9 ± 0.9	15.8 ± 3.3	17.8 ± 2.4	20.4 ± 2.1	10.6 ± 0.2	13.7 ± 1.2	7.1 ± 0.6	10.2 ± 2.1
GPT-5-mini	23.0 ± 1.4	26.8 ± 2.4	12.7 ± 1.1	19.7 ± 2.1	8.3 ± 1.3	9.8 ± 2.1	26.1 ± 1.6	30.4 ± 2.8	14.4 ± 1.3	22.3 ± 2.5	9.4 ± 0.6	11.1 ± 2.5	18.4 ± 1.1	21.5 ± 1.9	10.2 ± 0.9	15.8 ± 1.7	6.6 ± 0.4	7.8 ± 1.7
GLM-4.5	21.8 ± 2.5	24.8 ± 3.4	8.0 ± 1.4	11.9 ± 1.7	7.6 ± 1.0	9.6 ± 0.9	24.1 ± 2.9	27.4 ± 4.0	8.8 ± 1.6	13.1 ± 2.0	8.4 ± 1.2	10.6 ± 1.1	16.8 ± 1.9	19.1 ± 2.6	6.2 ± 1.1	9.2 ± 1.3	5.9 ± 0.8	7.4 ± 0.7
DeepSeek-V3	19.2 ± 1.4	23.3 ± 2.0	9.8 ± 0.5	14.5 ± 3.7	7.5 ± 0.8	9.9 ± 1.1	23.0 ± 1.6	27.9 ± 2.4	11.7 ± 0.6	17.4 ± 4.4	9.0 ± 1.0	11.9 ± 1.3	15.4 ± 1.1	18.7 ± 1.6	7.9 ± 0.4	11.7 ± 3.0	6.0 ± 0.6	8.0 ± 0.9
GPT-4o-mini	15.5 ± 2.0	19.0 ± 1.1	7.8 ± 0.8	11.0 ± 1.6	5.1 ± 0.5	7.8 ± 0.6	17.6 ± 2.3	21.5 ± 1.3	8.8 ± 0.9	12.5 ± 1.8	5.8 ± 0.6	8.8 ± 0.7	12.5 ± 1.6	15.4 ± 0.9	6.3 ± 0.6	8.9 ± 1.3	4.1 ± 0.4	6.3 ± 0.5
Grok-3-mini	13.5 ± 1.4	16.0 ± 2.8	4.8 ± 0.6	7.8 ± 1.2	4.7 ± 0.6	6.3 ± 0.6	16.5 ± 1.7	19.5 ± 3.4	5.9 ± 0.7	9.5 ± 1.5	5.7 ± 0.7	7.7 ± 0.7	10.1 ± 1.1	12.0 ± 2.1	3.6 ± 0.5	5.9 ± 0.9	1.5 ± 0.5	4.7 ± 0.5

Answer to RQ-3*: The ablation study reveals that both APR and GUI feedback make essential contributions to PlayCoder's effectiveness. The model robustness evaluation demonstrates that PlayCoder provides consistent improvements across diverse LLM architectures.*

5.4 Case Study

To demonstrate practical capabilities, we present a case study examining its multi-modal reasoning and adaptive testing strategies on a representative 2048 game implementation. This case study qualitatively showcases PlayCoder's advanced capacity for deep, context-aware reasoning and its ability to transform complex visual game states into structured testing protocols while maintaining strategic gameplay coherence. As shown in Figure 5, the scenario involves PlayCoder analyzing an early-game 2048 state. The case study demonstrates three aspects of PlayCoder's operation:

Multi-Phase Reasoning Architecture. PlayCoder employs a three-phase reasoning process that integrates testing objectives with gameplay optimization. First, the `VisualObserver` component captures and analyzes the current game state, identifying tile positions (r3c1=2, r3c4=2, r4c4=4) and recognizing the sparse early-game configuration. Second, the system constructs a testing protocol encompassing functional validation (merge algorithms, score computation, random tile generation), performance assessment (UI responsiveness), and boundary condition testing (invalid moves, termination detection). Third, PlayCoder synthesizes these analyses to select actions that simultaneously advance strategic gameplay while maximizing test coverage.

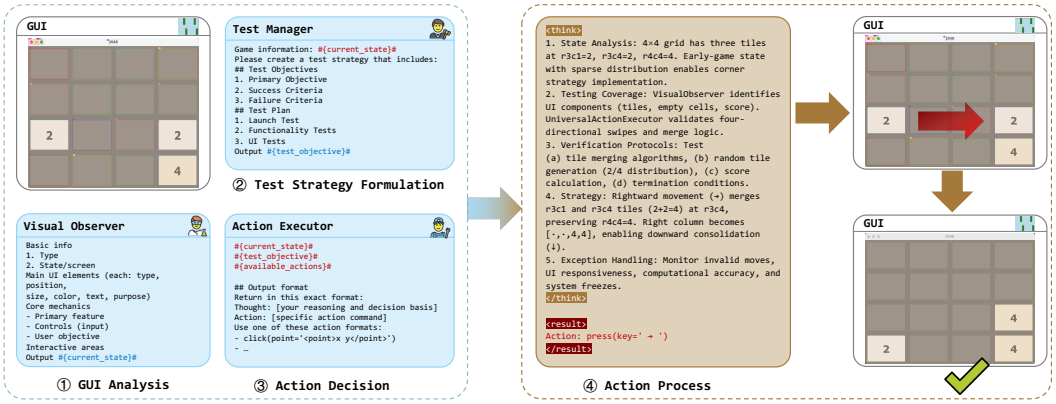


Fig. 5. Case study of PlayCoder testing a 2048 game.

Adaptive Test Strategy Generation. Unlike static testing approaches that rely on predefined test sequences, PlayCoder dynamically generates testing strategies based on real-time visual analysis. In the depicted scenario, the system recognizes that a rightward swipe (→) operation serves dual objectives: (1) implementing corner strategy optimization by merging the two 2-tiles into a 4-tile at position r3c4, and (2) validating critical game mechanics including swipe responsiveness, tile merger algorithms, and score calculation accuracy. This approach ensures comprehensive functionality validation while maintaining viable gameplay progression, addressing a key limitation of conventional GUI testing frameworks.

Exception-Aware Validation Protocol. The case study highlights PlayCoder's proactive approach to anomaly detection and edge case handling. The system explicitly considers failure scenarios including invalid move detection, UI freezing, computational accuracy errors, and improper termination conditions. This comprehensive exception handling demonstrates the framework's ability to identify behavioral failures that would be missed by compilation-based testing or unit test approaches, directly addressing the semantic correctness gaps identified in existing benchmarks. The presented case study exemplifies how PlayCoder transcends traditional testing limitations by integrating strategic reasoning with systematic validation. The system's ability to process visual information, generate structured testing protocols, and execute dual-objective actions (gameplay optimization + functionality validation) represents a significant advancement in automated GUI testing for interactive applications. This approach provides a robust foundation for detecting and repairing the silent logic flaws that characterize GUI application failures, thereby establishing behavioral correctness as a fundamental requirement for reliable code generation.

Success Case (2048). MetaGPT generated a version where the tile font color was white on a white background, making numbers invisible. Standard unit tests passed, but the game was unplayable. PlayTester detected a lack of visible changes and fixed the rendering code.

Failed Case. In scenarios requiring high-frequency feedback (e.g., a bug appearing only when running at $>$ 60 FPS) or long-duration survival (e.g., a crash occurring only after 2 minutes of gameplay in Flappy Bird), all baseline methods, including PlayCoder, failed. This failure mode is observed in 4 out of the 43 projects in PlayEval (roughly 9%), concentrated in fast-paced arcade games (Flappy Bird, Snake) and real-time physics simulations. We regard this as a limitation of behavioral testing under discrete polling, compounded by GPU compute bottlenecks that keep end-to-end inference latency from meeting the tight timing budgets of these scenarios; we leave higher-frequency visual sampling and lower-latency inference to future work.

6 Related Work

6.1 Code Generation with LLMs

Advances in pre-training have substantially improved neural code generation for both academia and industry [18, 35, 49, 56]. This progress has produced a rich family of large language models with strong coding ability [4, 12, 50, 55, 65, 66, 73, 75, 84], and subsequent work continues to scale model quality while also synthesizing the landscape through surveys [13, 30, 40, 63, 64]. Overall, the field has moved from comparatively shallow token prediction toward models that better capture complex structure and cross-file dependencies.

To adapt LLMs to diverse generation settings, a large body of work studies prompt engineering and introduces reusable interaction patterns, including Structured Chain-of-Thought [31, 74], Self-planning [25], Self-debug [11, 66], and Self-collaboration [16, 73]. Beyond generic prompting, repository-level methods make generation sensitive to project context: A3-CodGen [39] explicitly models local, global, and third-party library usage to support code reuse, while Shrivastava et al. [57] develop repository-aware prompt construction strategies. Most relevant to long-horizon completion, Hierarchical Context Pruning (HCP) [80] assembles prompts that respect topological dependency order and prune non-essential global and implementation detail, improving CrossCodeEval accuracy on five of six repository-trained code LLMs while increasing throughput. SolEval [52] introduces the first repository-level benchmark for Solidity code generation. PrefGen [53] builds on SolEval within a preference-driven training and evaluation framework, fine-tuning LLMs with supervised fine-tuning (SFT) and direct preference optimization (DPO) for repository-level Solidity generation. Parallel to these modeling ideas, agentic code generation frameworks decompose software work into coordinated procedures. MetaGPT [22] simulates a software organization via specialized roles; OpenManus [38] tackles complex tasks with a multi-agent execution stack; DeepCode [29] focuses on reliable automation for common development routines.

Beyond agents, recent benchmarks increasingly emphasize pragmatic coding competence [10, 21, 54]. LiveCodeBench [24] reduces evaluation leakage by continuously refreshing problem sets. CoderEval [76] foregrounds realistic coding scenarios, and Evocodebench [32] stresses generation inside practical software projects. ClassEval [17] shifts attention from isolated functions to class-level synthesis. Finally, SWE-Bench [26] measures GitHub issue resolution and has catalyzed a wave of follow-on studies on practical software engineering tasks [1, 2, 19, 20, 34, 60, 67, 71, 78, 79]. However, these benchmarks largely omit behavioral correctness for generated *GUI* applications. Moreover, mainstream generation stacks still emphasize compile- and test-oriented functional signals rather than grounding iteration in visual, interactive execution feedback.

6.2 GUI Interaction Understanding, Testing, and Generation

Traditional GUI automation relies on *rule-based* exploration. Random fuzzers such as Android Monkey exercise apps with pseudo-random inputs but lack systematic coverage and semantic checks [3]. Model- and search-based testing improves exploration: Dynodroid adds system awareness to input generation [44]; Sapienz jointly optimizes coverage, fault detection, and test suite size with multi-objective search [47]; Stoat learns stochastic state models for event sequences [58]. Earlier frameworks also compile event-flow graphs and widget hierarchies into scalable tests and oracles [48]. Collectively, these methods emphasize structural models and coverage more than validating *semantic* GUI behavior against natural-language intent. On the other hand, learning-based methods broaden GUI *understanding* and downstream automation. UIED [68] hybridizes CV and ML for cross-platform element detection; Screen Recognition [82] infers accessibility metadata from pixels; Owl Eyes [43] flags display defects visually. Deep GUI [72], ResPlay [81], and Baral

et al. [5] turn strong perception into black-box inputs, cross-platform record-and-replay, and mobile test oracles, respectively. Mansur et al. [46] further target UX risks via dark-pattern detection. Rico [14] supplies large-scale layouts for data-driven modeling; ScreenAI [8] improves widget recognition, captioning, and instruction following on screens. WebArena and Mind2Web [15, 85] benchmark multi-step web interaction. These resources support screen understanding and action planning, but not repository-aware synthesis and repair of full GUI application code. On *GUI generation and testing*, *pix2code* and *web2code* [6, 77] translate designs into code; *Seq2Act* [36] maps language to UI action traces; GPTDroid [42] and Humanoid [37] drive mobile exploration with LLMs or deep policies. Such systems mainly yield layouts, scripts, or action loops with limited guarantees on end-to-end runtime logic, a gap also reflected in benchmarks for automated GUI testing [5, 9]. Unlike coverage- or crash-oriented testers [44, 47, 58] and agents scored on completing tasks over existing GUIs [8, 15, 85], our approach feeds dynamic execution feedback into the code generation loop to detect and repair logic errors (e.g., rule violations in GUI games) that compile- and run-based checks fail to capture.

7 Threats to Validity

External Threats. PlayCoder's effectiveness is inherently constrained by the capabilities of underlying vision-language models. Current VLMs exhibit limitations in recognizing fine-grained GUI elements and interpreting complex visual semantics, which directly translate to constraints in our testing and validation capabilities. This threat will be mitigated by the evolution vision-language models in the future. Furthermore, the probabilistic nature of LLMs introduces inherent instability in complex contexts, causing response variations even if the temperature is set to zero. We mitigate this by conducting all experiments with 5 repetitions to ensure stable and reliable results.
Internal Threats. GUI-based behavioral testing relies on screenshot-based analysis, unable to capture every critical frame in dynamic applications (e.g., FPS games), potentially missing bug detection opportunities in unfavorable timing scenarios. Furthermore, the context retrieval mechanisms may face scalability challenges in large, complex repositories with extensive dependencies. The benchmark scenarios and number of applications evaluated may be limited in scope. We address this limitation by selecting applications that span multiple domains and platforms.

8 Conclusion

This paper addresses limitations in evaluating and generating GUI application code. We identify that existing benchmarks fail to capture behavioral correctness in interactive applications, where syntactically correct code can exhibit catastrophic logic flaws that traditional unit tests miss. We introduce PlayCoder, a novel multi-agent framework that integrates automated GUI testing with iterative program repair. Our approach employs PlayTester to simulate user interactions and detect behavioral deviations, paired with PlayRefiner that autonomously debugs and refines the generated code. Through systematic evaluation on diverse multilingual (Python, TypeScript, and JavaScript) GUI applications, we demonstrated that this collaborative framework significantly outperforms baseline methods, achieving higher rates of functional correctness and semantic alignment. Our work highlights the importance of runtime verification in complex code generation tasks and establishes a new paradigm for ensuring behavioral fidelity in interactive applications.

9 Data Availability

The replication package of this paper (including prompts, code, and datasets) is publicly available at https://github.com/Tencent/PlayCoder.

Acknowledgements

This work was supported in part by National Key Research and Development Program of China under Grant 2024YFB2705300, in part by the Shanghai Science and Technology Innovation Action Plan under Grant 23511100400.

References

[1] Reem Aleithan. 2025. Revisiting SWE-Bench: On the Importance of Data Quality for LLM-Based Code Models. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 235–236.

[2] Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. 2024. Swe-bench+: Enhanced coding benchmark for llms. *arXiv preprint arXiv:2410.06992* (2024).

[3] Android Developers. 2010. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey. Accessed: 2025-08-14.

[4] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[5] Kesina Baral, John Johnson, Junayed Mahmud, Sabiha Salma, Mattia Fazzini, Julia Rubin, Jeff Offutt, and Kevin Moran. 2024. Automating gui-based test oracles for mobile apps. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 309–321.

[6] Tony Beltramelli. 2017. pix2code: Generating Code from a Graphical User Interface Screenshot. *arXiv preprint arXiv:1705.07962* (2017).

[7] Yutong Bian, Xianhao Lin, Yupeng Xie, Tianyang Liu, Mingchen Zhuge, Siyuan Lu, Haoming Tang, Jinlin Wang, Jiayi Zhang, Jiaqi Chen, et al. 2025. You Don't Know Until You Click: Automated GUI Testing for Production-Ready Software Evaluation. *arXiv preprint arXiv:2508.14104* (2025).

[8] Federico Bianchi, Chiori Hori, Shalini Narayan, and et al. 2024. ScreenAI: A Vision-Language Model for UI and Document Understanding. *arXiv preprint arXiv:2404.08547* (2024).

[9] Dongping Chen, Yue Huang, Siyuan Wu, Jingyu Tang, Liuyi Chen, Yilin Bai, Zhigang He, Chenlong Wang, Huichi Zhou, Yiqiang Li, et al. 2024. Gui-world: A video benchmark and dataset for multimodal gui-oriented understanding. *arXiv preprint arXiv:2406.10819* (2024).

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

[11] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023).

[12] DeepSeek. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. https://huggingface.co/papers/2401.14196 Accessed: 2025-02-5.

[13] DeepSeek. 2025. DeepSeek-R1. https://github.com/deepseek-ai/DeepSeek-R1 Accessed: 2025-02-5.

[14] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Horton, and et al. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST)*. 845–854.

[15] Yuwei Deng, Xiao Liu, Yujia Xu, Wenpeng Yin, and et al. 2023. Mind2Web: Grounding Language Models to the Web with Continuous, Generalist Tasks. *arXiv preprint arXiv:2306.06070* (2023).

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

[17] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[18] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations*.

[19] Lianghong Guo, Wei Tao, Runhan Jiang, Yanlin Wang, Jiachi Chen, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. 2025. Omnigirl: A multilingual and multimodal benchmark for github issue resolution. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 24–46.

[20] Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Zejian Yuan, and Zejun Ma. 2025. SWE-Perf: Can Language Models Optimize Code Performance on Real-World Repositories? *arXiv preprint arXiv:2507.12415* (2025).

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

arXiv:2105.09938 (2021).

[22] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. International Conference on Learning Representations, ICLR.

[23] Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. *Advances in Neural Information Processing Systems* 35 (2022), 13419–13432.

[24] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. In *The Thirteenth International Conference on Learning Representations*.

[25] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology* 33, 7 (2024), 1–30.

[26] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *ICLR*.

[27] Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004* (2023).

[28] Geunwoo Kim, Pierre Baldi, and Stephen McAleer. 2023. Language models can solve computer tasks. *Advances in Neural Information Processing Systems* 36 (2023), 39648–39677.

[29] Data Intelligence Lab. 2025. DeepCode: Open Agentic Coding. https://github.com/HKUDS/DeepCode

[30] Caihua Li, Lianghong Guo, Yanlin Wang, Daya Guo, Wei Tao, Zhenyu Shan, Mingwei Liu, Jiachi Chen, Haoyu Song, Duyu Tang, et al. 2026. Advances and Frontiers of LLM-based Issue Resolution in Software Engineering: A Comprehensive Survey. *arXiv preprint arXiv:2601.11655* (2026).

[31] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–23.

[32] Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories. *arXiv preprint arXiv:2404.00599* (2024).

[33] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Zhi Jin, Hao Zhu, Huanyu Liu, Kaibo Liu, Lecheng Wang, Zheng Fang, et al. 2024. Deveval: Evaluating code generation in practical software projects. *arXiv preprint arXiv:2401.06401* (2024).

[34] Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. 2025. Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation. *arXiv preprint arXiv:2503.06680* (2025).

[35] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[36] Yushi Li, Qian Liu, Panupong Pasupat, Zi He, Yu Wang, Benjamin Hsu, and Yang Li. 2020. Mapping Natural Language Instructions to Mobile UI Action Sequences. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*. 1945–1955.

[37] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1070–1073.

[38] Xinbin Liang, Jinyu Xiang, Zhaoyang Yu, Jiayi Zhang, Sirui Hong, Sheng Fan, and Xiao Tang. 2025. OpenManus: An open-source framework for building general AI agents. doi:10.5281/zenodo.15186407

[39] Dianshu Liao, Shidong Pan, Xiaoyu Sun, Xiaoxue Ren, Qing Huang, Zhenchang Xing, Huan Jin, and Qinying Li. 2024. A3-CodGen: A Repository-Level Code Generation Framework for Code Reuse with Local-Aware, Global-Aware, and Third-Party-Library-Aware. *IEEE Transactions on Software Engineering* (2024).

[40] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[41] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2023. Chatting with gpt-3 for zero-shot human-like mobile automated gui testing. *arXiv preprint arXiv:2305.09434* (2023).

[42] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[43] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl eyes: Spotting ui display issues via visual understanding. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 398–409.

[44] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 224–234.

[45] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems* 36 (2023), 46534–46594.

[46] SM Hasan Mansur, Sabiha Salma, Damilola Awofisayo, and Kevin Moran. 2023. Aidui: Toward automated recognition of dark patterns in user interfaces. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1958–1970.

[47] Ke Mao, Mark Harman, Yue Jia, Yuanyuan Zhang, and Zheng Li. 2016. Sapienz: Multi-Objective Automated Testing for Android Applications. In *Proceedings of the 2016 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 94–105.

[48] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. 2003. Automatically Testing GUIs Using Event-Flow Graphs. *IEEE Transactions on Software Engineering* 29, 6 (2003), 531–555.

[49] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*.

[50] OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt/

[51] Zhiyuan Peng, Wei Tao, Xin Yin, Chenhao Ying, Yuan Luo, and Yiwen Guo. 2025. PlayCoder Replication Package: Source Code, Prompts, and Benchmark. https://github.com/Tencent/PlayCoder

[52] Zhiyuan Peng, Xin Yin, Rui Qian, Peiqin Lin, Yongkang Liu, Hao Zhang, Chenhao Ying, and Yuan Luo. 2025. SolEval: Benchmarking Large Language Models for Repository-level Solidity Smart Contract Generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. 4388–4411.

[53] Zhiyuan Peng, Xin Yin, Chenhao Ying, Chao Ni, and Yuan Luo. 2025. A Preference-Driven Methodology for High-Quality Solidity Code Generation. *arXiv preprint arXiv:2506.03006* (2025).

[54] Zhiyuan Peng, Xin Yin, Pu Zhao, Fangkai Yang, Lu Wang, Ran Jia, Xu Chen, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. 2026. RepoGenesis: Benchmarking End-to-End Microservice Generation from Readme to Repository. *arXiv preprint arXiv:2601.13943* (2026).

[55] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[56] Sijie Shen, Xiang Zhu, Yihong Dong, Qizhi Guo, Yankun Zhen, and Ge Li. 2022. Incorporating domain knowledge through task augmentation for front-end javascript code generation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1533–1543.

[57] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*. PMLR, 31693–31715.

[58] Ting Su, Guozhu Meng, Yang Chen, Ke Wang, Zhendong Su, Yang Liu, and others. 2017. Stoat: A Model-Based GUI Testing Tool for Android Applications. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 571–575.

[59] Hao Tang, Keya Hu, Jin Zhou, Si Cheng Zhong, Wei-Long Zheng, Xujie Si, and Kevin Ellis. 2024. Code repair with llms gives an exploration-exploitation tradeoff. *Advances in Neural Information Processing Systems* 37 (2024), 117954–117996.

[60] Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. 2024. Magis: Llm-based multi-agent framework for github issue resolution. *Advances in Neural Information Processing Systems* 37 (2024), 51963–51993.

[61] Chenxu Wang, Tianming Liu, Yanjie Zhao, Minghui Yang, and Haoyu Wang. 2025. LLMDroid: Enhancing Automated Mobile App GUI Testing Coverage with Large Language Model Guidance. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 1001–1022.

[62] Siyi Wang, Sinan Wang, Yujia Fan, Xiaolei Li, and Yepang Liu. 2024. Leveraging large vision-language model for better automatic web GUI testing. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 125–137.

[63] Yanlin Wang, Kefeng Duan, Dewu Zheng, Ensheng Shi, Fengji Zhang, Yanli Wang, Jiachi Chen, Xilin Liu, Yuchi Ma, Hongyu Zhang, et al. 2026. Towards an understanding of context utilization in code intelligence. *Comput. Surveys* (2026).

[64] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[65] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: empowering code generation with OSS-INSTRUCT. In *Proceedings of the 41st International Conference on Machine Learning*. 52632–52657.

[66] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 819–831.

[67] Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. 2025. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. *arXiv preprint arXiv:2501.05040* (2025).

[68] Mulong Xie, Sidong Feng, Zhenchang Xing, Jieshan Chen, and Chunyang Chen. 2020. UIED: a hybrid tool for GUI element detection. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1655–1659.

[69] Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Shuiguang Deng, et al. 2023. Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation. *arXiv preprint arXiv:2311.08588* (2023).

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

[71] John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, Diyi Yang, Sida Wang, and Ofir Press. 2025. SWE-bench Multimodal: Do AI Systems Generalize to Visual Software Domains?. In *The Thirteenth International Conference on Learning Representations*.

[72] Faraz YazdaniBanafsheDaragh and Sam Malek. 2021. Deep GUI: Black-box GUI input generation with deep learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 905–916.

[73] Xin Yin, Chao Ni, Tien N Nguyen, Shaohua Wang, and Xiaohu Yang. 2024. Rectifier: Code translation with corrector via llms. *arXiv preprint arXiv:2407.07472* (2024).

[74] Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. 2024. Thinkrepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1274–1286.

[75] Xin Yin, Chao Ni, Xiaodan Xu, and Xiaohu Yang. 2025. What You See Is What You Get: Attention-based Self-guided Automatic Unit Test Generation. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*.

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

[77] Sukmin Yun, Rusiru Thushara, Mohammad Bhat, Yongxin Wang, Mingkai Deng, Jinhong Wang, Tianhua Tao, Junbo Li, Haonan Li, Preslav Nakov, et al. 2024. Web2code: A large-scale webpage-to-code dataset and evaluation framework for multimodal llms. *Advances in neural information processing systems* 37 (2024), 112134–112157.

[78] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, et al. 2025. Multi-swe-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605* (2025).

[79] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, et al. 2025. SWE-bench Goes Live! *arXiv preprint arXiv:2505.23419* (2025).

[80] Lei Zhang, Yunshui Li, Jiaming Li, Xiaobo Xia, Jiaxi Yang, Run Luo, Minzheng Wang, Longze Chen, Junhao Liu, Qiang Qu, et al. 2025. Hierarchical context pruning: Optimizing real-world code completion with repository-level pretrained code llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 25886–25894.

[81] Shaokun Zhang, Linna Wu, Yuanchun Li, Ziqi Zhang, Hanwen Lei, Ding Li, Yao Guo, and Xiangqun Chen. 2023. Resplay: Improving cross-platform record-and-replay with gui sequence matching. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 439–450.

[82] Xiaoyi Zhang, Lilian De Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, et al. 2021. Screen recognition: Creating accessibility metadata for mobile applications from pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.

[83] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems* 36 (2023), 46595–46623.

[84] Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. OpenCodeInterpreter: Integrating Code Generation with Execution and Refinement. In *Findings of the Association for Computational Linguistics ACL 2024*. 12834–12859.

[85] Shuyan Zhou, Yuchen Xu, Xuezhi Li, Nathanael Schärli, and et al. 2023. WebArena: A Realistic Web Environment for Building Autonomous Agents. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*.

Received 2026-02-24; accepted 2026-03-24