

MACBENCH: Benchmarking Multi-turn Agentic Coding with Evolving Requirements in Frontend Web Development

Jiayi Gui¹, Bosi Wen², Haoke Zhang¹, Minglie Huang², Jie Tang², Cunxiang Wang^{1,†}

¹Zhipu AI, ²Tsinghua University,
iamjiayi0321@163.com, wangcunxiang303@gmail.com

Abstract

Modern coding agents are increasingly deployed for realistic software engineering, yet existing benchmarks primarily evaluate static, fixed-target tasks, failing to capture the long-horizon, multi-turn nature of real-world development. We introduce **MACBench**, a novel benchmark evaluating agents on sustained frontend web application development where text-based user requirements continuously evolve within a persistent workspace. Comprising 28 repository instances across 8 domains, MACBench features 699 dialogue turns and 956 evaluation items. It explicitly models realistic interaction patterns, challenging agents to maintain cross-turn coherence and continuous executability. Our dual-branch evaluation framework assesses both process robustness and final artifact quality, uniquely integrating intermediate viability checks with bounded runtime-feedback repair. Benchmarking 4 contemporary models reveals a severe reliability gap: the best-performing agent achieves only a 70.6% checklist success rate and a 10.7% strict instance success rate, struggling to satisfy requirements accumulated over long sessions. Furthermore, ablation studies reveal diminishing and sometimes negative returns from runtime-feedback repair: while it improves checklist success by an average of 10.4% and aids immediate failure recovery, excessive repair attempts frequently degrade final functional outcomes, suggesting they may introduce regressions or side effects in the evolving codebase. Finally, failure analysis identifies complex cross-turn application logic as the dominant bottleneck, accounting for 33–50% of unresolved errors.

1 Introduction

Modern coding agents are increasingly deployed for realistic software engineering, moving beyond

isolated code generation to repository-level interactions (Yang et al., 2024; Jimenez et al., 2024; Wang et al., 2025). However, real-world application development is inherently a long-horizon, multi-turn process. Users rarely provide a perfect specification upfront; instead, requirements iteratively evolve, constraints are refined, and architectures must adapt (Anthropic, 2026; Cassano and Rush, 2026). To succeed in this paradigm, agents must not only generate correct code but also maintain cross-turn coherence, manage project memory, and preserve continuous executability across extended sessions.

Current benchmarks fail to capture this dynamic complexity, as summarized in Table 1. Static evaluations (Jimenez et al., 2024; Fu et al., 2025) assess issue resolution but assume a rigidly fixed target objective. Recent interactive benchmarks (Dai et al., 2025; Han et al., 2025; Wu et al., 2025) incorporate user feedback, yet they primarily test *fixed-target error correction* that using feedback solely to guide an agent back to an unchanged initial goal. Consequently, existing frameworks do not evaluate the advanced capabilities required for sustained development: architectural foresight, cross-turn state management, and resilience to requirement drift or revisions within a persistent workspace (Figure 1). To systematically evaluate these capabilities, we introduce **MACBench** (Multi-turn Agentic Coding Benchmark), tailored for long-horizon web application development¹. MACBench comprises 28 rigorously curated repository instances (averaging 25 turns) spanning static and modern frontend architectures. Unlike isolated prompts, instructions are progressively revealed, blending standard feature additions with realistic human interaction patterns like requirement revisions and delayed constraints. This forces agents to continuously adapt their implementations within a persistent workspace with-

[†]Corresponding author

¹<https://github.com/Hypatiaalegra/MACBench>

Benchmark	Agentic Env.	Avg. Turns	Evolving Req.	Intermed. Check	Evaluation Methodology
HumanEval (Chen et al., 2021)	×	1.0	×	×	Test-code (Unit Tests)
SWE-bench (Jimenez et al., 2024)	✓	1.0	×	×	Test-code (Test Patches)
FeedbackEval (Dai et al., 2025)	×	1.0	×	×	Test-code (Execution Feedback)
ConvCodeWorld (Han et al., 2025)	✓	<10.0	×	×	Test-code (Unit Tests)
FronTalk (Wu et al., 2025)	×	10.0	✓	✓	Agent Judge
MACBench (Ours)	✓	25.0	✓	✓	Dual-branch (Test-code + Agent Judge)

Table 1: Comparison between MACBench and existing coding benchmarks. ConvCodeWorld doesn’t have a fix num of average turns tue to its nature. MACBench is uniquely designed for long-horizon agentic development, explicitly evaluating agents on requirement evolution and intermediate viability.

out breaking previously established functionality.

We evaluate 4 contemporary models, assessing both final artifact quality and process-level robustness. Results reveal a severe reliability gap: while agents easily construct initial scaffolds, they struggle to sustain progress over evolving sessions, with the strongest model achieving only a 10.7% strict instance success rate. Furthermore, our intermediate viability checks uncover diminishing and sometimes negative returns from runtime-feedback repair. While bounded feedback aids immediate failure recovery which boosting checklist success by an average of 10.4%, excessive repair attempts frequently degrade final functional outcomes, suggesting they may introduce regressions or side effects. These findings demonstrate that evaluating user-steered development requires measuring not just final correctness, but how agents preserve or erode functional coherence over time.

Our contributions are threefold:

- We introduce **MACBench**, a novel benchmark for long-horizon, multi-turn web development. Unlike existing static or fixed-target benchmarks, MACBench explicitly models continuous requirement evolution and complex cross-turn dependencies within a persistent workspace.
- We propose a dual-branch evaluation framework assessing both process robustness and final artifact quality. This framework uniquely integrates intermediate viability checks, bounded runtime-feedback repair, and comprehensive code/agent-based judging.
- We benchmark 4 state-of-the-art models, revealing a substantial reliability gap in sustaining long-horizon success. Furthermore, we observe diminishing and sometimes negative returns from runtime-feedback repair, showing that while it aids recovery, excessive attempts

may introduce regressions or side effects that degrade final functional outcomes.

2 MACBench

MACBench is a benchmark for evaluating coding agents on long-horizon multi-turn web application development. Each benchmark instance is constructed as a sustained development session in which user requirements unfold over time, all edits accumulate in a persistent workspace, and the agent must preserve consistency and executability while implementing an evolving product. In this section, we describe the two methodological components of the benchmark: (1) the dataset generation pipeline and (2) the evaluation framework.

2.1 Dataset Generation Pipeline

Constructing a benchmark for long-horizon multi-turn coding requires more than collecting isolated prompts. Therefore, we build MACBench through a multi-stage pipeline combining LLM-based synthesis, structured quality control, and expert human curation to encode realistic requirement evolution and preserve cross-turn coherence.

2.1.1 Task Definition and Taxonomy

MACBench focuses on web application development across two complexity levels: static HTML/CSS/JS applications and modern frontends (React/Vue) applications. This scope covers interface logic, state management, routing, and lightweight service interaction while remaining amenable to reproducible automatic evaluation.

Each instance represents a sustained session characterized by two complementary taxonomies (detailed with examples in Appendix A). First, *functional iteration types* define the development action: *New Requirement* introduces functionality, *Modification* alters existing features, *Optimization* improves efficiency or user experience, and *Refactoring* restructures code without changing visible

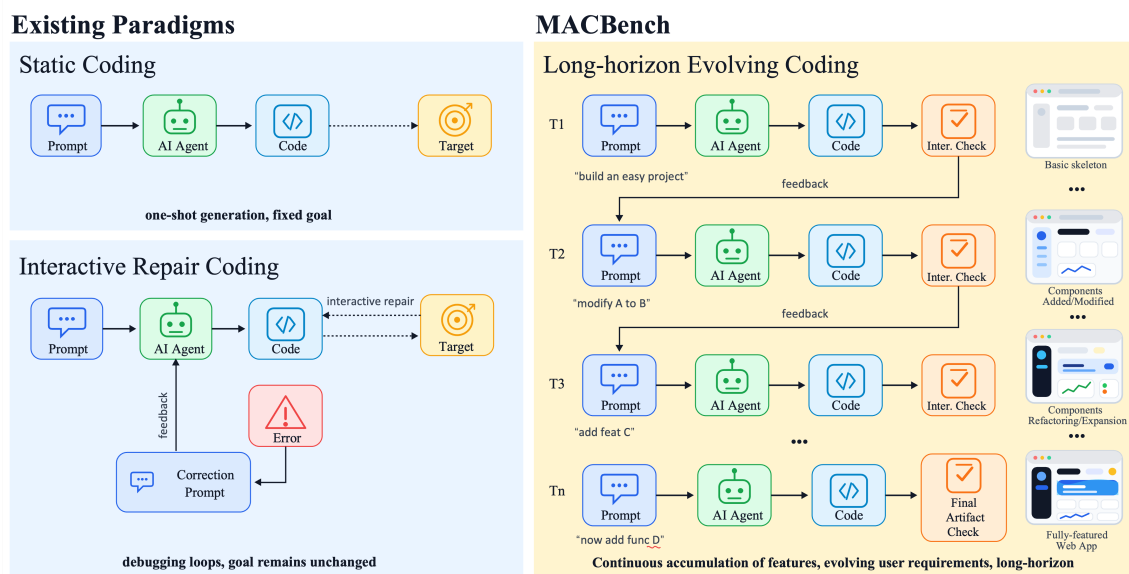


Figure 1: Comparison between existing coding-agent evaluation paradigms and MACBench. Existing paradigms mainly focus on either single-turn fixed-target generation or interactive repair toward an unchanged objective. In contrast, MACBench models long-horizon multi-turn coding, where requirements evolve across turns, runtime feedback is given with a repair mechanism, intermediate viability must be maintained through intermediate viability checks, and final quality is assessed on the completed artifact in a persistent workspace.

behavior. Second, *interaction patterns* capture challenging dialogue structures: *Revision* overturns previous requirements, *Delay* introduces critical constraints post-implementation, *Drift* shifts product direction, and *Explanation* requests task state clarification without coding. Together, these categorization operationalize the core challenge of agentic coding under evolving requirements rather than fixed targets.

2.1.2 Multi-Stage Data Construction

The construction process expands curated seed (description of target repository) into structured multi-turn development sessions. To generate realistic starting points, we use Claude-Opus-4.6 (Anthropic, 2026) to synthesize 100 initial project descriptions representing complex web applications. Each seed defines the overarching target repository goal and specifies a *stack profile*. Human experts evaluate these seed descriptions for realism and implementation feasibility, filtering out unreasonable or overly trivial candidates.

Multi-turn session synthesis. For each target repository, the pipeline generates a sequence of user turns that progressively reveals requirements across approximately 15–35 turns. The opening turn is constrained to establish the project skeleton rather than prematurely solve downstream requirements, and subsequent turns introduce func-

tional growth, revisions, delayed constraints, or explanatory interactions. To preserve multi-turn coherence, generation is coupled with turn-level validation rules that check temporal consistency, prevent invalid backward references, and ensure that later requests remain compatible with the prior history unless an explicit revision is intended. The functional iteration types and interaction-pattern annotations are then assigned to the generated turns to make the developmental structure explicit. The prompts used for the data generation pipeline are demonstrated in Appendix B.

Evaluation target construction. After the interaction sequence is established, the pipeline derives the artifacts needed for evaluation. It generates atomized final checklist items that represent the accumulated requirements of the completed application. To reflect functional importance, each item is assigned a weight of 3 (critical), 2 (important), or 1 (minor), which also undergoes strict quality validation. It also constructs lightweight *intermediate viability checks* for dependency-critical turns, namely intermediate probes that test whether the project remains buildable and whether key capabilities needed by later turns are still preserved. For requirements with executable oracles, the pipeline synthesizes test code conditioned on the repository specification and the inferred contract. Details of

verifier generation and repair are in Appendix B.

2.1.3 Quality Assurance and Human Curation

To prevent hidden inconsistencies and underspecified requirements, we integrate a three-stage quality assurance directly into the construction process. **1) In-the-loop Generation Filtering:** During synthesis, GPT-5.4 (OpenAI, 2026) acts as a quality controller to immediately regenerate components that violate temporal consistency or lack testability. **2) Cross-Model Validation:** Subsequently, Claude Opus 4.6 independently inspects the generated sessions, checklists, and verifiers to reduce model-specific artifacts and ambiguous items. **3) Expert Human Review:** Finally, each repository instance is examined by at least two human annotators who verify realism, cross-turn coherence, and evaluation clarity, revising problematic turns when necessary (details of the human review are provided in Appendix C). This rigorous pipeline ensures the benchmark’s difficulty stems from genuine long-horizon challenges rather than data noise.

2.1.4 Dataset Statistics

The finalized **MACBench** comprises 28 multi-turn repository instances across 8 domains (Figure 2), covering static HTML/CSS/JS and modern frontends (React/Vue). The dataset contains 699 dialogue turns and 2,410 cross-turn dependency edges (defined as directed relationships where a later turn relies on the key outputs of an earlier turn), reflecting the cascading nature of real-world software engineering. Turns introduce new requirements (60.9%), modifications (22.0%), optimizations (10.9%), and refactoring (6.2%), with 30.9% featuring challenging scenarios like revisions or context drift. Performance is rigorously assessed via 956 weighted checklist items using browser-based (79.2%) and test-code-based (20.8%) evaluations. The resulting codebases are non-trivial in scale, averaging 4,421.41 lines of code (up to 14,335) and 17.1 files (up to 47) per repository, where line and file counts exclude `node_modules`, `dist`, `build`, `.git`, cache directories, lock files, and common binary assets. Detailed statistics are summarized in Table 2.

2.2 Evaluation Framework

Agents are evaluated in a realistic coding-agent harness that provides file editing, shell execution, and browser interaction within a persistent workspace. For each benchmark instance, the agent receives the

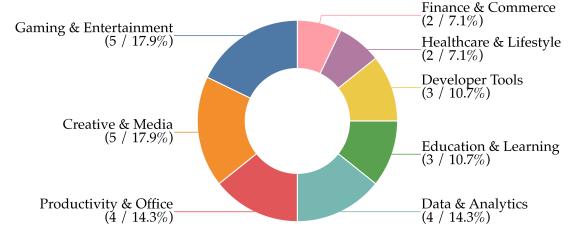


Figure 2: Distribution of the 28 target repositories across 8 application domains in MACBench.

Attributes	Value
<i>Repo-Level</i>	
# Repos	28
# Application domains	8
Avg. turns per repo	25.0
Min / Max turns per repo	16 / 36
Avg. LOC per repo	4,421.41
Max LOC per repo	14,335
Avg. files per repo	17.1
Max files per repo	47
Static HTML/CSS/JS	21.4%
Frontend (React)	39.3%
Frontend (Vue)	39.3%
<i>Turn-Level</i>	
# Turns	699
# Cross-turn dependency edges	2,410
Avg. dependency edges per repo	86.1
Avg. prompt length per turn	189 chars
<i>Evaluation-Level</i>	
# Checklist items	956
Avg. checklist items per repo	34.1
Browser-based evaluation items	79.2%
Test-code-based evaluation items	20.8%
Checklist weight(3/2/1)	58.8% / 35.3% / 6.0%

Table 2: Detailed summary statistics of MACBench.

user turns sequentially and edits the same project state across turns, mirroring sustained development rather than repeated one-shot generation. All turns are conducted within a single, continuous session: the context prefix accumulates across turns and the agent does not restart from scratch at each interaction, so that earlier requirements, edits, and workspace state remain visible throughout the trajectory.

The evaluation framework follows the structure illustrated in Figure 3. After each main turn, the framework runs a lightweight intermediate viability check instead of immediately executing the full final evaluation. These checks are applied at dependency-critical turns, where later requirements depend on key functionality introduced earlier. If a check fails, the framework provides the agent with the observed symptom only (e.g., a build error or failed critical behavior) and allows for a bounded

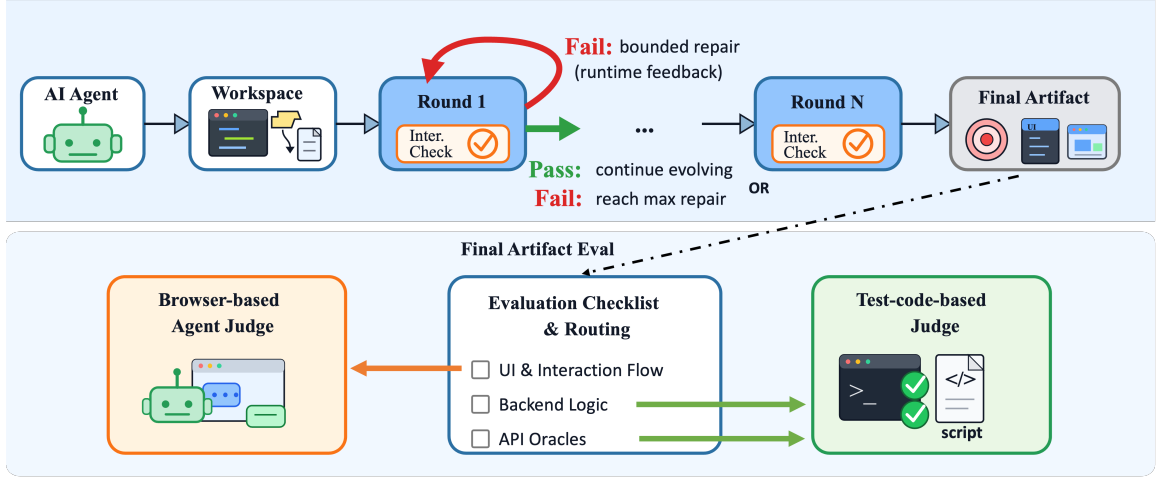


Figure 3: The Evaluation Framework of MACBench. (Top) Workflow & Intermediate Check: During the multi-turn session, dependency-critical turns trigger intermediate viability checks. Failures result in bounded, repair loops with runtime feedback. (Bottom) Final Judging: The completed final artifact undergoes comprehensive evaluation via two complementary branches: (left) a browser-based agent judge for user-facing UI behavior, (right) and a test-code-based judge for executable API logic.

number of repair attempts, capped at two per turn. The repair process is therefore symptom-driven rather than solution-revealing: it gives the agent actionable runtime feedback without disclosing hidden gold behavior or reference implementation. After all main turns and any bounded repairs are complete, the final workspace artifact is submitted to a comprehensive evaluation.

2.2.1 Intermediate Viability Checks and Final Judging

Intermediate viability checking serves two purposes. First, it verifies basic project viability, such as whether the application can still be built or whether the expected entry structure remains valid. Second, for turns that introduce capabilities explicitly required by later turns, it may include a *critical check* that tests whether the newly introduced functionality is sufficient to support subsequent development. This design keeps the long-horizon process executable while avoiding the cost of full end-to-end evaluation after every turn.

Final judging is performed on the completed artifact and combines two complementary modes (Figure 3, Bottom part). For user-facing requirements that depend on rendered behavior, interaction flow, or interface state, we employ a **browser-based agent judge** powered by Claude-Sonnet-4.6 (Anthropic, 2026) equipped with Playwright² Model Context Protocol (MCP) in Claude Code (Anthropic, 2025). This judge inspects the running

application from a user perspective by navigating pages, triggering interactions, and observing outcomes (the detailed prompt configuration is provided in Appendix E). For requirements with executable oracles, we utilize a **test-code-based judge** driven by testing frameworks (i.e., Vitest and Pytest). To ensure the reliability of this automated proxy for assessing complex web development, we conducted a rigorous human agreement analysis. Each automated judgment was cross-validated by two human experts, achieving an overall inter-annotator agreement of 97% exact match. Ultimately, this analysis demonstrated an 85.7% concordance between our dual-branch evaluation framework and the human consensus (details are provided in Appendix D).

2.2.2 Evaluation Metrics

We report both final-quality metrics and process-level robustness metrics. Building upon previous work (Qin et al., 2025), we adapt these metrics to our long-horizon setting. Let N denote the total number of repositories. For target repository i , let C_i denote its set of final checklist items, and $\mathbb{I}(c)$ be an indicator that equals 1 if checklist item $c \in C_i$ passes and 0 otherwise, m_i denote whether target repository i completes all main turns, r_i denote the number of repair attempts used for constructing target repository i , and b_i denote whether the final repository i avoids build-stage failure.

Instance-Level Metrics. We define four evaluation metrics: Main-Turn Completion Rate (MCR),

²<https://github.com/microsoft/playwright-mcp>

Model	Checklist-Level			Instance-Level			
	CSR	Avg.	Adj. Avg.	BSR	MCR	ISR	FPSR
<i>Proprietary Models</i>							
Claude-Opus-4.6	64.6	77.0	40.9	96.4	82.1	10.7	0.0
<i>Open-Source Models</i>							
GLM-5.1	58.6	70.3	37.7	96.4	57.1	3.6	0.0
Kimi-k2.6	52.4	66.3	32.5	100.0	71.4	0.0	0.0
DeepSeek-V4-Pro	70.6	80.7	38.2	100.0	75.0	3.6	0.0

Table 3: Main evaluation results on MACBench. **CSR** Checklist Success Rate serves as the primary metric for evaluating overall functional correctness. **Avg.**: Average weighted score; **Adj. Avg.**: Adjusted average score penalized by repair attempts; **BSR**: Build Success Rate; **MCR**: Main-Turn Completion Rate; **ISR**: Instance Success Rate; **FPSR**: First-Pass Success Rate. All metrics are reported as percentages (%).

Build Success Rate (**BSR**), Instance Success Rate (**ISR**), and First-Pass Success Rate (**FPSR**). Let S_i denote the perfect final checklist performance for instance i , where $S_i = \mathbb{I}(\forall c \in C_i, \mathbb{I}(c) = 1)$. The metrics are defined as:

$$\text{MCR} = \frac{1}{N} \sum_{i=1}^N \mathbb{I}(m_i = 1) \quad (1)$$

$$\text{BSR} = \frac{1}{N} \sum_{i=1}^N \mathbb{I}(b_i = 1) \quad (2)$$

$$\text{ISR} = \frac{1}{N} \sum_{i=1}^N \mathbb{I}(m_i = 1 \wedge S_i = 1) \quad (3)$$

$$\text{FPSR} = \frac{1}{N} \sum_{i=1}^N \mathbb{I}(m_i = 1 \wedge r_i = 0 \wedge S_i = 1) \quad (4)$$

MCR measures whether an agent physically sustains the interaction to the end of the required development sequence. BSR isolates the proportion of final repositories that preserve a runnable artifact at the end of the session. ISR applies a strict success criterion that requires both full main-turn completion and perfect final checklist performance. FPSR is stricter still, requiring such success without any intermediate repair.

Checklist-Level Metrics. To complement the success at the instance-level, we report three aggregate metrics: Checklist Success Rate (**CSR**), Average Score (**Avg.**) and Adjusted Average Score (**Adj. Avg.**). Let w_c denote the weight of checklist item c , and let $V_i = \frac{\sum_{c \in C_i} w_c \mathbb{I}(c)}{\sum_{c \in C_i} w_c}$ denote the weighted completion score for repository i . They are defined as:

$$\text{CSR} = \frac{\sum_{i=1}^N \sum_{c \in C_i} \mathbb{I}(c)}{\sum_{i=1}^N |C_i|} \quad (5)$$

$$\text{Avg.} = \frac{1}{N} \sum_{i=1}^N V_i \quad (6)$$

$$\text{Adj. Avg.} = \frac{1}{N} \sum_{i=1}^N (V_i \times \gamma^{r_i}) \quad (7)$$

where $\gamma = 0.9$ is a discount factor penalizing the number of repair attempts r_i . CSR measures overall unweighted checklist-level functional coverage. Avg. calculates the arithmetic mean of the weighted scores across all repositories. Finally, Adj. Avg. penalizes excessive reliance on intermediate repairs, reflecting the stability and efficiency of the development trajectory. Further details of the discount factor can be referred to Appendix G.

3 Experimental Results and Analysis

3.1 Evaluation Settings

We evaluate 4 state-of-the-art models with thinking mode on using Claude Code (v2.1.14) (Anthropic, 2025) as the standardized agent harness. To faithfully simulate long-horizon development, all multi-turn interactions for a given repository instance are executed within a single, continuous session. When intermediate viability checks fail, the bounded runtime-feedback repair mechanism permits a maximum of two recovery attempts per turn. Further details regarding environment isolation, context compression mechanisms, and execution timeouts are provided in Appendix F.

3.2 Main Results

The Reliability Gap and Silent Failures. Table 3 reveals a severe reliability gap in sustained long-horizon development. Even the most capable model, DeepSeek-V4-Pro, achieves only a 70.6% CSR and a 3.6% ISR. While agents demonstrate a

reasonable ability to sustain the development process—evidenced by MCR ranging from 57.1% to 82.1% and near-perfect BSR, they fundamentally struggle to satisfy all accumulated requirements. This stark contrast exposes a pervasive silent failure phenomenon: agents physically complete multi-turn sessions without fatal codebase corruption, yet continuously accumulate subtle regressions, missed constraints, and logic errors that ultimately compromise the final artifact. Notably, open-source models demonstrate highly competitive capabilities in this complex setting; DeepSeek-V4-Pro outperforms the proprietary Claude-Opus-4.6 across most metrics, while both DeepSeek-V4-Pro and Kimi-k2.6 maintain a flawless 100% BSR.

Performance under Runtime-Feedback Repair.

The weighted scores expose a heavy reliance on iterative runtime-feedback repair. The drop from Avg. to Adj. Avg. is substantial for every model, ranging from 32.6 points (GLM-5.1, 70.3%→37.7%) to 42.5 points (DeepSeek-V4-Pro, 80.7%→38.2%), and FPSR is exactly 0.0 for all four models, underscoring that flawless zero-shot execution over long horizons remains unattainable. While runtime feedback prevents early termination, this reliance manifests as sharply distinct behavioral profiles. DeepSeek-V4-Pro attains the strongest final artifacts (70.6% CSR, 80.7% Avg.) and never breaks the build, but pays the steepest process penalty—retaining only 47.3% of its undiscounted score and completing just 2 of 28 repositories without repair. Claude-Opus-4.6 is strongest once process cost is accounted for (40.9% Adj. Avg.), sustains sessions furthest (82.1% MCR), and needs no repair on 6 of 28 repositories, yet its occasional structurally destructive edits make it one of only two models to break the build (96.4% BSR). GLM-5.1 issues the fewest repair attempts ($\bar{r} = 15.1$) and is discounted least, but this reflects early-terminating sessions rather than efficient editing: its 57.1% MCR falls far below the others’ 71–82%, and it too broke a build. Kimi-k2.6 shows the opposite failure mode—the heaviest repair user ($\bar{r} = 21.8$) yet the weakest final quality (52.4% CSR, 0.0% ISR), while never destabilizing the codebase (100% BSR). These divergent profiles suggest future agents must balance conservative stability with advanced structural reasoning.

3.3 Ablation on Runtime-Feedback Repair

To quantify whether intermediate runtime feedback improves robustness or introduces destabilizing edits, we compare the default setting (bounded symptom-level feedback) against a *without-repair* baseline on a 10-instance subset using three representative models. The 10 instances were randomly sampled from the full set of 28 repository instances, and each configuration was run twice to obtain more stable estimates and reduce the influence of run-to-run variance. As shown in Table 4, enabling repair consistently improves CSR by an average of 10.4%. These gains are inversely correlated with baseline capability; notably, GLM-5.1 with repair nearly matches Claude-Opus-4.6 without repair. Furthermore, MCR improves by 10–20%, indicating that feedback prevents early blockers and sustains forward progress. However, ISR barely changes, revealing that while repair boosts partial functional coverage, it rarely pushes instances to perfect completion. Notably, both settings completed the full sequence of main turns on 100% of instances; the observed CSR and MCR differences therefore cannot be attributed to differences in trajectory length, and instead isolate the effect of runtime repair feedback itself.

Mechanism and Limitations. Runtime feedback primarily excels at catching *build-pass-but-runtime-fail* errors (e.g., browser-incompatible dependencies) that escape static verification, often resolving them in a single attempt. Crucially, early-turn repairs prevent cascading failures in tightly coupled tasks. However, this benefit is not absolute: in 20–30% of instances across models, the without-repair trajectory actually scores higher. In these cases, repair edits resolve the immediate symptom but introduce subtle side effects that compromise downstream functionality. This diminishing-returns pattern justifies our design choice to cap repairs at two attempts per turn, capturing most recoverable failures while limiting collateral regressions.

3.4 Analysis of Residual Failures

To understand the limitations of the default repair mechanism, we manually categorized all failed checklist items (scores below 1.0) across four models on the 10-instance subset, evaluating 150 items in total. As summarized in Table 5, application logic errors involving complex game mechanics, state machines, and multi-component interactions

Model	Setting	Checklist-Level		Instance-Level		
		CSR	Avg.	BSR	MCR	ISR
Claude-Opus-4.6	w/ repair	88.0	90.1	100.0	90.0	10.0
	w/o repair	80.7	82.3	100.0	70.0	10.0
GLM-5.1	w/ repair	79.7	83.1	100.0	90.0	15.0
	w/o repair	70.7	72.8	100.0	80.0	10.0
Kimi-k2.6	w/ repair	77.7	83.6	100.0	90.0	10.0
	w/o repair	62.7	65.0	80.0	80.0	10.0

Table 4: Ablation on runtime-feedback repair over a 10-instance subset. **w/ repair**: the default setting with bounded repair after viability check failures; **w/o repair**: repair loop disabled. All values are percentages (%).

Category	Claude	DS	GLM	Kimi
Application Logic	6	11	14	13
UI & Styling	3	6	4	9
Spec. Compliance	3	3	4	2
Env. & Runtime	4	4	4	0
Numerical Precision	2	2	2	2
Total	18	26	28	26

Table 5: Failed checklist items by category under the w/ repair setting. Claude / DS / GLM / Kimi represent Claude-Opus-4.6 / DeepSeek-V4-Pro / GLM-5.1 / Kimi-k2.6 accordingly.

dominate the distribution, accounting for 33% to 50% of failures depending on the model. For example, a single Pac-Man instance contributed up to 12 failed checks per model because intermediate viability checks ensure page loading but cannot capture deep game logic failure. UI and styling failures rank second, highlighted by Kimi-k2.6’s widespread layout issues and DeepSeek-V4-Pro’s systematic struggles with CSS overflow properties. Finally, numerical precision errors, such as off-by-one timing bugs in an ecological simulation, affected all four models equally. This indicates a universal vulnerability in handling strict quantitative constraints that current self-repair mechanisms fundamentally fail to resolve.

3.5 Case Study

We illustrate the repair dynamics through a 16-turn HTML comment converter task (Claude-Opus-4.6). At Round 1, the agent bundled a library (apg-exp) that compiled successfully but crashed in the browser due to a missing Node.js Buffer API. In the w/ repair setting, a viability check detected this runtime crash, enabling the agent to resolve it instantly (via a precompiled grammar) and achieve a perfect final score (Table 6). Conversely, without repair, this error persisted silently. While the agent successfully implemented decoupled UI

ID	Checklist Item	W	w/	w/o
C1	Two-panel layout	2	1.0	1.0
C2	Copy button feedback	1	1.0	1.0
C3	ABNF panel fold/reset	2	1.0	1.0
C4	Malformed ABNF handling	3	1.0	1.0
C5	Example selector menu	2	1.0	1.0
C6	Comment extraction	3	1.0	0.0
C7	Script/style exemption	3	1.0	0.0
C8	Live DOM preview	2	1.0	0.0
C9	Nesting depth display	3	1.0	0.0
Weighted Score			1.00	0.48

Table 6: Per-check comparison for the HTML comment converter task (Claude-Opus-4.6). **W**: checklist weight. Checks above the dashed line are UI-layout items; those below depend on the core conversion engine.

features (C1–C5), all core conversion functionalities (C6–C9) failed. This stark 0.52 score gap demonstrates how a single uncorrected early-turn failure can silently derail the core of a long-horizon development session.

4 Related Work

While coding agents have advanced toward realistic software engineering, prior works primarily evaluate isolated single-turn repository maintenance, short-horizon conversational assistance, or fixed-target interactive repairs. They fundamentally fail to capture the long-horizon, evolving nature of real-world development. **MACBench** bridges this gap by providing an evaluation framework that explicitly models non-linear requirement evolution within a persistent workspace.

Static and Single-Turn Code Evaluation. Traditional benchmarks assess functional correctness from single-shot prompts (Chen et al., 2021; Austin et al., 2021). Recent advancements elevate this to realistic environments, evaluating repository-level issue resolution (Jimenez et al., 2024) and product-

requirement implementations (Fu et al., 2025). Despite robust execution environments, these frameworks remain fundamentally static: their target objectives are rigidly defined prior to execution.

Multi-Turn and Interactive Evaluation. Recent efforts explore multi-turn conversational workflows (Kim et al., 2026), multimodal UI generation (Wu et al., 2025), and feedback-driven code repair (Dai et al., 2025; Han et al., 2025; Wang et al., 2024). While demonstrating the utility of environmental or user feedback, these studies largely assume stable contexts. They primarily investigate fixed-target correction, where feedback serves exclusively to guide the agent toward an unchanged initial goal, lacking the sustained requirement evolution central to our benchmark.

5 Conclusion

We introduce MACBench, a pioneering benchmark evaluating coding agents on long-horizon frontend web development with continuously evolving text-based requirements within a persistent workspace. Our framework uniquely integrates intermediate viability checks, bounded runtime repairs, and a comprehensive dual-branch final assessment. Benchmarking 4 state-of-the-art models exposes a severe reliability gap in sustained development. Crucially, we observe diminishing and sometimes negative returns from runtime-feedback repair, where excessive repair attempts may degrade final functional outcomes rather than reliably improving them. These findings highlight complex cross-turn application logic as the primary unresolved bottleneck, setting a clear frontier for next-generation coding agents.

Limitations

While MACBench provides a comprehensive framework for evaluating coding agents in long-horizon, multi-turn scenarios, we acknowledge several limitations that offer avenues for future research:

The benchmark is currently restricted to web application development and does not cover broader software engineering domains. While web development offers a rich environment for multi-turn interactions, real-world software engineering encompasses a much wider spectrum. Moreover, MACBench currently conveys requirements purely through natural language, whereas

developers often express intent via screenshots, UI references, or visual revision requests. Future works could expand the benchmark to include multimodal requirement expressions and could expand the benchmark to include backend-heavy architectures, complex database migrations, system-level programming, and data science workflows, testing agents across more diverse persistent workspaces.

The evaluation focuses on functional correctness without explicitly penalizing degrading code maintainability or sub-optimal performance.

The framework rigorously tests intermediate viability and final functional correctness via test-code-based judges and browser-based LLM judges. However, it does not explicitly penalize the accumulation of "spaghetti code" or performance drops resulting from excessive multi-turn patches. Future metrics could incorporate static analysis to evaluate the architectural health of the evolving codebase, together with efficient cross-turn regression measures that track how earlier satisfied requirements are affected by subsequent turns.

Evaluating agents across dozens of turns with intermediate checks and final judging incurs significant computational and financial costs.

End-to-end assessment is inherently resource-intensive: a single task spans 5 to 8 hours, requires provisioning 300 to 500 isolated containers per model, and costs approximately hundreds of dollars in API calls for evaluation alone per model, excluding variable inference expenses. Consequently, our current study is limited to a small selection of state-of-the-art models. Developing lighter-weight proxy metrics that strongly correlate with our rigorous execution-based evaluation could improve the scalability of long-horizon benchmarking. Such scalability would also enable systematic study of emergent behaviors under longer trajectories, such as agents spontaneously repairing prior issues while implementing later requirements, which we observe more often in stronger models.

References

- Anthropic. 2025. [Claude code best practices](#).
- Anthropic. 2026. [Claude sonnet 4.6](#).
- Anthropic. 2026. [Effective harnesses for long-running agents](#).
- Anthropic. 2026. [Introducing claude opus 4.6](#).

- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.
- Federico Cassano and Sasha Rush. 2026. [Training composer for longer horizons](#).
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Dekun Dai, MingWei Liu, Anji Li, Jialun Cao, Yanlin Wang, Chong Wang, Xin Peng, and Zibin Zheng. 2025. Feedbackval: A benchmark for evaluating large language models in feedback-driven code repair tasks. *arXiv preprint arXiv:2504.06939*.
- DeepSeek-AI. 2026. [Deepseek-v4-pro technical report](#).
- Lingyue Fu, Bolun Zhang, Hao Guan, Yaoming Zhu, Lin Qiu, Weiwen Liu, Xuezhi Cao, Xunliang Cai, Weinan Zhang, and Yong Yu. 2025. Automatically benchmarking llm code agents through agent-driven annotation and evaluation. *arXiv preprint arXiv:2510.24358*.
- Hojae Han, Seung-won Hwang, Rajhans Samdani, and Yuxiong He. 2025. Convcodeworld: Benchmarking conversational code generation in reproducible feedback environments. *arXiv preprint arXiv:2502.19852*.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157.
- Myeongsoo Kim, Shweta Garg, Baishakhi Ray, Varun Kumar, and Anoop Deoras. 2026. Codeassistbench (cab): Dataset & benchmarking for multi-turn chat-based code assistance. *Advances in Neural Information Processing Systems*, 38.
- Moonshot AI. 2026. [Kimi k2.6: Advancing open-source coding](#).
- OpenAI. 2026. [Introducing gpt-5.4](#).
- Yanzhao Qin, Tao Zhang, Tao Zhang, Yanjun Shen, Wenjing Luo, sunhaoze, Yan Zhang, Yujing Qiao, weipeng chen, Zenan Zhou, Wentao Zhang, and Bin CUI. 2025. [Sysbench: Can LLMs follow system message?](#) In *The Thirteenth International Conference on Learning Representations*.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangu Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2025. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pages 65882–65919.
- Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2024. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. In *International Conference on Learning Representations*, volume 2024, pages 32593–32627.
- Xueqing Wu, Zihan Xue, Da Yin, Shuyan Zhou, Kai-Wei Chang, Nanyun Peng, and Yeming Wen. 2025. Frontalk: Benchmarking front-end development as conversational code generation with multi-modal feedback. *arXiv preprint arXiv:2601.04203*.
- John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.
- Zhipu AI. 2026. [Glm-5.1](#).

A Detailed Taxonomy with Examples

Table 7 details the two complementary taxonomies used to construct and annotate the multi-turn sessions in MACBench, illustrating how user intent evolves dynamically across a persistent workspace.

B Dataset Generation Prompt Templates

In this section, we detail the core prompt templates used in our multi-stage data construction pipeline (Section 2.1). To facilitate understanding, the original prompts have been condensed to highlight the core instructions, constraints, and output formats while omitting repetitive examples. Table 8 presents the prompts for the data generator, which synthesizes the multi-turn interactions. Table 9 outlines the prompts used to construct both the final evaluation checklists and the intermediate viability checks. Finally, Table 10 details the prompts driving the generation, validation, and self-repair loop for executable test code for test-code-based judge.

C Details of Quality Control and Human Annotations

To guarantee the quality of MACBench and prevent hidden inconsistencies or underspecified requirements, we implement a rigorous quality assurance pipeline combining structured LLM-based filtering and expert human curation.

Taxonomy Category	Definition	Example Dialogue Snippet
<i>Functional Iteration Types</i>		
New Requirement	Introduces entirely new functionality or UI components to the workspace.	“Add a sidebar navigation menu with links to Dashboard, Settings, and Profile.”
Modification	Alters the behavior, styling, or logic of an already existing feature.	“Change the submit button color to blue and make it disabled if the form is empty.”
Optimization	Improves efficiency, accessibility, or UX without changing core functions.	“Add a debounce function to the search bar so it doesn’t fetch data on every keystroke.”
Refactoring	Restructures the underlying code implementation while preserving visible behavior.	“Extract the table component into a separate reusable React component.”
<i>Interaction Patterns</i>		
Revision	Explicitly changes or overturns a requirement established in previous turns.	“Actually, remove the sidebar we just built. Let’s use a top navigation bar instead.”
Delay	Introduces a critical constraint only after an initial implementation is committed.	“Now that the user list is rendering, I forgot to mention it must support pagination.”
Drift	Shifts the product direction or emphasis, testing long-range consistency.	“Instead of just a blog reader, let’s pivot this app to include an e-commerce store section.”
Explanation	Asks the agent to explain or justify its code without requesting new edits.	“Can you explain how the authentication state is currently being managed across routes without changing any code?”

Table 7: Taxonomy of Functional Iterations and Interaction Patterns with concrete examples.

C.1 Automated Quality Control Prompts

During the generation phase, we deploy LLMs as automated quality controllers to inspect the synthesized multi-turn sessions and evaluation artifacts. Table 11 details prompts used for inspecting and repairing the dataset. The inspection strictly enforces principles such as multi-turn consistency (ensuring new rules override old ones correctly) and evaluation flexibility (prioritizing functional correctness over exact wording or layout).

C.2 Expert Human Annotation Protocol

Following the automated filtering, the dataset undergoes a rigorous Human-in-the-Loop (HITL) review. To ensure high inter-annotator agreement, the process begins with an alignment phase: annotators independently review identical trial repository instances, followed by consensus meetings with quality assurance leads to resolve discrepancies and standardize the review criteria.

During the formal annotation phase, expert reviewers evaluate each benchmark instance across three primary dimensions:

1. Instruction Validity and Logic. Annotators read the entire multi-turn sequence to ensure the simulated user behaves rationally. They actively identify and revise four types of prompt defects:

- **Logical Contradictions:** Conflicting require-

ments within or across turns that are not explicitly marked as intentional revisions.

- **Technical Impossibility:** Requests that fundamentally clash with the established architecture (e.g., requesting persistent cross-tab synchronization without allowing local storage).
- **Ambiguity:** Vague requests (e.g., “make the UI look better”) that lack verifiable acceptance criteria.
- **Missing Prerequisites:** Referencing components or concepts that were never introduced in earlier turns (unless intentionally designed as a *Delay* scenario).

2. Taxonomy and Tag Verification. Annotators verify that the functional iteration tags (e.g., *New Requirement*, *Refactor*) and interaction patterns tags (*Revision*, *Delay*, *Drift*, *Explanation*) are accurately applied. For instance, they ensure that a turn tagged as *Revision* genuinely alters a previously established rule, and that *Explanation* turns strictly avoid introducing new coding requirements.

3. Evaluation Target Refinement. Reviewers manually inspect the generated checklist and intermediate viability items to ensure they are perfectly aligned with the prompt history. The refinement process ensures:

- **No Hallucination:** Removing checklist items that evaluate features never requested in the

Multi-Turn Data Generator

You are an expert software product manager simulating a real user guiding an AI coding agent through a long-horizon web development session. Given a seed project description, generate a sequence of user turns.

Core Requirements

1. **Round 0 (Scaffolding)**: Establish the project skeleton. Include basic UI elements but do not implement core business logic yet. Leave room for expansion.
2. **Subsequent Rounds**: Incrementally add 1-2 sub-features per round. Maintain a natural, conversational tone (e.g., "Great, now let's add..."). Describe requirements from a user's perspective (visible UI or observable behavior) without dictating implementation details.
3. **Complexity Dimensions**: Inject constraints (e.g., "max 3 items"), boundary conditions (e.g., empty states), error handling, and cross-component interactions into the requests.
4. **Interaction Scenarios**: Explicitly embed the following challenging scenarios in specific rounds, tagging them in the JSON output:
 - *Revision*: Alter a previously agreed-upon rule without repeating the original text.
 - *Delay*: Intentionally omit critical constraints in an early round and introduce them later.
 - *Drift*: Refer to an existing component using a different colloquial name.
 - *Explanation*: Request clarification or state confirmation without modifying code.
5. **Strict Prohibitions**: Do NOT write specific CSS values, HTML tags, or dictate file structures/frameworks beyond the initial setup.

Inputs

- Project Description: `{description}`
- Stack Hint: `{stack_hint}`
- Target Round Number: `{round_num}`

Output Format

Output a JSON array containing `round_id`, `prompt` (the user instruction), and `scenario_tags`.

Table 8: The condensed prompt template for generating the long-horizon multi-turn interaction sequence.

prompts.

- **Verifiability**: Rewriting checklist descriptions to ensure they describe observable, deterministic behaviors rather than subjective qualities.
- **Mode Correctness**: Ensuring the `judge_mode` is correctly assigned (playwright for UI/DOM interactions and verifier for API/database validation).
- **Dependency Efficiency**: Pruning non-critical intermediate viability checks to maintain evaluation efficiency, ensuring only blocking features are tested mid-session.

Whenever an annotator modifies a test-code-based checklist or intermediate viability item based on test-code, the automated test-code generation pipeline (Table 10) is re-triggered to synthesize the corresponding execution scripts, ensuring the final benchmark remains fully executable.

C.3 Annotator Recruitment and Compensation

To ensure the high quality of the human-in-the-loop curation and align with ethical guidelines for human evaluation, we established strict recruitment criteria for our annotators. We recruited a pool of expert annotators consisting of computer science graduate students. All participants were required to have a minimum of one year of practical expe-

rience in modern web development and possess professional working proficiency in English, ensuring they could accurately evaluate complex, long-horizon software requirements. Based on the the complexity of their assigned repository instances, annotators were compensated between \$150 and \$250 per day for their contributions. The total expenditure for the human annotation and quality assurance process amounted to approximately \$3,000. This compensation structure was designed to provide a fair, competitive rate commensurate with their specialized software engineering expertise.

D Human-Judge Agreement Analysis

Beyond validated automated judges, we also report the quality of tasks generated prior to human correction. In our statistics, the task-level acceptance rate after review is 100.0%, indicating that all generated repository tasks were retained after quality control. However, only 29.7% of the instruction/checklist pairs required no modification. Most of the edits addressed details such as consistency of the instruction-checklist, wording of the checklist, feasibility, and correctness. This confirms that, while the synthetic generation pipeline is effective for coverage and scale, human review remains important for ensuring reliable evaluation quality.

Evaluation Target Construction

Part A: Final Checklist Generation

Given the project description, the synthesized multi-turn instructions, generate the final evaluation checklist items.

- **Granularity:** Each item must evaluate one atomic functional dimension (e.g., UI rendering, API correctness, or database persistence).
- **Weight Assignment (1-3):** Assign weights based on importance. Assign weight 3 to complex end-to-end workflows, cross-round data migration consistency, and regression checks.
- **Verifiability:** Frontend items must be verifiable via visual screenshots and simulated clicks.
- **Conflict Resolution:** If a requirement was altered by a *Revision* or *Delay* tag, the checklist must reflect the **final** agreed-upon behavior.

Part B: Intermediate Check Generation

For each round (except the last), generate a "critical check" that verifies if the core functionality required for subsequent rounds is operational.

- **Focus:** Only include critical features (e.g., "API endpoint returns 200 and correct schema").
- **Mapping:** Explicitly list the `depended_by_rounds` (which future rounds will fail if this check fails).
- **Mode:** Categorize the check as either playwright (UI interaction) or verifier (functional test case).

Inputs (Shared for Part A & B)

- Project Description: `{description}`
 - Multi-turn Instructions: `{rounds_text}`
-

Table 9: The prompt template for deriving final checklists and intermediate viability checks.

Verifier Synthesis and Self-Repair Loop

Stage 1: Generate Verifier Specs

You are a full-stack test engineer. Given the API contract and the required verifier items, generate `vi test` test cases. Each test must be **independent and self-contained** (create its own test data via POST and clean up afterward). Assertions must verify specific response body fields, not just HTTP status codes.

Inputs: `{description}`, `{verifier_items}`

Stage 2: Validate Verifier Specs (Quality Control)

Review the generated test cases against the requirement description. Output an `alignment_score` (1-5) and identify issues across the following dimensions:

- *Functional Mismatch:* Does the test actually verify the described behavior (e.g., data persistence after refresh)?
- *Missing Boundaries:* Are edge cases (empty inputs, long strings, 4xx errors) covered?
- *Weak Assertions:* Does the test lack deep field-level assertions?
- *Cross Dependency:* Does the test rely on global states or other tests?

Inputs: `{description}`, `{relevant_prompts}`, `{specs_to_review}`

Stage 3: Fix Verifier Spec (Self-Repair)

Given a failed test case and its validation feedback (identified issues), repair the test code. Fix the specific problems (e.g., add missing boundary checks, correct API paths) while maintaining the independence and style of the original test. Do not unnecessarily rewrite the entire file.

Inputs: `{description}`, `{relevant_prompts}`, `{specs_with_issues}`

Table 10: The prompts template used for the generation, validation, and self-repair of executable test code.

To validate the reliability of the automated evaluation framework, we conducted a human agreement analysis. We randomly sampled 100 final checklist items from the evaluation trajectories of three diverse models. The samples were evenly distributed, comprising 50 items assessed by the browser-based agent judge and 50 items assessed by the test-code-based judge. Expert human annotators manually inspected the final artifacts to determine whether the implemented functionalities strictly met the given requirements. To ensure rigorous quality control, each item was independently evaluated by at least two annotators achieving an overall inter-annotator agreement of 97% exact match, with the final human judgment deter-

mined by majority vote. The concordance between the automated evaluation outcomes and the human annotations is 85.7%, demonstrating a high overall agreement. This strong consistency confirms that our dual-branch framework serves as a robust and reliable automated proxy for assessing complex, long-horizon web development.

E Evaluation Framework Configuration

For evaluating user-facing functionality, we utilize a browser-based agent judge. Table 12 illustrates its prompt, which strictly enforces a user-centric evaluation perspective. Notably, this judge can assign partial credit (a score of 0.5) for incompletely implemented features. While these partial scores

Automated Quality Control and Self-Repair

Stage 1: Quality Inspection

You are an expert quality assurance agent for multi-turn coding datasets. Review the provided task data against the following core principles and output any identified issues.

Core Principles

- **Clarity & Consistency:** Checklists must only evaluate features explicitly requested in the prompts. Ensure later turns logically align with earlier ones unless an explicit *Revision* is intended.
- **Flexible Evaluation:** Prioritize functional correctness over exact wording (unless explicitly requested). Do not over-constrain visual layouts. Accept reasonable alternative implementations.
- **Testability & Environment:** Ensure requested features (e.g., multi-user roles) have accessible entry points (e.g., mock login) for evaluation. Ensure dependencies align with the stack profile.

Inputs: {description}, {rounds_text}, {checklist_text}, {intermediate_check_text}, {stack_profile_text}

Output Format: Output a JSON array of issues. Each issue must include severity (error/warning), category, description, and actionable fix_options (e.g., remove, modify_item, merge).

Stage 2: Artifact Repair

You are an artifact repair expert. Given the original evaluation artifacts and the issues identified during quality inspection, output the corrected artifacts.

Repair Guidelines

- Fix unclarity by removing hallucinated checklist items or downgrading their weights.
- Fix over-constrained items by rewriting descriptions to focus on semantic functionality rather than exact text/layout.
- Do not modify items that were not flagged with issues.

Inputs: {rounds_text}, {checklist_text / intermediate_check_text}, {issues_text}

Table 11: The prompt template used for automated dataset inspection and self-correction.

contribute proportionally to the weighted Average (Avg.) and Adjusted Average (Adj. Avg.) metrics, they are strictly excluded from the Checklist Success Rate (CSR), which demands full functional completion (i.e., treated as 0 in CSR calculation). Conversely, underlying functional logic is assessed using a test-code-based judge. This verifier executes generated Vitest scripts to validate the programmatic logic behind UI behaviors, providing the deterministic pass/fail signals essential for assessing strict functional correctness.

F Detailed Evaluation Settings and Cost

The models evaluated include GLM-5.1 (Zhipu AI, 2026), Claude-Opus-4.6 (Anthropic, 2026), Kimi-k2.6 (Moonshot AI, 2026) and DeepSeek-V4-Pro (DeepSeek-AI, 2026). To ensure a standardized environment, all inference is conducted using Claude Code (v2.1.14) (Anthropic, 2025) as the underlying agent harness. Crucially, to faithfully simulate long-horizon development, all multi-turn interactions for a given repository instance are executed within a single, continuous session. As the context window naturally expands, the harness automatically triggers a context compression mechanism that invokes the specific model under test to summarize and compact its own memory. For strict reproducibility, both inference and evaluation stages operate in isolated containers. The inference and evaluation containers are provisioned with pre-

configured environments. To accommodate the extensive nature of constructing these repositories, we enforce a maximum execution timeout of 36,000 seconds per repository instance during inference, and 7,200 seconds per checklist item during evaluation. Additionally, when intermediate viability checks fail, the bounded runtime-feedback repair mechanism permits a maximum of two recovery attempts per turn.

Regarding computational scale, completing the full benchmark typically requires approximately 500M–800M tokens during the inference stage, with the exact usage varying across models due to differences in tool-use and code-revision behaviors. The evaluation stage, conducted with Claude-Sonnet-4.6 as the judge, consumes approximately 80M tokens. For models operating with a context window of around 200K tokens, the average number of context compactions per instance is about 2.0 under the *w/ repair* setting and 1.2 under the *w/o repair* setting. We report these statistics to make the computational scale of long-horizon, persistent-workspace evaluation more transparent.

G Sensitivity of the Discount Factor

The Adjusted Average score uses γ to discount repeated runtime-feedback repair attempts, reflecting process inefficiency while preserving the primary role of final functional correctness. To assess robustness, we recompute the Adjusted Average un-

Browser-Based Agent Judge for User-Facing Evaluation

You are an expert software testing agent equipped with Playwright and terminal tools. Your task is to evaluate a locally deployed web project and determine if it satisfies a specific checklist requirement.

Core Guidelines

- **User-Centric Interaction:** You must strictly simulate real user behavior. Only use keyboard and mouse operations (e.g., click, type) that an actual user would perform. Do not use non-standard testing backdoors.
- **Visual Strictness:** If a feature is logically correct in the code but visually broken or inaccessible in the UI, you must consider it a failure. Ensure your screenshots capture the relevant areas completely.
- **Dependency Blocking:** If a prerequisite bug prevents you from reaching or testing the target feature, the current checklist item must be scored as a failure.

Inputs

- Project URL: `{project_url}`
- Workspace Path: `{workspace_path}`
- Task Description: `{task_description}`
- Multi-Turn Prompt History: `{prompts_history}`
- Target Checklist Item: `{checklist_item_description}`

Output Format

Evaluate the item and assign a score based on three tiers:

- **1.0 (Pass):** All functional requirements of the item are met.
- **0.5 (Partial):** Core functionality is implemented, but minor requirements are missed.
- **0.0 (Fail):** Core functionality is missing, critically flawed, or blocked by prior errors.

After detailing your reasoning, your **very last line** must be exactly formatted as follows (without markdown backticks):

VERDICT_JSON:{"score": 1.0}

Table 12: The prompt template configuring the Browser-based Agent Judge to evaluate user-facing checklist items.

Model	Adj. Avg.				Avg.	$\bar{r}$
	$\gamma=.80$	$\gamma=.85$	$\gamma=.90$	$\gamma=.95$		
Claude-Opus-4.6	32.8 ₍₁₎	36.1 ₍₁₎	40.9 ₍₁₎	49.9 ₍₁₎	77.0	18.5
GLM-5.1	29.7 ₍₂₎	33.0 ₍₂₎	37.7 ₍₃₎	46.6 ₍₃₎	70.3	15.1
Kimi-k2.6	25.8 ₍₄₎	28.4 ₍₄₎	32.5 ₍₄₎	40.7 ₍₄₎	66.3	21.8
DeepSeek-V4-Pro	29.1 ₍₃₎	32.8 ₍₃₎	38.2 ₍₂₎	48.9 ₍₂₎	80.7	16.0
Spread (max–min)	7.0	7.7	8.4	9.2	14.4	–

Table 13: Sensitivity of the Adjusted Average score to the discount factor γ . Subscripts denote the rank under each setting. **Avg.** is the undiscounted score (equivalently $\gamma=1$) and $\bar{r}$ is the mean number of repair attempts per repository. The $\gamma=0.90$ column reproduces the Adj. Avg. column of Table 3. All values are percentages (%) except $\bar{r}$.

der $\gamma \in \{0.8, 0.85, 0.9, 0.95\}$. Table 13 reports the results together with the undiscounted score for reference.

The qualitative conclusions of Table 3 are stable over this range: Claude-Opus-4.6 ranks first and Kimi-k2.6 last under every setting. The only movement is a swap between GLM-5.1 and DeepSeek-V4-Pro, which cross between $\gamma = 0.85$ and $\gamma = 0.9$ and are separated by at most 2.3 points anywhere in the grid, so no substantive claim turns on the exact value of γ . Beyond the ordering, γ controls the strength of the process penalty. Because long-horizon sessions accumulate many repair attempts ($\bar{r}$ between 15.1 and 21.8), the discount is highly sensitive in this regime: at $\gamma = 0.8$ a repository with an average repair load retains only $0.8^{18.5} \approx 1.6\%$ of its functional score, compressing the four systems into a 7.0-point band even

though they differ by 14.4 points in final functional quality. In this case, process cost simply overwhelms outcome quality. At $\gamma = 0.95$ the same repository retains $\approx 39\%$, leaving the metric close to the undiscounted Average and weakening its process sensitivity. We adopt the intermediate value $\gamma = 0.9$ (retaining $\approx 14\%$), which registers repair inefficiency: the gap between Avg. and Adj. Avg. ranges from 32.6 to 42.5 points—without erasing the functional signal.

The value of the discounted metric is clearest for the models ranking top two. DeepSeek-V4-Pro achieves the highest undiscounted quality of any agent (80.7 Avg., 70.6 CSR), yet Claude-Opus-4.6 surpasses it on Adj. Avg. at every γ . The reason is that DeepSeek retains the least value under discounting: only 47.3% of its undiscounted score at $\gamma = 0.9$, compared with 53.1% for Claude-Opus-

4.6 and 53.6% for GLM-5.1. This is because its repairs are concentrated on the repositories that ultimately score well. In other words, DeepSeek-V4-Pro attains its quality advantage through a less efficient development trajectory, a distinction the Average alone cannot capture. Another interesting fact is that no repository was completed perfectly without at least one repair ($\text{FPSR} = 0.0$ for every model in Table 3), underscoring how pervasive the repair loop is in this setting.