ViBench: A Benchmark on Vibe Coding

Peter Zhong[*,1,3] Pashootan Vaezipoor[*,2] Fuyang Cui[*,‡] Vaibhav Kumar[1] Azin Asgarian[2]
James Austin[1] Toby Ho[1] Paul Inder[2] Imen Kedir[1] Zhen Li[1] Nick Ondo[†]
Asna Shafiq[2] Ibrahim Sheikh[1] Edouard Sioufi[1] Setareh Soltanieh[2] Ben Wilde[2]
Jacky Zhao[1] Ryan Carelli[†] Heather Miller[3] Michele Catasta[1]
[1]Replit, Foster City, CA, USA [2]Georgian AI Lab, Toronto, ON, Canada [3]Carnegie Mellon University, Pittsburgh, PA, USA
[*]Equal contribution. [†]Work completed while at Replit. [‡]Work completed while at Georgian AI Lab.

Abstract

"Vibe coding"—where users describe applications in natural language and AI agents autonomously handle all implementation—has rapidly emerged as a mainstream development paradigm. Yet existing benchmarks evaluate mainly agent's ability at code completion or fixing bug, not end-to-end application creation from a user's perspective. We introduce ViBench, the first open-source benchmark for evaluating AI agents on realistic, end-to-end web application development. ViBench comprises tasks derived from production traces across 15 applications spanning different domains, covering both zero-to-one creation and feature extension—the two core workflows of vibe coding. Unlike prior benchmarks, tasks are specified entirely through user-facing requirements with no implementation constraints, and evaluation occurs at the abstraction level of an end user interacting with the application. We contribute an adaptive automatic evaluator that uses REPL-based browser automation to verify arbitrary implementations without assuming anything about code structure, achieving 99% step-level agreement with human experts across 1,082 test steps. Evaluating nine models—five closed-source and four open-weight—across 105 artifacts, we find that even the best models are far from reliable: Opus 4.6 and GPT-5.2 lead with only 46% and 42% Pass@1—fully correct artifacts on the first try—while no open-weight model exceeds 12% Pass@1. Feature extension on a stable reference codebase proves easier than zero-to-one creation, but errors compound when models extend their own code: seven of nine models degrade in the Vibe-on-Vibe setting, with complete failure rates rising sharply. Notably, open-weight models that perform competitively on coding benchmarks such as SWE-bench struggle to generalize to end-to-end vibe coding, exhibiting complete failure rates up to 40%—underscoring that end-to-end application development demands capabilities beyond what current coding benchmarks measure. As vibe coding matures into mainstream development practice, ViBench provides the open-source infrastructure needed to measure progress, compare models, and guide practitioners toward reliable AI-assisted development, beyond just a "Vibe Check".

ACM CAIS '26, San Jose, CA, USA

ACM ISBN 979-8-4007-2415-2/26/05
https://doi.org/10.1145/3786335.3813162

CCS Concepts

• **Software and its engineering** → **Software verification and validation**; *Software development techniques*; • **Computing methodologies** → *Natural language processing*.

Keywords

vibe coding, AI agents, software engineering benchmarks, end-to-end evaluation, web applications, LLM evaluation

ACM Reference Format:
Peter Zhong[*,1,3] Pashootan Vaezipoor[*,2] Fuyang Cui[*,‡] Vaibhav Kumar[1] Azin Asgarian[2], James Austin[1] Toby Ho[1] Paul Inder[2] Imen Kedir[1] Zhen Li[1] Nick Ondo[†], Asna Shafiq[2] Ibrahim Sheikh[1] Edouard Sioufi[1] Setareh Soltanieh[2] Ben Wilde[2], Jacky Zhao[1] Ryan Carelli[†] Heather Miller[3] Michele Catasta[1] . 2026. ViBench: A Benchmark on Vibe Coding. In *ACM Conference on AI and Agentic Systems (ACM CAIS '26), May 26–29, 2026, San Jose, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3786335.3813162

1 Introduction

Within the past two years, the advent and rapid improvement of Large Language Models have introduced significant changes in AI-powered programming. The field has seen a transition from simple code completion and edits to agents powered by advanced LLMs that generate entire applications or build features with minimal developer guidance [37]. This autonomous, AI-driven development experience, now known as "vibe coding" [17], has grown into a $36B industry [31] in just one year. Popular services such as Replit, Lovable and V0 promise that a "vibe coder," devoid of any technical experience, can create fully functional applications handling complex, potentially multi-tenant projects—complete with databases, authentication, and other infrastructure—through prompting [20, 30, 36].

But just how good are coding agents, and by extension their underlying models? Early coding benchmarks focused on code completion or repair, often involving a single function. Benchmarks like HumanEval [10], MBPP [4], and ClassEval [13] assess a model's ability to complete short functions or classes, verified by running unit tests. SWE-bench [16] and its intellectual successors (SWE-bench Verified [27], SWE-bench Pro [12]) capture the ability of LLMs to resolve issues in existing codebases. Models must patch bugs in popular libraries like Django and Astropy based on well-defined bug reports that detail reproduction steps and expected behavior, with unit tests determining whether issues were correctly resolved.

However, these existing evaluations are inadequate for vibe-coding tasks. A vibe coder typically delegates technical decisions—from high-level architecture and framework selection to low-level data structures and functions—entirely to the AI agent. Vibe coders care about end-to-end correctness from the perspective of a user interacting with the generated application. Unit tests, which rely on specific programmatic structures, are inappropriate since we cannot ascertain these structures a priori. Automated end-to-end tests, à la SWE-Lancer [26], depend on page structure, layout, or specific DOM elements—decisions also left to the agent. Therefore, evaluation should occur at the abstraction level of an end user interacting with the application, without imposing technical constraints.

Motivated by this gap, we introduce ViBench, a benchmark designed to evaluate agents on realistic, end-to-end web application development. ViBench focuses on two task types: zero-to-one app creation and feature extension. Applications exhibit complexities such as offline operation, multi-tenancy, multiplayer interaction, and real-time updates. Zero-to-one tasks consist of a concise, natural-language Product Requirement Document (PRD) devoid of technical constraints (e.g., specific frameworks or database schemas), leaving implementation choices entirely to the agent. Feature extension tasks provide a reference implementation of a correctly implemented base application along with a feature PRD describing the desired changes.

Each task includes 2 to 4 natural-language test plans describing a list of steps an end user or QA tester would take to verify correct implementation. We introduce an evaluator that uses a notebook-based [14, 18] browser-use [7] harness to assess the correctness of generated applications. To validate this evaluation process, we conducted a preliminary alignment phase in which two human evaluators performed verification manually to label the preliminary set of artifacts generated by the agents against their corresponding test plans. We then compared human judgments against the automatic evaluator, achieving 99% agreement on individual steps.

ViBench focuses on web applications with self-contained functionality, excluding tasks requiring external API integration (with the exception of LLM endpoints), long-horizon verification, or real-time interaction. We evaluate nine state-of-the-art models using OpenHands [37], an LLM-agnostic academic agent system, and analyze results and failure modes in Section 5. As AI-driven development matures, we believe benchmarks must evaluate what users actually care about: working software, not passing unit tests. ViBench and its evaluation harness are publicly available here.

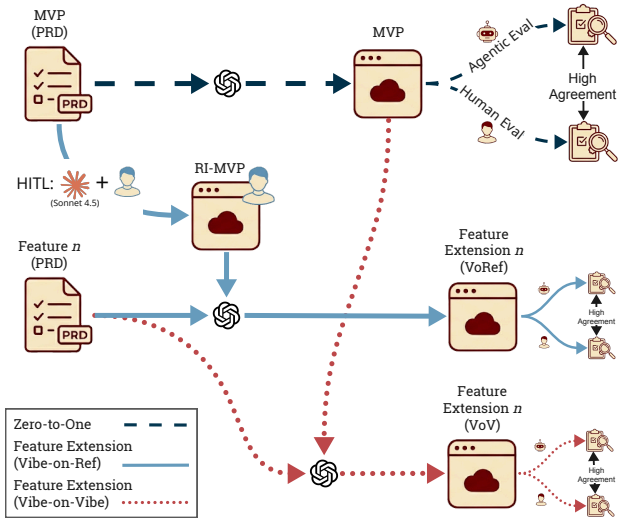


Figure 1: Overview of the ViBench pipeline: Zero-to-one task where a PRD is fed to an LLM agent to implement the MVP *(top)*; Vibe-on-Ref (VoRef) starting from a Reference Implementation (RI-MVP) – created via human-in-the-loop refinement with Claude Sonnet 4.5, the agent has to extend it with new features *(middle)*; Vibe-on-Vibe (VoV) tasks test the endogenous feature extension capabilities of the agent, by asking them to directly vibe-code the feature on top of their zero-to-one generation *(bottom)*. All task types are evaluated by an agentic evaluator and human evaluators, achieving high agreement.

2 Related Work

AI coding evaluation has moved from snippet-level generation to end-to-end agentic software engineering. SWE-bench [16] set the canonical issue-resolution setting, with SWE-bench Verified [27] improving reliability via human validation and SWE-bench-Live [42] reducing data leakage through a real-time pipeline. In parallel, benchmarks broadened environment coverage: RepoBench [19] targets repository-level completion, Web-Bench studies browser workflows, SWE-bench Multimodal [39] adds visual reasoning, and SWE-Lancer [26] frames tasks through an economic lens of freelance engineering work.

As model capability increased, evaluation expanded to efficiency and operational competence. SWE-fficiency [21] measures computational overhead; CodeClash [5] introduces competitive multi-round code evolution; τ-bench [40] tests tool-use, user interaction, and policy adherence; Terminal-Bench [33] evaluates system-level operations; and cline-bench [32] targets high-fidelity tasks grounded in real open-source failure modes.

The latest frontier emphasizes interactive, agentic workflows. FeatBench [9] evaluates feature implementation from abstract requirements in the vibe-coding paradigm, and FrontendBench [43] contributes complex interactive front-end tasks with sandboxed evaluation. "You Don't Know Until You Click" [6] motivates this shift by showing that code inspection alone misses runtime behavior, while browser-use [7] enables autonomous functional interaction for verification. Even with this progress, existing setups still under-capture vibe coding because they do not fully accommodate agent-chosen architectures and therefore remain weak at user-level end-to-end correctness. ViBench addresses this gap by validating arbitrary implementations without technical assumptions and focusing on whether software fulfills user requirements.

Closest to our work are concurrent benchmarks App-Bench [3] and Vibe Code Bench [35] (vals.ai). App-Bench evaluates full-stack build-from-scratch performance with expert manual grading; in contrast, ViBench uses human evaluation only as a sanity check for automated scoring, enabling faster and more scalable benchmarking

while preserving human alignment. Vibe Code Bench also targets build-from-scratch tasks, but does not cover feature extension over a base implementation, and both benchmark and evaluator are closed-source. ViBench instead uses tasks from real production traces and provides an open-source harness spanning both initial app creation and iterative feature development.

3 ViBench

ViBench is designed to emulate realistic vibe coding tasks. We drew tasks from anonymized production traces of Replit [30], a popular AI-powered application development platform. This section describes how we constructed the benchmark, the task setup provided to agents, and the evaluation process.

3.1 Benchmark Construction

Application Selection. We selected applications from Replit production traces, meeting the following criteria: (1) web applications using standard HTML interaction patterns, excluding visual games, native mobile apps, or console applications; (2) applications with interactive logic, excluding static landing pages or content-only sites; (3) applications where correctness is apparent, excluding those without clear success criteria such as AI strategy advisors; (4) applications not fundamentally dependent on external services, excluding those tightly coupled to third-party APIs for their core functionality; and (5) applications that were the subject of active development with agent activity spanning more than three days. These criteria ensure that tasks are amenable to automatic verification and sufficiently interesting to assess.

Feature Extraction. From each application, our team—all with software engineering experience—identified a set of core features constituting the base Minimum Viable Product (MVP), as well as a set of *independent* features that could serve as candidates for feature extensions. Feature extensions vary in complexity but must add a substantial new pattern of interaction on top of the base MVP. This process was informed by the vibe coder's interactions with Replit as they incrementally added features to their application, though some separation decisions were made by our team when the historical interaction trace did not yield a clean feature decomposition. An application was included in ViBench only if the base MVP was nontrivial to implement and at least two nontrivial feature extensions could be extracted.

We kept features as faithful as possible to the original user requirements, with targeted modifications to ensure applications are anonymized, deterministic (e.g., specifying dice rolls through environment variables), and independent of external providers (e.g., replacing OAuth with username/password authentication).

PRD Creation. From the base MVP descriptions and feature extension specs, our team manually created PRDs in a uniform format focusing exclusively on business-facing content: user flows, roles, business rules, and user-visible (and therefore testable) states. All technical details—data models, APIs, schemas, libraries—were omitted. Any static assets important to the task, such as images or JSON files, are also curated. Some example PRDs can be found in Appendix A; a description of all applications can be found in Appendix G.

Reference Implementation (RI).. Since feature extensions build upon a stable foundation, we seek to isolate the model's performance on new features from any errors inherited from the base code. To achieve this, we create a separate *Reference Implementation (RI)* of each MVP PRD. We developed each RI using Claude Sonnet 4.5 with human-in-the-loop correction, iteratively refining the output until it fully satisfied the PRD. Therefore, by construction, all base-level tests pass on the RI, ensuring that subsequent failures reflect errors in the feature extension rather than legacy bugs.

Test Plan Creation. For base MVPs, test plans cover all substantial interaction patterns specified in the PRD. For feature extensions, we create two types of test plans: one regression test and one or more feature-specific tests. The regression test navigates through any changes introduced by the feature (e.g., new authentication flows or reorganized layouts) but verifies that base MVP functionality remains intact. Feature-specific tests cover the new interactions introduced by the feature PRD. All test plans are expressed purely in terms of behaviors specified in the PRD—with specific values, actions, and validation criteria—but make no assumptions about implementation details. To the greatest extent possible, we ensured that any reasonable implementation of the PRD could pass these tests. To maximize reproducibility, we focus exclusively on high-level functional correctness; visual design and performance are out of scope.

Test Plan Format. Some test plans assume a particular application state to provide more focused coverage—for example, verifying appointment rescheduling requires that a past appointment already exists. These test plans include a seeding section that specifies the relevant data that must exist in the database before execution. Each test plan comprises sequentially executed steps, where each step groups closely associated actions and verifications. Some verifications are marked as non-fatal, meaning the test can continue even if that verification fails; this allows for more granular understanding of where and how an implementation breaks. Each step has a point value, denoting its relative importance to the rest of the test plan. A test plan's *graded score* is the fraction of total points earned, yielding a continuous measure of partial correctness; an artifact achieves *Pass@1* only when every associated test plan scores 100%, indicating no bugs or missing functionality.

Quality Assurance. At least two team members reviewed each PRD and test plan to ensure they were clearly defined, unambiguous, and implementation-agnostic.

Benchmark Summary. The final benchmark comprises 15 applications across domains including messaging, multiplayer turn-based games, and social media. These yield 15 zero-to-one tasks and 45 feature extensions; each feature extension is evaluated in both VoRef and VoV modes, producing 105 total artifacts (15 + 45 + 45). They are evaluated by 194 test plans containing 1,407 total steps. Depending on the complexity of each step, it could contain up to 7 distinct actions and verifications grouped together. ViBench could be easily extended by adding new and possibly more complex application ideas as long as these applications are amenable to browser-based testing. While the evaluation is performed using

OpenHands, the tasks and test plans are designed to be general and can be adapted to other specialized harnesses.

Limitations. The benchmark enforces constraints to ensure testability and focus on core functionality. External API integration complicates setup and verification, so we generally exclude it—with one exception: given the prevalence of AI-integrated software and the statelessness of the API, we permit OpenAI endpoints. We also exclude features requiring long-horizon verification (e.g., waiting 30 minutes for a notification), fast real-time interaction (e.g., action games), and non-web applications. Finally, VIBENCH evaluates only functional correctness, not visual aesthetics or design quality. While visual polish is important to end users, it is inherently subjective and difficult to evaluate automatically; we leave aesthetic evaluation to future work.

3.2 Task Setup

VIBENCH comprises two task types (Figure 1). In *zero-to-one* tasks, the agent receives only the PRD of the MVP and must generate the full application from scratch. In *feature extension* tasks, the agent receives the feature PRD, the original MVP PRD, and a base codebase that it must modify to implement the feature. We classify feature extension tasks into categories based on the origin of the base codebase: in *Vibe-on-Ref (VoRef)* tasks, the feature extensions are built atop of the known-correct RI-MVP. Alternatively, the model can build the feature on the MVP that it built earlier in its own zero-to-one task. This second setup assesses the model's ability to continue building on work that may itself be incorrect, and reveals how much performance degrades when compounding on its own errors. We refer this second build as *Vibe-on-Vibe (VoV)*.

Environment. All tasks are conducted in a standardized Debian Docker environment with internet connectivity. Python and Node.js are pre-installed, though the agent may install additional packages. While tasks are agnostic to specific data management strategies, we provide a PostgreSQL instance accessible via environment variable to ensure uniformity in building and evaluation. We chose PostgreSQL because most popular vibe coding platforms, including Lovable and Replit, provide PostgreSQL to their users.

Deployment. The agent builds or augments the application in a designated folder and must produce two scripts: a setup script that installs dependencies, and a start script that launches the server. Evaluation runs these scripts in a fresh instance of the same Debian Docker environment to ensure no OS-level inconsistencies, but without any state carried over from development—a capable agent must not only ensure the application works during local development, but have the foresight to produce scripts that start cleanly from a cold state.

3.3 Evaluation

Seeding. Some test plans require the application database to be in a particular state before testing. However, because agents have complete control over database schema, we cannot write seeding SQL scripts a priori. Instead, a seeding agent—powered by Claude Sonnet 4.5[1]—inspects the application code and schema to produce a seeding script. The seeding agent receives the seeding section of the test plan, the application codebase, and the setup and start scripts; it then generates a script that calls the setup script and injects the required data. All date-based elements are specified relative to the current time to ensure the seeding script remains valid across runs. In manual inspection, we found that when the seeding agent fails to produce a working script, the cause is typically a failure in the application itself (e.g., server won't start or schema is malformed). For applications that could be seeded, all inspected scripts contained correct seeding logic. Once seeding (if applicable) and setup complete, the start script launches the server, which serves as the target for either a human evaluator or the automatic evaluator described in Section 4.

[1]We suspect other frontier models of a similar weight class could also serve as the seeding agent.

4 Automatic Evaluator

A benchmark relying solely on human evaluators cannot scale. VIBENCH therefore contributes an automatic evaluator that uses an LLM-powered agent to execute test plans against generated applications. The central challenge is that, unlike traditional QA automation, the evaluator cannot assume anything about the application's structure—the agent under evaluation has complete control over UI layout, DOM structure, and interaction patterns. Our evaluator must therefore be adaptive: capable of inspecting the generated codebase, understanding the application's specific implementation, and dynamically constructing browser interactions to verify correctness.

4.1 Approach

We implement the evaluator as a tool-augmented LLM agent built on the OpenHands [37] harness. Given a test plan and the generated application artifact, the agent reads the source code to understand the application's structure, then iteratively executes browser actions to carry out each test step.

The key design choice is a *REPL*[2]*-based execution model* for browser automation. Rather than generating a complete test script upfront—which would require perfect prediction of selectors, timing, and interaction sequences—the agent operates in a notebook-style environment where it can execute Playwright [24] commands incrementally. Each execution returns the resulting page state (screenshot and ARIA snapshot [23]), allowing the agent to observe outcomes before deciding on subsequent actions. Variables and browser contexts persist across executions, enabling complex multi-step interactions while remaining responsive to the actual application behavior.

This incremental approach mirrors how a human QA tester would operate: inspect the page, perform an action, verify the result, and proceed—adapting to unexpected layouts or interaction patterns as they arise.

4.2 Tools

The evaluation agent has access to three categories of tools:

[2]Read-Eval-Print Loop

Standard tools. File reading/writing, bash command execution, and task management—standard OpenHands tools provided without modification. These allow the agent to inspect the application codebase and understand its structure before interacting with it.

Playwright Script Execution. The agent writes and executes Playwright scripts in a persistent notebook environment. Variables created in one execution (browser contexts, pages, stored values) remain available in subsequent executions. Each execution returns console logs, an ARIA accessibility snapshot [23], and a screenshot of the active page.

Page State Inspection. For test plans requiring verification across multiple browser contexts—such as checking whether a message sent by one user appears for another—the agent can inspect the state of any open page without switching the active context.

Together, these tools enable the agent to handle the full range of ViBench test plans, including multi-tenancy verification (multiple simultaneous user sessions), offline simulation, and file upload/-download interactions.

4.3 Design Rationale

We considered several alternative approaches to automatic evaluation before arriving at the REPL-based design.

Pre-generated test scripts. Traditional Playwright testing involves writing complete test scripts that execute end-to-end. However, generating such scripts requires accurate prediction of element selectors, page structure, and timing—all of which vary across agent submissions. In practice, even expert engineers struggle to write robust scripts for unfamiliar applications without iterative debugging. A pre-generated script that works for one submission may fail on another that implements the same PRD with different UI choices. The REPL approach sidesteps this brittleness by allowing the evaluator to adapt its actions based on observed page state.

Atomic browser-action tools. Browser automation frameworks like Playwright MCP [25] and Chrome DevTools MCP [11] expose individual actions (click, type, scroll) as separate tools. While intuitive, this approach has two limitations. First, each tool requires extensive documentation to specify its parameters and usage patterns, increasing prompt complexity. Second, the action space is constrained to the predefined tool set, making complex operations like multi-browser coordination, network interception, or offline simulation difficult to express. By contrast, Playwright's full API is well-represented in LLM training data, and the agent can compose arbitrary Playwright code to handle edge cases without tool-specific scaffolding.

Computer-use models. Vision-based computer-use agents [1, 29] operate by taking screenshots and predicting coordinate-based actions (e.g., click at position X, Y). While more general than browser-specific approaches, they are suboptimal for web application testing. Form filling requires many screenshot-action cycles. Verification relies solely on visual inspection, which cannot access non-visible state and may be unreliable depending on VLM architecture limitations[3] [34]. Complex operations like simulating offline mode or managing multiple browser profiles require navigating system UI rather than programmatic control. The Playwright-based approach provides direct programmatic access to these capabilities.

4.4 Human Alignment

To validate that the automatic evaluator produces judgments comparable to human experts, we conducted an alignment study comparing automatic and manual evaluation on 10 out of the 15 ViBench tasks.

Methodology. We selected 10 out of the 15 ViBench applications and randomly assigned each to one of four models: Claude Sonnet 4.5, Gemini 3 Pro, GPT-5[4], and Qwen Coder 480B[5]. We included multiple models to ensure diversity in the artifacts assessed, as different models may produce implementations with varying UI patterns, code structures, and failure modes. Each model completed both zero-to-one and feature extension tasks for its assigned applications.

Two human evaluators with software engineering experience independently evaluated all test plans against the generated artifacts, marking each step as pass or fail. The evaluators then met to resolve disagreements, reaching full consensus on all steps. The automatic evaluator subsequently assessed the same artifacts using identical seeding scripts.

Results. Across 136 test plans comprising 1,082 steps, human evaluators marked 59% of steps as passing and 41% as failing. The automatic evaluator agreed with human consensus on all but 10 steps—achieving *99.07% step-level agreement* and *93.4% test-plan-level agreement.*[6] Of the disagreements, six were false positives (the automatic evaluator passed steps that humans failed) and two were false negatives. The disagreements fell into the following categories:

- **Specification ambiguity:** In one case, the test plan required clicking an edit button, but the application automatically transitioned to edit mode upon navigation. Human evaluators accepted this as a reasonable implementation; the automatic evaluator marked it as a failure due to the absent button.
- **Scoring judgment:** One test plan included a non-fatal step verifying that an input field received focus automatically. Both evaluators detected that the field was not focused, but the automatic evaluator judged the failure too minor to penalize, while human evaluators assigned zero points per the test plan specification.
- **Browser interaction differences:** In some cases, the automatic evaluator could extract information from the DOM that was not visually apparent to human evaluators, or could input text faster than a human before a page refresh.
- **Overeager evaluator:** In early iterations, the evaluator occasionally attempted to fix bugs or refresh the page to continue the test plan rather than assessing the application as-is. Prompt refinement virtually eliminated this behavior.

[3]Vision-language models typically downsample images to a fixed resolution and tokenize them into patches (e.g., 16×16 pixels), with each patch projected into a single embedding. Fine-grained UI details—such as checkbox states, badge counts, or subtle icon changes—can be lost in this process, whereas ARIA snapshots and DOM inspection provide unambiguous access to such state.

[4]The alignment study was conducted before the final benchmark evaluation and used GPT-5, an earlier model in the GPT-5 family. Final results (Section 5) use GPT-5.2.

[5]Our final results excluded Qwen Coder due to API access issues.

[6]One test plan had two steps with disagreements, accounting for the difference between step-level and test-plan-level agreement rates.

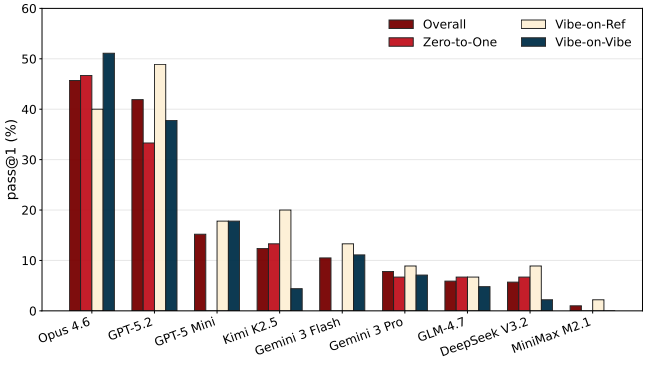


Figure 2: ViBench Pass@1 by Model and Evaluation Slice

While the slight bias toward false positives means benchmark scores may marginally overstate model performance, the overall error rate remains below 0.6% of steps.

5 Results

5.1 Experimental Setup

Models. We ran ViBench on five frontier closed-source models: GPT-5.2, GPT-5-mini, Gemini 3 Pro, Gemini 3 Flash, and Opus 4.6; plus four open-weight models: DeepSeek-V3.2, GLM-4.7, MiniMax-M2.1, and Kimi-K2.5. For open-weight models, we solely employ Fireworks as the API provider. Detailed model API names are listed in Table 2. We enabled the highest reasoning capabilities for all models.

Agent Harness. All agents were instantiated with OpenHands [37] using standard file editing, terminal execution, and task management tools. For the GPT-5 family, we used ApplyPatch [28] as the file editing tool, while all other models used StrReplace [2], following provider recommendations. LLM performance is sensitive to diff format choice, and providers train models on specific formats for reliability [15]. Agents were limited to 3 hours and 300 total steps. Every 10 steps, the agent was reminded of the remaining budget. Across 105 artifacts, DeepSeek-V3.2 hit the step limit 4 times and MiniMax-M2.1 hit it once; all other models reported completion before reaching the limit. Therefore, the step limit is not a bottleneck for any model.

5.2 Model Performance

Scoring and costs. Table 1 reports results across four evaluation slices: zero-to-one (15), Vibe-on-Ref (45), Vibe-on-Vibe (45), and Overall (105). To contextualize ViBench against classical repository-level coding benchmarks, gray parenthesized tags next to model names report SWE-bench Verified rank as (relative / absolute): relative is the rank after restricting to models that overlap with our ViBench comparison set, while absolute is the model's rank on the full SWE-bench Verified leaderboard.

An artifact's **Graded Score** is the cumulative points earned across all its test plans divided by the total possible points. The table reports **Pass@1** (fraction of artifacts that achieve a perfect graded score on the first try) and **Complete Failure Rate** (fraction that receive a graded score of 0, e.g., failed builds or non-runnable artifacts). **Build $** is the average cost of producing an artifact with the coding agent. Separately, the average cost of evaluating a single artifact with the automatic evaluator is $4.89; evaluation may terminate early if a fatal step fails. Zero-to-one evaluates building the full application from scratch, while Vibe-on-Ref and Vibe-on-Vibe evaluate feature extensions built on top of a reference implementation or the model's own zero-to-one output, respectively.

Key results. Across the full benchmark under the standard OpenHands harness, Opus 4.6 and GPT-5.2 form a clear top tier: Opus 4.6 achieves the best overall reliability (high Pass@1 and low complete failure rate), while GPT-5.2 is close behind at lower build

Table 1: Results across four evaluation slices: Overall, Zero-to-One, Vibe-on-Ref, and Vibe-on-Vibe. Parenthesized counts next to slice names indicate the number of artifacts in that slice. Pass@1 reports first-try full passes (100% graded score), while Complete Failure Rate reports complete failures (0% graded score). The gray parenthesized tags indicate SWE-bench Verified rank as (relative/absolute). Alternate-harness rows use the same underlying model with a different coding agent harness. In Vibe-on-Vibe, the parenthesized arrow next to each model name is the Pass@1 delta vs. Vibe-on-Ref.

	Model	Pass@1	Complete Failure Rate	Graded Score (Avg.)	Build $ (Avg.)
Overall (105 = 45 + 45 + 15)	Opus 4.6 + Claude Code	54 (51.4%)	0 (0.0%)	89.0	4.91
	Opus 4.6 (1/4)	48 (45.7%)	3 (2.9%)	81.0	3.79
	GPT-5.2 (2/9)	44 (41.9%)	11 (10.5%)	74.0	1.75
	GPT-5.2 + Codex	41 (39.0%)	14 (13.3%)	72.0	1.37
	GPT-5-mini (-)	16 (15.2%)	12 (11.4%)	56.6	0.26
	Kimi-K2.5 (4/18)	13 (12.4%)	10 (9.5%)	57.8	2.30
	Gemini 3 Flash (3/13)	11 (10.5%)	38 (36.2%)	37.7	0.51
	Gemini 3 Pro (5/24)	8 (7.8%)	33 (32.4%)	38.8	3.58
	GLM-4.7 (6/30)	6 (5.9%)	25 (24.5%)	36.3	2.49
	DeepSeek-V3.2 (7/36)	6 (5.7%)	42 (40.0%)	29.1	4.88
	MiniMax-M2.1 (-)	1 (1.0%)	36 (35.3%)	30.4	1.29
Zero-to-One (15)	Opus 4.6 + Claude Code	8 (53.3%)	0 (0.0%)	93.3	4.01
	Opus 4.6	7 (46.7%)	0 (0.0%)	86.5	3.02
	GPT-5.2	5 (33.3%)	3 (20.0%)	64.5	1.88
	GPT-5.2 + Codex	5 (33.3%)	3 (20.0%)	60.7	1.14
	Kimi-K2.5	2 (13.3%)	0 (0.0%)	67.4	1.88
	Gemini 3 Pro	1 (6.7%)	6 (40.0%)	33.1	2.93
	GLM-4.7	1 (6.7%)	7 (46.7%)	20.7	2.18
	DeepSeek-V3.2	1 (6.7%)	11 (73.3%)	11.5	4.92
	GPT-5-mini	0 (0.0%)	5 (33.3%)	39.7	0.17
	Gemini 3 Flash	0 (0.0%)	7 (46.7%)	21.6	0.54
	MiniMax-M2.1	0 (0.0%)	9 (60.0%)	12.3	1.06
Vibe-on-Ref (45)	Opus 4.6 + Claude Code	21 (46.7%)	0 (0.0%)	86.4	5.32
	GPT-5.2	22 (48.9%)	1 (2.2%)	83.9	1.94
	Opus 4.6	18 (40.0%)	0 (0.0%)	82.2	4.54
	GPT-5.2 + Codex	22 (48.9%)	5 (11.1%)	77.7	1.46
	Kimi-K2.5	9 (20.0%)	3 (6.7%)	63.1	2.49
	GPT-5-mini	8 (17.8%)	2 (4.4%)	61.7	0.31
	Gemini 3 Flash	6 (13.3%)	9 (20.0%)	54.2	0.60
	DeepSeek-V3.2	4 (8.9%)	5 (11.1%)	54.2	4.68
	Gemini 3 Pro	4 (8.9%)	8 (17.8%)	44.0	4.49
	GLM-4.7	3 (6.7%)	5 (11.1%)	51.8	2.77
	MiniMax-M2.1	1 (2.2%)	5 (11.1%)	49.7	1.21
Vibe-on-Vibe (45)	Opus 4.6 + Claude Code (4↑)	25 (55.6%)	0 (0.0%)	90.3	4.79
	Opus 4.6 (5↑)	23 (51.1%)	3 (6.7%)	78.0	3.30
	GPT-5.2 (5↓)	17 (37.8%)	7 (15.6%)	67.1	1.52
	GPT-5.2 + Codex (8↓)	14 (31.1%)	6 (13.3%)	70.1	1.36
	GPT-5-mini (0)	8 (17.8%)	5 (11.1%)	57.2	0.23
	Gemini 3 Flash (1↓)	5 (11.1%)	22 (48.9%)	26.5	0.41
	Gemini 3 Pro (1↓)	3 (7.1%)	19 (45.2%)	35.2	2.85
	GLM-4.7 (1↓)	2 (4.8%)	13 (31.0%)	25.4	2.28
	Kimi-K2.5 (7↓)	2 (4.4%)	7 (15.6%)	49.3	2.26
	DeepSeek-V3.2 (3↓)	1 (2.2%)	26 (57.8%)	9.9	5.06
	MiniMax-M2.1 (1↓)	0 (0.0%)	22 (52.4%)	16.1	1.45

cost. Alternate-harness runs show that harness choice can materially affect reliability: Opus 4.6 with Claude Code achieves 54/105 full-pass artifacts and no complete failures. Zero-to-one tasks are the most fragile slice, and GPT-5.2 in particular drops more sharply on zero-to-one than on extensions, driven by runtime failures rather than near-misses. Gemini and open-weight models are more often derailed by end-to-end breakage, as reflected in high complete failure rates; among open-weight models, Kimi-K2.5 is the strongest performer on our benchmark. By contrast, feature extension on a stable reference implementation (Vibe-on-Ref) is substantially easier for most models, with higher Pass@1 and fewer complete failures. We unpack these outcomes through trajectory-level and qualitative failure-mode analyses in the following subsections. Per-app graded scores are presented in Table 3.

5.3 Vibe-on-Ref vs. Vibe-on-Vibe

In practice, developers often extend the MVP that the agent just generated, rather than restarting from a clean codebase. Vibe-on-Vibe directly measures this compounding setting by comparing extension performance on a model's own MVP against extension performance on a stable reference implementation.

Overall, *performance decreases when extending self-generated MVPs*. Among the nine standard OpenHands runs, relative to Vibe-on-Ref, *seven of nine models lose full-pass artifacts* in Vibe-on-Vibe, consistent with error compounding when the MVP is imperfect. For example, GPT-5.2 drops from 22/45 passes (48.9%) on RI to 17/45 (37.8%) on its own MVP, with average graded score declining from 83.9 to 67.1 and the complete failure rate rising from 2.2% to 15.6%.

The degradation is more pronounced for Gemini and open-weight models, where we observe both fewer perfect artifacts and frequent transitions into complete failures, indicating that earlier build and integration issues often propagate into later feature work. For instance, Gemini 3 Flash sees its complete failure rate increase from 20.0% to 48.9% and average graded score fall from 54.2 to 26.5, while DeepSeek-V3.2 drops from 4/45 passes (8.9%) to 1/45 (2.2%) with average graded score collapsing from 54.2 to 9.9.

The *sole exception among the standard OpenHands runs* is Opus 4.6, which improves from 18/45 passes (40.0%) on RI to 23/45 (51.1%) on Vibe-on-Vibe (+5). We attribute this to two potential factors: (i) highest-quality self-generated MVPs, which reduce the amount of "technical debt" that extensions must absorb, and (ii) stronger distributional self-consistency, where the MVP and extensions share implicit conventions (APIs, component boundaries, naming), lowering integration friction relative to extending an external RI.

Overall, Vibe-on-Vibe exposes a central deployment risk: *errors compound across builds*, and Vibe-on-Ref can overestimate practical reliability by masking the downstream cost of imperfect first-pass codebases.

5.4 Trajectory Level Failure Modes

Complete Failures. An interesting observation is the prevalence of artifacts that were broken end-to-end. Often this results from setup failures due to early stopping prior to task completion, missing dependencies, compilation failures, incorrect configuration, or major UI bugs. This happens more frequently for open-weight and Gemini models, where the agent typically performs shallow self-validation before returning the artifact. Many of these failures would be caught by basic smoke checks, such as rerunning the setup and start scripts and issuing a few simple `curl` requests. On the contrary, GPT and Claude families both leverage the terminal tool to perform these validations and therefore have much fewer complete failures. While for fairness all models are given the same prompt and the same tools,[7] this may reflect the effectiveness of tuning prompts and tools to the idiosyncratic nature of each model.

Building Failures. The primary build-time failure we observe arises when the agent escalates from targeted recovery attempts to indiscriminate process termination (e.g., killing all Python processes). This broad action can terminate the agent harness itself, converting an otherwise recoverable issue into a hard failure. We observe this behavior across the Gemini family and most open-weight models; within our open-weight set, Kimi-K2.5 is the only model that does not exhibit it. Importantly, this is an instruction-following failure: our system prompts explicitly forbid blanket termination, yet after repeated unsuccessful attempts to stop a misbehaving server, agents often revert to "kill-everything" strategies.

Seeding Failures. Seeding failures occur when the environment setup script or the application itself is implemented incorrectly, preventing end-to-end execution. This failure mode is highly prevalent among open-weight models (e.g., DeepSeek-V3.2, MiniMax-M2.1) and the Gemini family, while remaining comparatively rare for the GPT and Claude families. The primary bottlenecks are missing backend dependencies (e.g., `sqlalchemy`, `PyYAML`, `alembic`) and absent frontend build artifacts (e.g., missing `vite` binaries or `dist` directories), which frequently derail database and server initialization. Other common factors include fundamental syntax errors and missing execution permissions for setup scripts. We want to note that while the agents possess the tools to verify package installations, environment setups, and syntax issues on their own during build-time, the persistence of these basic errors suggests that their self-validation processes remain largely superficial. Open-weight models are particularly fragile on this failure case, partially suggesting weaker generalization to different execution environments/agent harness.

Qualitative Evaluation. To facilitate future agent and model development, we conducted a comprehensive qualitative analysis of the apps built by the agents and their trajectories. Terminal Bench [33] derived broad failure categorization based on the Multi-Agent System Taxonomy (MAST) [8] and conducted trajectory analysis with Docent [22]. Inspired by prior works [38], we adopt a simpler approach and built a more tailored end-to-end pipeline for qualitative evaluation. Our analysis uses a simple three-step pipeline. First, we use a ReAct agent (GPT-5.2; [41]) to inspect each artifact, the build agent's trajectory, and evaluator results, then write a summary explaining why the score is imperfect. To capture command-level issues, we also run a pre-defined analytics scripts that flag brittle tool-use patterns (e.g., invalid command usage or environment mistakes). We build this summary set by sampling 74 imperfect artifacts, balanced across models (8 per model when possible). An example qualitative summary is presented in

[7] With the exception of file edit tools, which follow the provider's recommended patch format.

Appendix F.3. Second, we prompt GPT-5.2 to propose a taxonomy that covers the observed failures, with five broad categories and fewer than fifteen fine-grained categories (Appendix F.1). Third, we prompt GPT-5-mini to extract failure modes from each summary and map each one to the taxonomy, which we then aggregate into Figure 3.

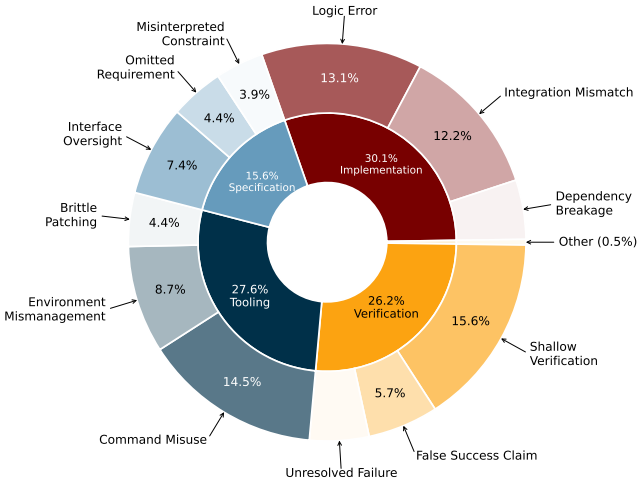


Figure 3: Failure Mode Taxonomy: Distribution of failure modes across all models and artifacts.

Figure 3 summarizes 435 failure modes identified across 74 analyzed artifacts. At the broad level, failures split across **Implementation** (getting the implementation right; 30.1%), **Tooling** (using commands and environments reliably; 27.6%), and **Verification** (checking end-to-end workflows; 26.2%), with a smaller but meaningful share in **Specification** (meeting the stated requirements; 15.6%) and almost none in **Other** (external defects; 0.5%). The dominant bottlenecks are therefore mundane: agents write plausible code, but often ship without fully wiring components together or confirming that a user can complete the intended workflow. At the fine-grained level (Section F.1), the most frequent diagnoses are *Shallow Verification* (15.6%), *Command Misuse* (14.5%), *Logic Error* (13.1%), and *Integration Mismatch* (12.2%).

Comparing model families (Figure 6), we see consistent signatures. GPT-5.2 is the most verification-heavy (31.9%), driven by *Shallow Verification* (23.4%): it often checks parts (e.g., a server starts) but does not exercise full user flows. The Gemini family shows a larger tooling footprint (29.0–32.5%), with many errors from *Command Misuse* and *Environment Mismanagement*, which prevents otherwise-correct code from ever being tested. Opus 4.6 is more balanced across buckets, but shows a higher rate of **Specification** issues (20.4%), often *Interface Oversight* where small UX details block progress. Among open-weight models, implementation failures are more prominent (e.g., DeepSeek-V3.2 at 35.2%), with recurring *Integration Mismatch* and *Dependency Breakage* as components and libraries are stitched across the stack. They also show more early-stage fragility: tooling and verification issues often prevent a clean end-to-end run. As a result, small setup mistakes can quickly turn into complete failures.

6 Discussion

Scope Limitations. ViBench currently evaluates only web applications without external API dependencies. This scope excludes two common vibe coding scenarios.

First, mobile applications. While desktop and terminal applications are rare among vibe coders, mobile apps are increasingly common and introduce unique challenges around touch interactions, device simulation, and platform-specific behavior. Extending ViBench to mobile is a natural direction for future work.

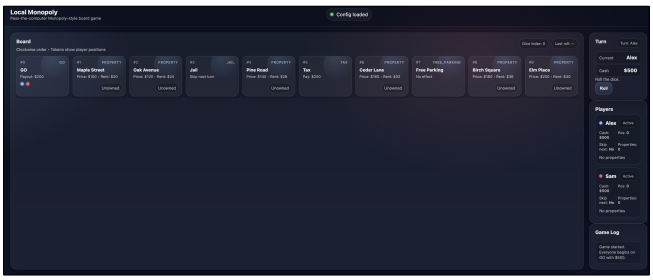
Second, many real-world applications integrate external APIs (payment processors, social login, third-party data services). We exclude these due to the complexity of maintaining API keys, handling side effects, and adapting to API evolution. A potential middle ground is to provide mock API servers with stable interfaces and expose their documentation to the coding agent, allowing evaluation of API integration patterns without external dependencies.

Evaluation Cost. The current evaluation approach, while achieving high alignment with human judgments, incurs non-trivial costs ($1.49 per test plan and $4.89 per artifact on average). For large-scale evaluations, this cost may become prohibitive. We see opportunities for cost reduction through intelligent routing of test plans to appropriately-sized models, caching of common interaction patterns, and more selective screenshot capture.

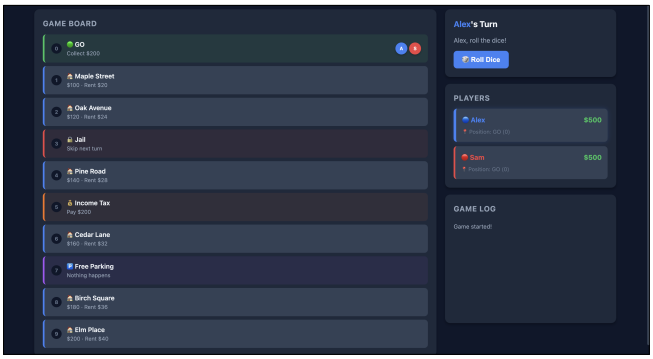
Living Benchmark. ViBench is intended to be a live benchmark that can grow with both agent capabilities and real user demands. The task format is designed to make new PRDs, feature extensions, and test plans straightforward to add without changing the evaluation harness. As an initial post-acceptance expansion, we report five additional PRDs in Appendix E; these tasks are relatively easy compared with the main benchmark, and future releases will add more diverse and challenging applications with broader model coverage.

Visual Evaluation. ViBench deliberately excludes visual aesthetics from evaluation due to their subjective nature. While some broken UI artifacts—such as covered controls or overly small text—could be assessed more objectively, visual evaluation as a general problem remains a challenge. However, for many vibe coding use cases, visual polish is important. Figure 4 illustrates this challenge: both implementations of the Monopoly application pass the same functional tests, but their visual presentations differ substantially. Which design a user prefers depends on context and personal taste. Future work could explore an automated visual evaluation framework assessed against a larger set of human-labeled visual artifacts to emulate human preferences.

Partial Success. ViBench treats Pass@1 as its primary metric. Graded score is useful diagnostically but misleading as a headline number: under the standard OpenHands harness, Opus 4.6 averages 81% graded score yet only 46% Pass@1, meaning most of its artifacts exhibit at least one defect. This gap is partly structural. Our evaluator executes test steps sequentially, accumulating points until a fatal failure halts the test plan. An artifact can therefore earn a high graded score simply because the early, often simpler steps pass before a critical defect surfaces later. Figure 5 illustrates this: a 93%-scoring Pilot's Logbook is missing required filtering on



(a) GPT-5.2



(b) Opus 4.6

Figure 4: Two implementations of the Monopoly application. Both are functionally correct and received full scores, yet exhibit completely different visual styles — GPT-5.2 produced a less readable horizontal layout while Opus 4.6 created a vertical and more polished design with great readability.

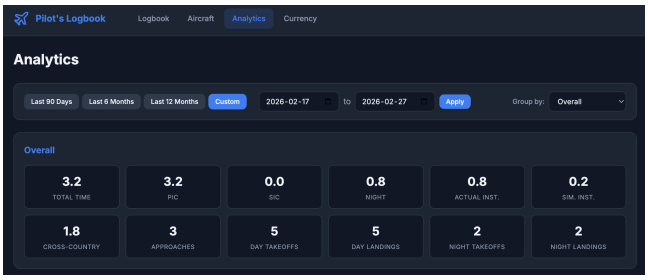


Figure 5: A screenshot of the Analytics page of the Pilot's Logbook MVP built by Opus 4.6. The page has incomplete filtering options, which is explicitly required by the PRD. Despite this missing functionality, the artifact can receive a high graded score.

its Analytics page—functionality that the evaluator only reaches after the earlier steps already contribute most of the points. From a vibe coder's perspective, this artifact is simply unfinished, consistent with the specification-adherence issues noted in Section 5.4. Pass@1 avoids this inflation by treating correctness as binary—the application either meets its full specification or it does not—aligning with how users actually judge the output. That said, graded score remains a useful secondary signal: it distinguishes near-complete artifacts from total failures and helps identify which parts of a specification models consistently struggle with.

7 Conclusion

ViBench fills a critical gap in AI coding evaluation. While existing benchmarks assess code completion or issue resolution, vibe coders need to know: can this agent build my application? ViBench answers this question directly by evaluating end-to-end application development—from zero-to-one creation through iterative feature extension—using tasks derived from real production traces and specified entirely through user-facing requirements. Our adaptive evaluator verifies arbitrary implementations without assuming anything about code structure, achieving reliable agreement with human experts while enabling fully automated, reproducible evaluation.

The benchmark reveals meaningful differentiation across nine models that simpler evaluations would miss. Under the standard OpenHands harness, Opus 4.6 and GPT-5.2 form a clear top tier, yet even they achieve only 46% and 42% Pass@1—meaning fewer than half of their artifacts are fully correct on the first try. Alternate-harness runs suggest that harness design can materially affect reliability: Opus 4.6 with Claude Code reaches 54/105 full-pass artifacts with no complete failures. Open-weight models fare worse still, with no model exceeding 12% Pass@1 despite competitive performance on benchmarks like SWE-bench. Feature extension on a stable reference codebase proves substantially easier than zero-to-one creation in graded score (44–84% vs. 12–87%), though Pass@1 remains below 50% on both slices under OpenHands, with Opus 4.6 leading zero-to-one (47% Pass@1) and GPT-5.2 leading feature extension (49% Pass@1). Critically, errors compound in the most realistic setting: when models extend their own MVPs (Vibe-on-Vibe), Pass@1 drops for seven of nine standard OpenHands runs—GPT-5.2 falls from 49% to 38%, and several open-weight models collapse below 5%—while complete failure rates more than double for most models.

Across all standard OpenHands runs, fully correct artifacts remain rare even when graded scores are high—the best model averages 81% graded score yet achieves only 46% Pass@1—revealing a persistent gap between near-functional prototypes and production-ready applications. Closing this gap will likely require improvements in attention to detail, faithful specification adherence, more rigorous self-verification before delivery, and better alignment between models and the agent harnesses that operationalize them.

As vibe coding matures into mainstream development practice, ViBench provides the open-source infrastructure needed to measure progress, compare models, and guide practitioners toward reliable AI-assisted development.

References

[1] Anthropic. 2024. Introducing computer use, a new Claude 3.5 Sonnet, and Claude 3.5 Haiku. https://www.anthropic.com/news/3-5-models-and-computer-use.
[2] Anthropic. 2025. Text Editor Tool. https://platform.claude.com/docs/en/agents-and-tools/tool-use/text-editor-tool.
[3] AppBench Team. 2025. AppBench: Evaluating AI Agents on Application Development. https://appbench.ai/.
[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021). https://arxiv.org/abs/2108.07732

[5] Anonymous Authors. 2025. CodeClash: Benchmarking Goal-Oriented Software Engineering. *arXiv preprint arXiv:2511.00839* (2025).

[6] Yutong Bian, Xianhao Lin, Yupeng Xie, Tianyang Liu, Mingchen Zhuge, Siyuan Lu, Haoming Tang, Jinlin Wang, Jiayi Zhang, Jiaqi Chen, Xiangru Tang, Yongxin Ni, Sirui Hong, and Chenglin Wu. 2025. You Don't Know Until You Click: Automated GUI Testing for Production-Ready Software Evaluation. *arXiv preprint arXiv:2508.14104* (2025). https://arxiv.org/abs/2508.14104

[7] Browser-Use. 2025. Make websites accessible for AI agents. https://github.com/browser-use/browser-use.

[8] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Why Do Multi-Agent LLM Systems Fail? arXiv:2503.13657 [cs.AI] https://arxiv.org/abs/2503.13657

[9] Haorui Chen, Chengze Li, and Jia Li. 2025. FeatBench: Evaluating Coding Agents on Feature Implementation for Vibe Coding. *arXiv preprint arXiv:2509.22237* (2025). https://arxiv.org/abs/2509.22237

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021). https://github.com/openai/human-eval

[11] Chrome DevTools. 2024. Chrome DevTools MCP. https://github.com/ChromeDevTools/chrome-devtools-mcp.

[12] Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. 2025. SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks? *arXiv preprint arXiv:2509.16941* (2025). https://arxiv.org/abs/2509.16941

[13] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-Level Code Generation. *arXiv preprint arXiv:2308.01861* (2023). https://arxiv.org/abs/2308.01861

[14] Google. 2017. Google Colaboratory. https://colab.research.google.com/.

[15] Fabian Hertwig. 2025. Code Surgery: How AI Assistants Make Precise Edits to Your Files. https://fabianhertwig.com/blog/coding-assistants-file-edits/.

[16] 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 *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*. https://www.swebench.com/

[17] Andrej Karpathy. 2025. There's a new kind of coding I call "vibe coding". X (formerly Twitter). https://x.com/karpathy/status/1886192184808149383 Accessed: 2025-02-01.

[18] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. 87–90. https://jupyter.org/

[19] Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. *arXiv preprint arXiv:2306.03091* (2023).

[20] Lovable. 2025. AI-powered full-stack app builder. https://lovable.dev/.

[21] Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, Vijay Janapa Reddi, and Parthasarathy Ranganathan. 2025. SWE-fficiency: Can Language Models Optimize Real-World Repositories on Real Workloads? *arXiv preprint arXiv:2511.06090* (2025). https://arxiv.org/abs/2511.06090

[22] Kevin Meng, Vincent Huang, Jacob Steinhardt, and Sarah Schwettmann. 2025. Introducing Docent. https://transluce.org/introducing-docent.

[23] Microsoft. 2024. ARIA Snapshots – Playwright Documentation. https://playwright.dev/docs/aria-snapshots.

[24] Microsoft. 2024. Playwright – Fast and reliable end-to-end testing for modern web apps. https://playwright.dev/.

[25] Microsoft. 2024. Playwright MCP. https://github.com/microsoft/playwright-mcp.

[26] Samuel Miserendino et al. 2025. SWE-Lancer: Can Frontier LLMs Earn $1 Million from Real-World Freelance Software Engineering? OpenAI. https://openai.com/index/swe-lancer/

[27] OpenAI. 2024. Introducing SWE-bench Verified. OpenAI Blog. https://openai.com/index/introducing-swe-bench-verified/

[28] OpenAI. 2025. Apply Patch Tool. https://platform.openai.com/docs/guides/tools-apply-patch.

[29] OpenAI. 2025. Computer-Using Agent. https://openai.com/index/computer-using-agent/.

[30] Replit. 2025. Introducing Replit Agent. https://blog.replit.com/introducing-replit-agent.

[31] Jakob Steinschaden. 2025. Vibe coding startups: valuations grew by 350% in one year, huge revenue multiples. Trending Topics. https://www.trendingtopics.eu/vibe-coding-startups-valuations-grew-by-350-in-one-year-huge-revenue-multiples/ Accessed: 2025-09-11.

[32] The Cline Team. 2025. cline-bench: A Real-World Open Source Benchmark for Agentic Coding. https://cline.bot/blog/cline-bench-initiative.

[33] The Terminal-Bench Team. 2025. Terminal-Bench: A Benchmark for AI Agents in Terminal Environments. https://github.com/laude-institute/terminal-bench.

[34] Shengbang Tong, Zhuang Liu, Yuexiang Zhai, Yi Ma, Yann LeCun, and Saining Xie. 2024. Eyes Wide Shut? Exploring the Visual Shortcomings of Multimodal LLMs. *arXiv preprint arXiv:2401.06209* (2024). https://arxiv.org/abs/2401.06209

[35] Hung Tran, Langston Nashold, Rayan Krishnan, Antoine Bigeard, and Alex Gu. 2025. Vibe Code Bench: Can AI Models Build Web Applications from Scratch? Vals AI. https://vals.ai/benchmarks/vibe-code Accessed: 2025-11-21.

[36] Vercel. 2024. v0 – AI-powered UI generation. https://v0.dev/.

[37] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. *arXiv preprint arXiv:2407.16741* (2024). https://github.com/All-Hands-AI/OpenHands

[38] Blair Yang, Fuyang Cui, Keiran Paster, Jimmy Ba, Pashootan Vaezipoor, Silviu Pitis, and Michael R. Zhang. 2024. Report Cards: Qualitative Evaluation of Language Models Using Natural Language Summaries. arXiv:2409.00844 [cs.LG] https://arxiv.org/abs/2409.00844

[39] John Yang et al. 2025. SWE-bench Multimodal: Do AI Systems Generalize to Visual Software Domains?. In *Proceedings of the 13th International Conference on Learning Representations (ICLR)*.

[40] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. *arXiv preprint arXiv:2406.12045* (2024). https://arxiv.org/abs/2406.12045

[41] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL] https://arxiv.org/abs/2210.03629

[42] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. 2025. SWE-bench Goes Live! *arXiv preprint arXiv:2505.23419* (2025). https://arxiv.org/abs/2505.23419

[43] Hongda Zhu, Yiwen Zhang, Bing Zhao, Jingzhe Ding, Siyao Liu, Tong Liu, Dandan Wang, Yanan Liu, and Zhaojian Li. 2025. FrontendBench: A Benchmark for Evaluating LLMs on Front-End Development via Automatic Evaluation. *arXiv preprint arXiv:2506.13832* (2025). https://arxiv.org/abs/2506.13832

Appendix Table of Contents

A Example PRD

The following is an example PRD for the Family Social Network base MVP:

```
Family Social Platform (MVP)

A social platform where users form confirmed family
    relationships (Spouse, Parent, Child, Sibling)
    and share posts visible only to direct
    relationships -- never friends-of-friends.

CORE CONSTRAINTS
- No transitive visibility: only Active direct
    relationships can see your posts/birth date
- One relationship per user pair (Pending or Active
    blocks new requests; to change type, end then re-
    request)
- Mutual acceptance required for relationships;
    requests never expire

1) ACCOUNTS & PROFILES

Registration & Login
- Sign up with email (unique, case-insensitive) and
    password (min 8 characters)
- After signup, user must complete profile (Username +
     Display Name required) before accessing any
    features
- Login errors must not reveal whether email or
    password was incorrect
- Logout ends the session and returns to
    unauthenticated state
- Delete Account: removes all user data; all
    relationships (Active and Past) involving this
    user show "Deleted User" as the other party in
    other users' Past sections

Profile Fields
- Username: required, 3 - 30 chars, alphanumeric +
    underscores only, unique (case-insensitive)
- Display Name: required
- Bio, Profile Photo, Birth Date: optional; Birth Date
     cannot be in the future

Profile Visibility
- Public (anyone can see): Display Name, Username,
    Profile Photo, Bio, Active relationships list
- Relationship-only (requires Active relationship):
    Birth Date, posts list

2) FAMILY RELATIONSHIPS

Relationship Types
When sending a request, select how you view this
    person:
- "Parent" = this person is my parent
- "Child" = this person is my child
- "Spouse" = this person is my spouse
- "Sibling" = this person is my sibling

Reciprocal Types (after acceptance)
- Spouse  <->  Spouse, Sibling <-> Sibling
- Parent  <->  Child (sender sees "Parent", recipient
    sees "Child")
- Child  <->  Parent (sender sees "Child", recipient
    sees "Parent")

Request Flow
- Send: appears in your Outgoing Requests and their
    Incoming Requests (recipient sees what type
    sender selected, not the reciprocal type yet)
- Accept: relationship becomes Active
- Decline: moves to Past for both parties
- Cancel (sender only): moves to Past for both parties
- Cannot request relationship with yourself

Ending Relationships
- Either party can end an Active relationship
- Both immediately lose access to each other's posts
    and birth date
- Relationship moves to Past for both

Relationships Page Sections
- Incoming Requests, Outgoing Requests, Active, Past
- Active relationships displayed publicly on profiles,
     grouped by type, alphabetical by username
- Past relationships visible only to the owner; shows:
     other user's name, type, status (Declined/
    Canceled/Ended), date moved to Past
- After any status (Declined/Canceled/Ended), either
    party may send a new request

3) POSTS

Creation & Content
- Post requires at least one of: non-empty caption,
    image(s)
- Up to 4 images per post

Editing & Deletion
- After publish: only caption can be edited; images
    are immutable
- Author can delete their own posts (removed
    everywhere)

Visibility
- Visible to author (always) and users with Active
    relationship at viewing time
- Relationship ends -> immediate loss of access to
    each other's posts

[... continued ...]
```

B Example Test Plan

The following is an excerpt from a test plan for the Family Social Network:

```
<test_plan>

<purpose>
Validate admin waitlist management: viewing waitlist
    with ordering/positions, promotion behavior when
    no spots exist, admin-initiated cancellation
    triggering auto-promotion, manual promotion after
     capacity increase, removing members from a
    waitlist, and ensuring a removed member can re-
    join.
</purpose>

<seeding_and_precondition>
**Event** ("Admin Managed Screening"):
- Date/time: 12 days from now at 8:00 PM
- Location: "Cinema Room, Brooklyn"
- Category: screening
- Capacity: 2
- Price: $20.00

**Members** (profiles completed):
```

```
- Eve: username "adminwlmemberE", password "AdminWLE
     !2025", name "Eve AdminTest", email "adminwl.
     e@example.com", role "Filmmaker" -  RSVP'd to
     event
- Frank: username "adminwlmemberF", password "AdminWLF
     !2025", name "Frank AdminTest", email "adminwl.
     f@example.com", role "Writer" -  RSVP'd to event

Event is full (2 RSVPs = capacity).

**Waitlist** (in join order):
- Grace: username "adminwlmemberG", password "AdminWLG
     !2025", name "Grace AdminTest", email "adminwl.
     g@example.com", role "Photographer" - position 1
- Henry: username "adminwlmemberH", password "AdminWLH
     !2025", name "Henry AdminTest", email "adminwl.
     h@example.com", role "Designer" -  position 2
- Iris: username "adminwlmemberI", password "AdminWLI
     !2025", name "Iris AdminTest", email "adminwl.
     i@example.com", role "Visual Artist" - position 3
</seeding_and_precondition>

<steps>
<step>

<name>
test_admin_views_waitlist_with_positions
</name>

In a new browser context `context_admin`, open the app
      and log in as admin ("core_admin" / "CoreAdmin
     !2025").

Navigate to an admin view for managing "Admin Managed
     Screening" and open its waitlist. (This may be
     inside the event details page, a "Waitlist" tab,
     or a global waitlist management screen filtered
     to this event.)

Verify the waitlist shows the three seeded members in
     first-come-first-served order:
- Grace AdminTest (adminwl.g@example.com) before Henry
      AdminTest before Iris AdminTest

Commentary:
- The PRD requires admin waitlist visibility including
      position/order and join time, but does not
     prescribe the exact table layout.
- The UI may show an explicit numeric "position" field
      (1/2/3) or may simply order the list top-to-
     bottom. Either is acceptable as long as the
     ordering matches the seeded join order.

Verify:
- (non-fatal) If explicit positions are shown, they
     correspond to Grace=1, Henry=2, Iris=3.
- (non-fatal) Each entry shows a waitlist join time (
     format is implementation-defined).

<points>8</points>
</step>

[... additional steps ...]
</steps>

<full_points>66</full_points>

</test_plan>
```

C Model Alias to API Name Mapping

Table 2 maps paper-facing model aliases to their concrete API model names on LiteLLM.

Table 2: Model alias to API model name mapping on LiteLLM

Alias	API model name
GPT-5.2	`openai/gpt-5.2-2025-12-11`
GPT-5-mini	`openai/gpt-5-mini-2025-08-07`
Gemini 3 Pro	`gemini/gemini-3-pro-preview`
Gemini 3 Flash	`gemini/gemini-3-flash-preview`
Opus 4.6	`anthropic/claude-opus-4-6`
DeepSeek-V3.2	`fireworks_ai/deepseek-v3p2`
GLM-4.7	`fireworks_ai/glm-4p7`
MiniMax-M2.1	`fireworks_ai/minimax-m2p1`
Kimi-K2.5	`fireworks_ai/kimi-k2p5`

D Per-App Graded Scores

Table 3 reports per-app graded scores aggregated over all artifacts for each app.

E Additional PRD Results

We additionally evaluate five applications added after the primary benchmark run:

- `book_journey`
- `fleet_management`
- `language_learning`
- `logistics`
- `resume_builder`

You can see the results in Table 4. These results are reported separately from the headline VIBENCH aggregate. Metrics follow Table 1; each score is artifact-averaged, Pass@1 reports artifacts with a perfect graded score, and Complete Failure Rate reports artifacts with a 0% graded score. Per-app graded scores for these additional applications is reported in Table 5.

These additional PRDs illustrate that VIBENCH is intended to be a live and easily extensible benchmark: new applications, features, and test plans can be added as agent capabilities and user needs evolve. The five PRDs reported here are relatively easy compared with the main benchmark, and we commit to adding more diverse and challenging applications in future releases. Due to the short interval between paper acceptance and the camera-ready deadline, we were unable to obtain enough inference quota to run the open-weight models on this additional set; therefore, we report only the closed-source model results.

F Qualitative Evaluation

F.1 Failure Taxonomy

Implementation

- Logic Error: Basic programming mistakes such as syntax errors, unhandled nulls, bad conditional checks, or incorrect algorithmic choices.
- Integration Mismatch: Broken contracts between components, such as the frontend targeting missing DOM IDs or the backend rejecting unexpected payload shapes.
- Dependency Breakage: Unpinned, missing, or conflicting library dependencies that result in runtime crashes or broken builds.

Specification

Table 3: Per-app graded scores (%), aggregated over each app's zero-to-one build and feature-extension artifacts.

App	Opus 4.6	GPT-5.2	GPT-5-mini	Kimi-K2.5	Gemini 3 Flash	Gemini 3 Pro	GLM-4.7	DeepSeek-V3.2	MiniMax-M2.1
collabrative_kaban	65.0	63.4	34.8	46.6	31.2	22.3	26.7	21.5	21.0
creative_community	96.1	83.9	62.8	87.1	28.3	67.7	26.0	32.5	47.8
energy_audit	89.6	93.8	74.1	77.9	41.9	39.7	61.2	38.6	55.2
family_friendly_venue	86.1	86.4	28.6	34.8	48.6	24.6	26.4	38.5	27.2
family_social	83.1	46.5	72.7	83.6	21.6	32.1	52.9	24.1	22.3
furniture_freight	88.0	61.0	84.1	64.9	91.8	23.0	39.6	43.4	41.6
hvac	88.1	92.8	51.7	56.1	9.7	47.2	61.8	32.1	33.2
mafia	88.3	74.5	77.5	63.1	21.6	32.2	23.1	27.2	33.2
market_place	98.1	99.1	84.0	70.1	48.7	67.2	80.6	72.2	36.9
monopoly	84.2	87.7	20.1	38.3	47.7	35.7	26.2	14.7	25.1
online_whiteboard	74.4	17.1	46.2	69.5	69.8	43.5	28.6	18.2	49.2
pilot_logbook	62.0	61.4	40.5	45.2	11.0	16.7	13.6	1.2	4.7
slack	97.9	100.0	83.6	35.5	36.8	66.7	12.3	26.0	43.2
srm	53.2	66.8	41.7	21.6	21.2	10.0	27.5	13.9	2.9
wedding	39.1	69.3	32.7	66.0	24.1	31.5	38.3	14.1	17.4

Table 4: Compact results on five additional PRDs. Overall includes 5 MVP artifacts and 28 feature-extension artifacts per model, except Gemini 3 Pro, which has 26 feature-extension artifacts. Z2O, VoRef, and VoV columns report average graded score.

Model	Pass@1	CFR	Overall	Z2O	VoRef	VoV
Opus 4.6	21 (63.6%)	3 (9.1%)	84.5	76.5	89.7	82.1
Gemini 3 Flash	16 (48.5%)	1 (3.0%)	79.2	70.8	86.9	74.4
GPT-5.2	19 (57.6%)	4 (12.1%)	77.8	60.0	97.0	64.9
GPT-5-mini	12 (36.4%)	4 (12.1%)	64.3	52.8	84.4	48.3
Gemini 3 Pro	13 (41.9%)	5 (16.1%)	63.7	37.1	77.5	58.8

Table 5: Per-app graded scores (%) for the five additional PRDs, aggregated over each app's zero-to-one build and feature-extension artifacts.

App	Opus 4.6	GPT-5.2	GPT-5-mini	Gemini 3 Flash	Gemini 3 Pro
book_journey	97.9	95.9	96.5	96.9	91.1
fleet_management	97.1	100.0	54.2	85.9	97.1
language_learning	80.0	53.7	37.3	75.9	35.8
logistics	47.6	36.4	73.2	39.0	44.8
resume_builder	87.5	91.7	60.2	85.5	42.9

- Omitted Requirement: Completely missing a requested feature, endpoint, or critical UI element required by the specification.
- Misinterpreted Constraint: Implementing a feature but violating explicit business logic rules, such as overly restrictive access controls or incorrect math.
- Interface Oversight: Failing to implement necessary user experience details like visible error messages, submit buttons, or proper UI state resets.

Verification

- False Success Claim: Asserting that the code works or that tests passed without executing an adequate validation step.
- Unresolved Failure: Executing a test, explicitly observing a failure or stack trace, but shipping the code without resolving the defect.
- Shallow Verification: The agent performs superficial checks, such as isolated backend pings, that fail to validate end-to-end user workflows.

Tooling

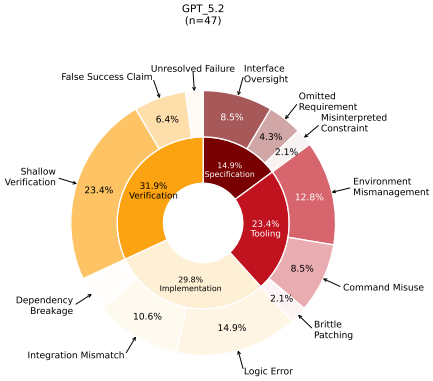
- Command Misuse: Violating terminal tool constraints, such as executing multiple commands at once or providing invalid timeout parameters.
- Brittle Patching: Failing to apply code edits correctly due to string match failures, lack of file context, or producing duplicate code.
- Environment Mismanagement: Running commands in the wrong directory, skipping setup scripts, or failing to install dependencies needed for test harnesses.

Other

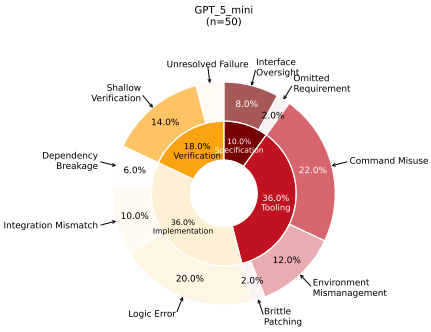
- External Defect: Highly specific or external failures, such as evaluation test harness mismatches or issues outside the agent's direct control.

F.2 Per-Model Failure Mode Pie Charts

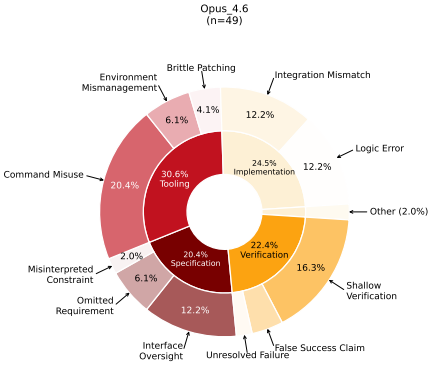
This section presents the per-model category distributions used in our qualitative analysis (Figure 6).



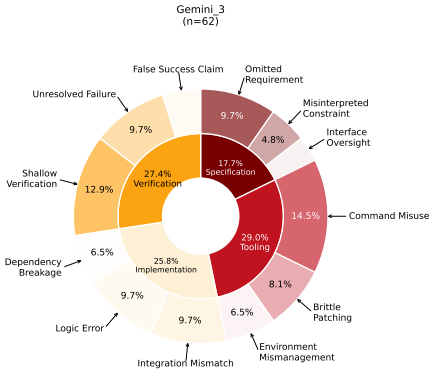
(a) GPT-5.2



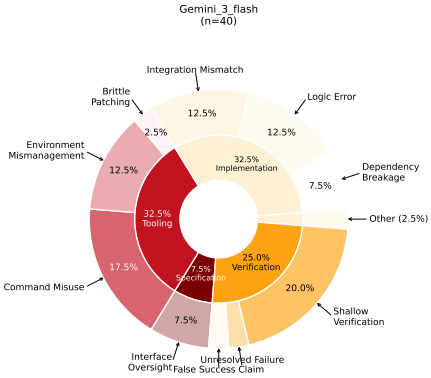
(b) GPT-5-mini



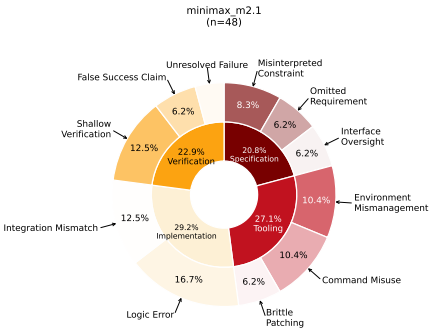
(c) Opus 4.6



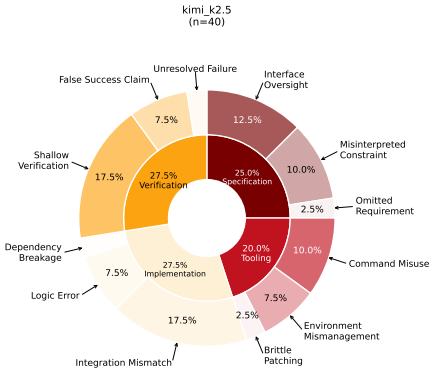
(d) Gemini 3 Pro



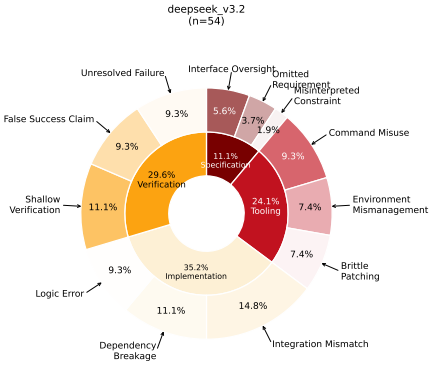
(e) Gemini 3 Flash



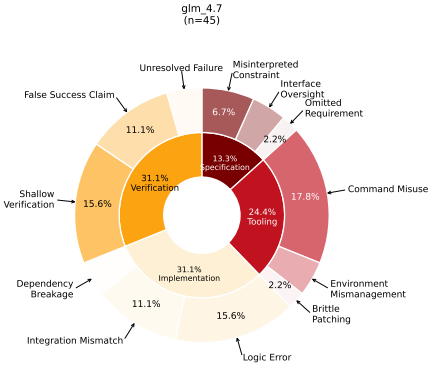
(f) MiniMax-M2.1



(g) Kimi-K2.5



(h) DeepSeek-V3.2



(i) GLM-4.7

Figure 6: Per-model failure mode category distributions.

F.3 Example Qualitative Summary

Below is an example qualitative summary for the Family Social Network MVP artifact, built by GPT-5.2. It was a complete failure (graded score of 0 on all tests), and the summary explains the root cause: the build agent shipped an authentication implementation with unstable dependency selection (unpinned transitive 'bcrypt') and insufficient end-to-end validation of the critical signup path before finishing.

```
# Agent Failure Modes and Root Causes (P0, narrative findings)

## 1) Fatal signup failure from passlib/bcrypt incompatibility (blocks all tests)

- **Tests + outcomes:**
  - `test1` **score 0/42** (FATAL at Step 1: signup rejects a valid 14-char password with "password cannot be longer than 72 bytes") [1]
  - `test2` **score 0/70** (FATAL at Step 1: cannot create required user accounts) [1]
  - `test3` **score 0/70** (FATAL at Step 1: cannot create any accounts) [1]

- **BUILD-agent failure mode:** shipped an authentication implementation with *unstable dependency selection* (unpinned transitive `bcrypt`) and *insufficient end-to-end validation* of the critical signup path before finishing.

- **Root-cause construction chain (agent behavior -> artifact -> outcome):**
  1. BUILD agent added `passlib[bcrypt]==1.7.4` but did **not** pin the transitive `bcrypt` package in `requirements.txt` [2][3].
  2. Both the build setup run and the evaluator's fresh install resolved that unpinned dependency to **`bcrypt==5.0.0`** [4][5].
  3. At runtime, passlib's bcrypt handler attempts to read `_bcrypt.__about__.__version__` and crashes with `AttributeError: module 'bcrypt' has no attribute '__about__'`, which then surfaces as the misleading UI error about the "72 bytes" password limit [5][1].
  4. The signup handler catches `ValueError` broadly and flashes the raw exception string (`_flash(request, str(e))`), leaking internal hashing/library errors into the user-facing form validation channel [6].
  5. BUILD agent's manual validation only exercised `GET /` (redirect to login) and `GET /login` via `curl`, and the captured server log shows no signup attempt—so the hashing breakage was not discovered prior to completion [7][8].

- **Impact:** account creation is a hard precondition for every evaluation plan; breaking signup causes a **Step-1 fatal stop** across all tests and yields **0 points**.

# Tool-Use and Command-Level Failure Patterns (P1, required)

## Command failure stats (from `/report/utilities/trace_analysis.json`)

- **terminal_observations:** 50 [9]
- **failed_terminal_observations:** 1 [9]
- **terminal_failure_rate:** 0.02 [9]
- **failure_type_counts:**
  - **non_zero_exit:** 1 [9]
  - **timeout:** 0 [9]
  - **tool_error_flag:** 0 [9]

**Important note on recurrence:** no recurring/trend-level command failure patterns were detected. The single terminal failure appears in an evaluation trace (test3) and does not represent a repeated BUILD-stage tool failure pattern [9].

## Recurring failed command groups (grouped)

| Group Key | Command (preview) | Prefix | Failure Types | Count | Session Count | Exit Codes | Evidence |
|---|---|---|---|---|---|---|---|
| `7ec5d75a.\|non_zero_exit` | `cd /app && python3 -c "..hash_password('password123')."` | `cd` | non_zero_exit | 1 | 1 | 1 | Isolated failure in `test_plans/test3/.../event-00049-...json` [9] |

## Interpretation (highest-impact patterns)

- **Command/tool failures were not the dominant issue.** The BUILD-like session recorded **0 terminal failures** while still shipping a critical dependency incompatibility [9].
- The higher-impact BUILD-agent weakness is **brittle verification coverage**: validation focused on "server starts + login page renders" (`curl` GETs) rather than executing the **state-changing signup path** that depends on password hashing [7][8]. This allowed a transitive dependency break (`bcrypt==5.0.0` + `passlib==1.7.4`) to persist into evaluation.

**Did the BUILD agent notice/test failing behavior before finishing?** No. The available build traces and server log show GET-only smoke checks (index/login) and no signup attempt, despite auth being evaluation-critical [7][8].

# References

[omitted for brevity]
```

G Application Descriptions

This appendix summarizes each application in ViBench, including the base MVP and feature extensions. Many of these applications include features that might be considered non-standard but in fact represent requests that the source platform receives. Users tend to customize applications to fit their own niche.

Mafia Game. A multiplayer party game where players are secretly assigned as Citizens or Mafia. During Day phases, players vote to eliminate suspects; at Night, Mafia secretly selects a victim. Citizens win by eliminating all Mafia; Mafia wins when they equal or outnumber Citizens. *Features:* (1) Multi-room support with shareable join codes; (2) Spectator mode with omniscient view of roles and votes; (3) Phase-based chat with Mafia-only night communication.

Communication App. A team communication app with channels, direct messages, and message search. *Features:* (1) Message threads with aggregated "My Messages" view; (2) Message editing, deletion, and reactions; (3) Invite-based registration and user-created channels.

Whiteboard. A real-time collaborative whiteboard with anonymous link-based access via separate edit and view-only URLs. *Features:* (1) Action history with revert functionality; (2) Board duplication; (3) Content import from other boards.

Energy Audit. A field inspection tool for verifying building code compliance, supporting checklists, blower door and duct leakage tests, and photo documentation. *Features:* (1) Structured builder records linked to jobs; (2) Photo tagging by inspection type, status, priority, and location; (3) Analytics dashboard with compliance rates and test result charts.

Family Social Network. A private social platform where users form verified family relationships (Spouse, Parent, Child, Sibling) and share posts visible only to direct family members. *Features:* (1) Photo albums; (2) Per-post visibility controls by relationship type; (3) Comments and reactions.

HVAC Service Platform. A service request platform for HVAC and plumbing. Clients submit bookings, technicians triage and execute jobs, and issue invoices. Includes property and equipment tracking. *Features:* (1) RFP and bidding workflow with vendor management and purchase orders; (2) Admin-only analytics dashboard for revenue, jobs, and client metrics; (3) Job rescheduling for appointments in scheduled status.

Marketplace. A peer-to-peer marketplace using unique links instead of user accounts. Sellers list items; buyers browse and order; payment is arranged offline. *Features:* (1) Advanced search with multi-select filters, price range, and sorting; (2) Session-based saved items; (3) Email-based order lookup; (4) Offline mode with queued listings and auto-sync.

Collaborative Kanban. A real-time collaborative sprint board for small teams with no authentication. Users enter a display name and join a single shared board with four fixed columns (Backlog, In Progress, Review, Done). Cards can be created, edited, moved via drag-and-drop, and deleted, with changes syncing instantly to all connected users. *Features:* (1) Multiple persistent boards with unique URLs and admin passwords; (2) Card assignment to connected users with notifications and a "My Cards" filter; (3) Invite-code-gated user accounts with a "My Work" portal showing assigned cards and story point summaries.

Creative Community. An invite-only creative community where artists discover and RSVP to events with capacity limits. Admins manage invite codes, events, users, attendance, and payments. Payment occurs off-platform; admins record payment status and apply no-show fees. *Features:* (1) Artist portfolios with external media links and member-to-member collaboration credits; (2) Members-only community hub with admin announcements and member comments; (3) RSVP cancellation support and automatic waitlist promotion.

Family-Friendly Venue Finder. A mobile-first web app for discovering family-friendly venues in Pittsburgh with no login required. Users browse venues by category, set preferences (cuisine, dietary options, facilities, keywords), and view results sorted by distance with driving and walking times. *Features:* (1) Admin interface for managing establishments with photos and reviews; (2) Visitor-submitted venue suggestions with community voting and approval; (3) Comments with AI moderation for family-friendliness and device-level banning.

Furniture Freight Calculator. A deterministic calculator for furniture delivery quotes with no payments or user accounts. An admin configures locations, rate cards, and a furniture catalog. Users calculate quotes by selecting delivery type, origin, destination, items, and services; the system matches rate cards using exact or city-level tiers. *Features:* (1) CSV upload for locations, rate cards, and catalog, plus a paginated rate card viewer; (2) Approximate matching using normalized strings, admin-managed synonyms, and longest common substring logic; (3) Enhanced rate card management with inline editing, undo, filtering, sorting, and CSV export.

Monopoly. A pass-the-computer Monopoly-style board game for 2–4 players in a single browser window. Board layout, properties, and dice sequences are loaded from environment variables. Players take turns rolling dice, moving, buying properties, paying rent and taxes, and can go bankrupt. The game ends when one player remains. *Features:* (1) Deterministic Chance cards with effects such as gaining or losing money and moving to specific spaces; (2) Networked multiplayer where players join via unique URLs from their own devices with real-time state sync; (3) Admin configuration screen for customizing board layout and property economics before starting a game.

Pilot Logbook. A single-user digital pilot logbook for recording flights and viewing analytics and currency status. Users manage aircraft records, log flights with detailed time and count data, filter logbook entries, view totals grouped by category, and track day, night, and instrument currency requirements. *Features:* (1) Flight templates for prefilling the add-flight form and saved filter views with optional relative dates; (2) Instructor endorsements via same-device digital signatures with immutable snapshots and public verification URLs; (3) Owner role for aircraft maintenance tracking with inspection due dates and status badges.

Supplier Relationship Management. A procurement platform managing the full purchasing lifecycle from request through fulfillment. Admin and Buyer roles manage purchase requests on a Kanban board, suppliers with performance scoring, RFQs with quote collection and comparison, winner selection, and purchase orders with status tracking. *Features:* (1) Inventory management with stock tracking, low-stock alerts, movement transactions, and replenishment requests that create pre-filled purchase requests; (2) Multi-organization support with complete data isolation and a SuperAdmin role; (3) Configurable procurement strategies (package deal, single-source, hybrid) with system recommendations and multi-supplier award support.

Wedding Venue Platform. A wedding venue platform connecting Couples and Venue Managers. Couples search venues by postcode, date, and guest count, view availability calendars, book tour slots, and request wedding bookings. Managers list venues, manage tour availability, and approve or decline bookings. *Features:* (1) Task management within confirmed bookings where managers create tasks and couples toggle status on assigned tasks; (2) Real-time in-app messaging between couples and venue managers with unread indicators and popup notifications.