Scaling Environments for Code Generation Agents: A Production Framework for Agentic Prompt-to-App Generation

Anonymous Author(s)
Affiliation
Address
email

Abstract

We present app.build, an open-source framework that improves LLM-based application generation through systematic validation and structured environments. Our approach combines multi-layered validation pipelines, stack-specific orchestration, and model-agnostic architecture, implemented across three reference stacks. Through evaluation on 30 generation tasks, we demonstrate that comprehensive validation achieves 73.3% viability rate with 30% reaching perfect quality scores, while open-weights models achieve 80.8% of closed-model performance when provided structured environments. The open-source framework has been adopted by the community, with over 3,000 applications generated to date. This work demonstrates that scaling reliable AI agents requires scaling environments, not just models—providing empirical insights and complete reference implementations for production-oriented agent systems.

1 Introduction

1.1 The Production Reliability Gap

While AI coding agents demonstrate impressive capabilities on standard benchmarks of isolated tasks like HumanEval [Chen et al., 2021] and MBPP [Austin et al., 2021], relying on them to build production-ready applications without human supervision remains infeasible. Recent repository-level systems such as Devin [Labs, 2024] and SWE-agent [Yang et al., 2024] represent significant advances, yet their performance on real-world software engineering tasks reveals a substantial gap between research benchmarks and production requirements.

This gap manifests across multiple dimensions. Function-level benchmarks like HumanEval evaluate isolated code generation but fail to capture system-level concerns including error handling, integration complexity, and production constraints [Liu et al., 2023]. Even state-of-the-art systems like AutoCodeRover, achieving 19% efficacy on SWE-bench at $0.43 per issue [Zhang et al., 2024], demonstrate that raw model capability alone is insufficient for reliable automated software development.

The core challenge lies in treating LLMs as standalone systems rather than components requiring structured environments. Current approaches predominantly focus on making models "smarter" via either training or prompt engineering, but this paradigm fails to address fundamental reliability issues inherent in probabilistic generation. Recent surveys [Jiang et al., 2024, Paul et al., 2024] note the field requires a shift from model-centric to environment-centric design.

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

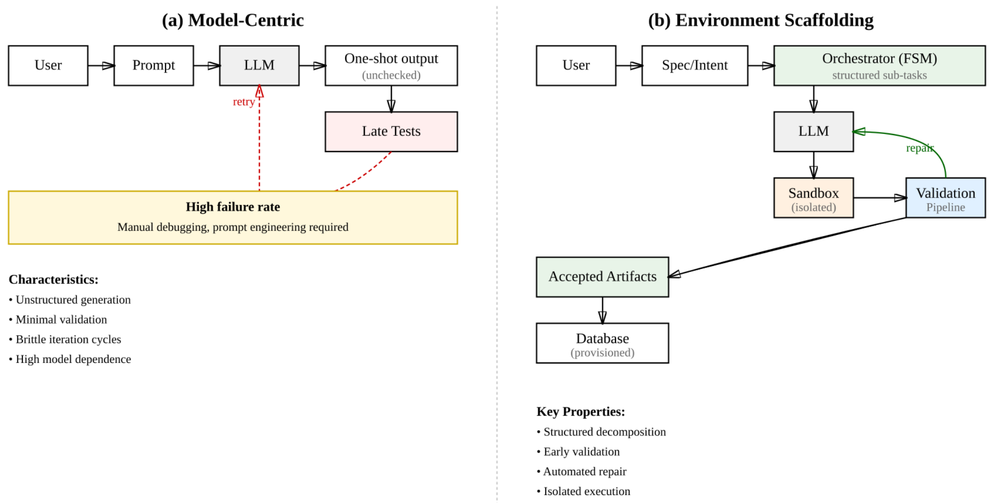

Figure 1: **Environment scaffolding vs. model-centric generation.** ES wraps the model with a finite, validated workflow that catches errors early and repairs them before proceeding.

1.2 Our Approach: Environment Scaffolding

Definition. We define *environment scaffolding (ES)* as an **environment-first** paradigm for LLM-based code generation where the model operates inside a structured sandbox that constrains actions and provides continuous, deterministic feedback. Rather than relying on larger models or prompt-only techniques, ES *improves the context* around the model — shaping the action space, providing templates and tools, and validating each step — so that creativity is channeled into *safe, verifiable* outcomes.

Principles.

1. **Structured task decomposition.** The agent works through an explicit sequence of well-scoped tasks (e.g., schema $\rightarrow$ API $\rightarrow$ UI), each with clear inputs/outputs and acceptance rules.
2. **Multi-layered validation.** Deterministic checks (linters, type-checkers, unit/smoke tests, runtime logs) run *after every significant generation*, catching errors early and feeding them back for automatic repair.
3. **Runtime isolation.** All code executes in isolated sandboxes (containers) with ephemeral state, enabling safe trial-and-error and reproducible re-runs.
4. **Model-agnostic integration.** The scaffolding is decoupled from any particular LLM; different backends can be swapped without changing the workflow.

Why ES vs. model-centric approaches? Traditional (model-centric) systems prompt an LLM to generate the full solution in one or few passes, with checks (if any) at the end. ES, in contrast, enforces a guarded, iterative loop: generate $\rightarrow$ validate $\rightarrow$ repair, per sub-task. Figure 1 and Table 1 summarize the contrast.

1.3 Contributions

Our work advances *environment-first* agent design. The main contributions are:

- **Environment Scaffolding Paradigm.** We formalize *environment scaffolding (ES)* and show how structuring the action space with per-step validation enables reliable code generation without model-specific tricks.

Table 1: **Environment scaffolding (ES) vs. model-centric generation.**

Aspect	Model-Centric	Environment Scaffolding (Ours)
Task decomposition	Single/loosely guided multi-step; no fixed structure	Explicit pipeline (FSM): schema → API → UI
Validation	Late or ad-hoc checks	Integrated per-step: linters, type checks, unit/smoke tests
Error recovery	Manual/ad-hoc retries	Automatic repair loop using error feedback
Execution isolation	Often none; runs on host	Isolated containers; reproducible runs
Model dependence	Strong (prompt/model specific)	Model-agnostic; environment guides behavior
Observability	Limited, coarse logs	Per-step metrics, artifacts, and logs

- **Open-Source Framework (app.build).** We release an implementation of ES that targets three stacks (TypeScript/tRPC, PHP/Laravel, Python/NiceGUI) and ships with validators and deployment hooks.[1]
- **Empirical Evaluation.** Across end-to-end app-building tasks, we quantify the effect of validation layers and iterative repair, and compare multiple LLM backends under the same environment.
- **Methodological Insight.** We find that improving the *environment* (constraints, tests, repair loops) often matters more than scaling the model for production reliability.
- **Community Adoption.** The framework has been used to generate thousands of applications in practice, suggesting ES is useful beyond controlled experiments.

2 Background and Related Work

2.1 Agentic Software Engineering

The evolution of AI coding agents has progressed from simple code completion to autonomous software engineering systems capable of repository-level modifications. **SWE-bench** [Jimenez et al., 2024] established the gold standard for evaluating repository-level understanding with 2,294 real GitHub issues from 12 Python projects. The accompanying **SWE-agent** [Yang et al., 2024] demonstrated that custom agent-computer interfaces significantly enhance performance, achieving 12.5% pass@1 through careful interface design rather than model improvements.

Repository-level agents have emerged as a distinct research direction. **WebArena** [Zhou et al., 2024] revealed that even GPT-4 achieves only 14.41% success versus 78.24% human performance in realistic environments, demonstrating that environment design matters more than model capability. **GAIA** [Mialon et al., 2023] reinforces this with 92% human versus 15% GPT-4 performance on practical tasks. **AutoCodeRover** [Zhang et al., 2024] combines LLMs with spectrum-based fault localization, achieving 19% efficacy on SWE-bench at $0.43 per issue. More recently, **Agentless** [Xia et al., 2024] challenged complex agent architectures with a simple three-phase process (localization, repair, validation) achieving 32% on SWE-bench Lite at $0.70 cost, suggesting that sophisticated architectures may not always improve performance.

Multi-agent systems have consistently outperformed single-agent approaches. **AgentCoder** [Huang et al., 2024] employs a three-agent architecture (Programmer, Test Designer, Test Executor) achieving 96.3% pass@1 on HumanEval with GPT-4, compared to 71.3% for single-agent approaches. **MapCoder** [Islam et al., 2024] extends this with four specialized agents replicating human programming cycles, achieving 93.9% pass@1 on HumanEval and 22.0% on the challenging APPS benchmark. **MetaGPT** [Hong et al., 2024] demonstrates role-based agents communicating through structured documents, achieving 85.9% pass@1 on HumanEval with 100% task completion on software development tasks.

[1] See repository overview for supported stacks and validators.

2.2 Production Quality in Generated Code

Ensuring production-ready AI-generated code requires validation approaches beyond simple correctness testing. **Static analysis integration** has shown promise, with intelligent code analysis agents combining GPT-3/4 with traditional static analysis to reduce false-positive rates from 85% to 66%. **Testing frameworks** have evolved to address AI-specific challenges. Test-driven approaches like TiCoder achieve 45.97% absolute improvement in pass@1 accuracy through interactive generation. Property-based testing frameworks show 23.1–37.3% relative improvements over established TDD methods by generating tests that capture semantic properties rather than specific implementations.

AST-based validation provides structural correctness guarantees. AST-T5 leverages Abstract Syntax Trees for structure-aware analysis, outperforming CodeT5 by 2–3 points on various tasks. Industry deployment reveals gaps between offline performance and practical usage. CodeAssist collected 2M completions from 1,200+ users over one year, revealing significant discrepancies between benchmark performance and real-world usage patterns.

2.3 Tree Search

Tree search enhances LLM-based solutions and serves as a way to increase compute budget beyond internal model reasoning token budget. The closest approach is used by Li et al. in S* Scaling [Li et al., 2025] by combining iterative feedback with parallel branches taking different paths toward solving the problem. Sampling more trajectories increases success rate significantly, which is evident by difference in pass@1 and pass@3 often by 30% or more.

2.4 Runtime Isolation and Scaling

Sandboxing is a cornerstone due to web applications requiring much more elaborate testing than running unit tests. It includes setup and teardown of databases and browser emulation. For parallel scaling, we use Dagger.io for its caching capabilities and Docker compatibility.

3 Problem Setup and Method

3.1 Problem Formulation

LLM-based code generation enables rapid prototyping but often produces code that does not meet production standards. We formalize this as an environment design problem where success depends not just on model capability but on the structured constraints and validation feedback provided by the generation environment.

3.2 Architecture

High-level design. The app.build agent implements ES with a central *orchestrator* that decomposes a user's specification into stack-specific stages and executes each stage inside an isolated sandbox with validation before acceptance. The same workflow applies across supported stacks (TypeScript/tRPC, PHP/Laravel, Python/NiceGUI). Per-stage validators are stack-aware (e.g., ESLint+TypeScript and Playwright for tRPC; PHPStan and feature tests for Laravel; `pytest`/`ruff`/`pyright` for Python), and the platform provisions managed Postgres databases and CI/CD hooks.

Execution loop. For each sub-task, the agent (i) assembles minimal context (files, interfaces, constraints), (ii) prompts the LLM, (iii) executes the result in a sandbox, (iv) collects validator feedback, and (v) either accepts the artifact or re-prompts to repair. This iterative loop provides robustness without assuming a particular model, and scales by parallelizing sandboxes and caching environment layers.

4 Experimental Setup

We designed experiments using a custom prompt dataset and metrics to evaluate viability and quality of generated applications.

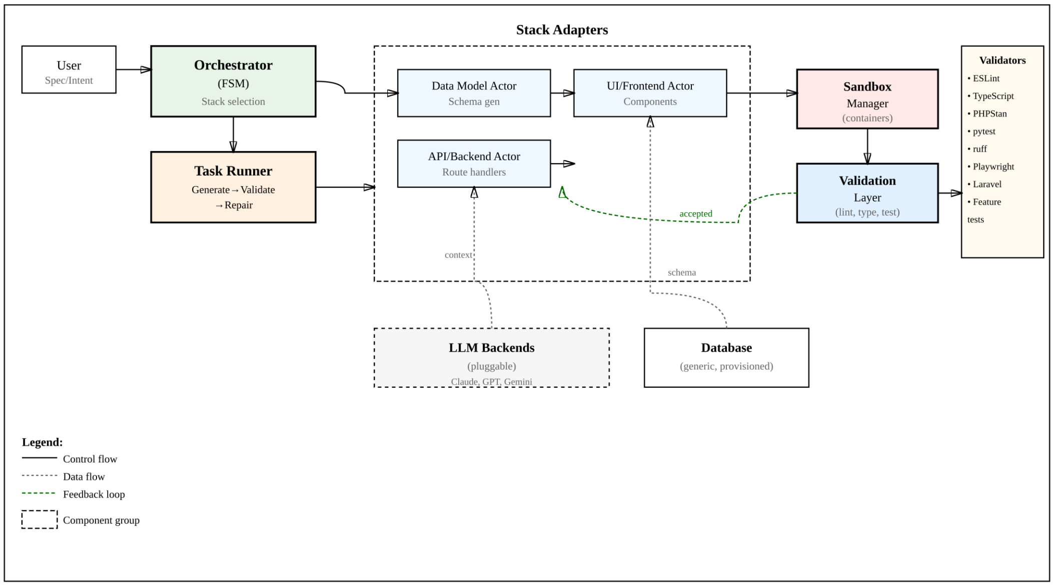

Figure 2: **app.build architecture** expressed through environment scaffolding. The orchestrator plans stages per stack; each sub-task runs in a sandbox, is validated, and only then merged. CI/CD and DB provisioning are integrated.

4.1 Evaluation Framework

4.2 Prompt Dataset

The evaluation dataset comprises 30 prompts designed to assess system performance across diverse application development scenarios. Independent human contributors with no prior exposure to the app.build system created evaluation prompts. Contributors developed tasks reflecting authentic development workflows from their professional experience. Prompts were filtered to exclude enterprise integrations, AI/ML compute requirements, or capabilities beyond framework scope. Raw prompts underwent automated post-processing using LLMs to anonymize sensitive information and standardize linguistic structure. The resulting dataset consists of 30 prompts spanning a complexity spectrum (low: static/single-page UI; medium: single-entity CRUD; high: multi-entity/custom logic). See the full list of prompts in Appendix A.

4.3 Metrics

Each application generated by the agent was evaluated by the following metrics, designed to assess its viability and quality under preset time and cost constraints.

- Viability rate ($V = 1$) and non-viability rate ($V = 0$)
- Perfect quality rate ($Q = 10$) and quality distribution (mean/median for $V = 1$ apps)
- Validation pass rates by check (AB-01, AB-02, AB-03, AB-04, AB-06, AB-07)
- Quality scores (Q, 0–10) using the rubric in Section 4.5
- Model/cost comparisons where applicable

4.4 Experimental Configurations

We designed three experimental configurations to systematically evaluate factors affecting app generation success rates:

Configuration 1: Baseline. We generated baseline tRPC apps with default production setup and all checks ON to assess default generation success rate, cost and time.

Table 2: Check weights and definitions used in scoring (see rubric in Section 4.5). All checks share equal weight after NA re-normalization; AB-01 and AB-02 are hard gates for Viability V.

Check ID	Check Description	Weight (share)	Notes
AB-01	Boot & Home	1/6	Hard gate for Viability V
AB-02	Prompt Correspondence	1/6	Hard gate for Viability V
AB-03	Create Functionality	1/6	
AB-04	View/Edit Operations	1/6	
AB-06	Clickable Sweep	1/6	
AB-07	Performance Metrics	1/6	Continuous; normalized to $[0, 1]$

Note. See mapping of PASS/WARN/FAIL to numeric scores and viability definition in Section 4.5.

Configuration 2: Model Architecture Analysis. Using the tRPC stack, we evaluated open versus closed foundation models. Claude Sonnet 4 served as the baseline coding model, compared against Qwen3-Coder-480B-A35B [Yang et al., 2025] and GPT OSS 120B [OpenAI et al., 2025] as open alternatives.

Configuration 3: Testing Framework Ablation. We conducted three ablation studies on the tRPC stack isolating the impact of each type of checks by turning them off independently: (3a) disabled isolated Playwright UI smoke tests; (3b) disabled ESLint checks; and (3c) removed handlers tests, eliminating backend validation.

4.5 Assessor Protocol and Scoring

To systematically assess generated application quality, we implement a structured evaluation protocol comprising six standardized functional checks executed by human assessors. The evaluation reports two independent outcomes: a binary viability indicator (V) and a 0–10 quality score (Q).

Viability (binary):

$$V = \begin{cases} 1 & \text{if AB-01 and AB-02 are not FAIL} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Quality (0–10):

$$Q = 10 \times \frac{\sum_{c \in A} w \times s_c}{\sum_{c \in A} w} \tag{2}$$

where A is the set of applicable checks (excluding NA); all checks use equal weights prior to NA re-normalization; and per-check grades s_c are mapped as follows:

- AB-01 (Boot): PASS = 1.0, WARN = 0.5, FAIL = 0.0
- AB-02 (Prompt correspondence): PASS = 1.0, WARN = 0.5, FAIL = 0.0
- AB-03, AB-04, AB-06 (Clickable Sweep): PASS = 1.0, WARN = 0.5, FAIL = 0.0
- AB-07 (Performance): continuous metric normalized to $[0, 1]$

5 Results

5.1 Environment Scaffolding Impact (TypeScript/tRPC only)

Evaluating 30 TypeScript/tRPC applications, we observe that 73.3% (22/30) achieved viability ($V = 1$), with 30.0% attaining perfect quality ($Q = 10$) and 26.7% non-viable ($V = 0$). Once viability criteria are met, generated applications exhibit consistently high quality.

Smoke tests (AB-01, AB-02) determine viability. Among viable applications ($V = 1$, $n = 21$), quality averaged 8.78 with 77.3% achieving $Q \geq 9$. Non-viability ($V = 0$) arises from smoke test failures or missing artifacts.

Table 3: Aggregated evaluation results for TypeScript/tRPC ($n = 30$ prompts). Viability V and quality Q are defined in Section 4.5. "Perfect quality" denotes $Q = 10$ (all applicable checks PASS). "Non-viable" denotes $V = 0$ (AB-01 or AB-02 = FAIL). Mean quality is computed over viable apps only ($V = 1$).

Metric	Value	Key Insight
Total Applications	30	TypeScript/tRPC stack only
Viability Rate ($V = 1$)	73.3%	22/30 viable applications
Perfect Quality ($Q = 10$)	30.0%	9/30 fully compliant applications
Non-viable ($V = 0$)	26.7%	8/30 failed smoke tests
Mean Quality ($V = 1$ apps)	8.78	High quality when viable

Note. Scoring rubric and check definitions in Section 4.5.

Table 4: Check-specific outcomes across $n = 30$ TypeScript/tRPC tasks. See Section 4.5 for check definitions, PASS/WARN/FAIL grading, and the viability rule. NA indicates the check was not applicable to a prompt (e.g., AB-04 when no view/edit flows are required). "Pass Rate (excl. NA)" is computed over applicable cases only.

Check	Pass	Warn	Fail	NA	Pass Rate (excl. NA)
AB-01 (Boot)	25	2	3	0	83.3%
AB-02 (Prompt)	19	3	5	3	70.4%
AB-03 (Create)	22	2	0	6	91.7%
AB-04 (View/Edit)	17	1	1	11	89.5%
AB-06 (Clickable Sweep)	20	4	1	5	80.0%
AB-07 (Performance)	23	3	0	4	88.5%

Note. AB-07 is a continuous metric normalized to $[0, 1]$; thresholding for PASS/WARN/FAIL is specified in Section 4.5.

5.2 Open vs Closed Model Performance

We evaluated Claude Sonnet 4 against two open-weights models using the TypeScript/tRPC stack with simplified validation pipeline ensuring the app is bootable and renders correctly. Claude achieved 86.7% success rate, establishing our closed-model baseline at \$110.20 total cost. Qwen3-Coder-480B-A35B reached 70% success rate (80.8% relative performance) while GPT OSS 120B managed only 30% success rate. Both open models were accessed via OpenRouter, resulting in significantly lower costs: \$12.68 for Qwen3 and \$4.55 for GPT OSS.

The performance gap reveals that environment scaffolding alone cannot eliminate the need for capable foundation models. However, leading open-weights models like Qwen3 demonstrate that structured environments can enable production-viable performance at substantially reduced costs. The 9x cost reduction for 19% performance loss represents a viable tradeoff.

Operational characteristics differed notably between model types. Open models required more validation retries, evidenced by higher LLM call counts (4,359 for Qwen3, 4,922 for GPT OSS vs 3,413 for Claude). Healthcheck pass rates (86.7% for Qwen3 vs 96.7% for Claude) indicate open models generate syntactically correct code but struggle with integration-level correctness, emphasizing the importance of comprehensive validation.

5.3 Ablation Studies: Impact of Validation Layers

To understand how each validation layer contributes to application quality, we conducted controlled ablations on the same 30-prompt cohort. Each ablation removes one validation component while keeping others intact.

Baseline Performance (all validation layers active):

- Viability: 73.3% (22/30 apps pass both AB-01 Boot and AB-02 Prompt)
- Mean Quality: 8.06 (among all 30 apps)

Finding 1: Removing Unit Tests Trades Quality for Viability

- Viability: 80.0% (+6.7 pp) – fewer apps fail smoke tests
- Mean Quality: 7.78 (-0.28) – quality degrades despite higher viability
- Key degradations: AB-04 View/Edit drops from 90% to 60% pass rate
- Interpretation: Backend tests catch critical CRUD errors. Without them, apps boot successfully but fail on data operations.

Finding 2: Removing Linting Has Mixed Effects

- Viability: 80.0% (+6.7 pp)
- Mean Quality: 8.25 (+0.19) – slight improvement
- Trade-offs: AB-03 Create drops 8.3 pp, AB-04 View/Edit drops 7.6 pp
- Interpretation: ESLint catches legitimate issues but may also block valid patterns. The performance gain suggests some lint rules may be overly restrictive.

Finding 3: Removing Playwright Tests Significantly Improves Outcomes

- Viability: 90.0% (+16.7 pp) – highest among all configurations
- Mean Quality: 8.62 (+0.56) – meaningful quality improvement
- Broad improvements: AB-02 Prompt +11.8 pp, AB-06 Clickable +5.7 pp
- Interpretation: Playwright tests appear overly brittle for scaffolded apps. Many apps that fail E2E tests actually work correctly for users.

5.4 Synthesis: Optimal Validation Strategy

Our ablation results reveal clear trade-offs in validation design:

Validation Layer Impact Summary:

1. **Unit/Handler Tests**: Essential for data integrity. Removing them increases perceived viability but causes real functional regressions (especially AB-04 View/Edit).
2. **ESLint**: Provides modest value with some false positives. The small quality impact (+0.19) and mixed per-dimension effects suggest selective application.
3. **Playwright/E2E**: Currently causes more harm than good. The +16.7 pp viability gain and quality improvements indicate these tests reject too many working applications.

Recommended Validation Architecture: Based on these findings, we recommend:

- **Keep**: Lightweight smoke tests (boot + primary route), backend unit tests for CRUD operations
- **Refine**: ESLint with curated rules focusing on actual errors vs style preferences
- **Replace**: Full E2E suite with targeted integration tests for critical paths only

This pragmatic approach balances catching real defects while avoiding false rejections. When quality is paramount and compute budget less constrained, comprehensive validation including strict E2E tests remains viable—trading lower success rates for guaranteed production quality.

5.5 Failure Mode Analysis

Failure modes in tRPC runs cluster into categories:

- **Boot/Load failures**: template placeholders or incomplete artifacts
- **Prompt correspondence failures**: generic templates from generation failures
- **CSP/security policy restrictions**: blocked images or media by default policies

- **UI interaction defects**: unbound handlers, non-working controls
- **State/integration defects**: data not persisting across refresh; broken filters; login issues
- **Component misuse**: runtime exceptions from incorrect component composition

These defects align with our layered pipeline design: early gates catch non-viable builds, while later gates expose interaction/state issues before human evaluation.

5.6 Prompt Complexity and Success Rate

We categorize prompts along a simple rubric and analyze success impacts:

- **Low complexity**: static or single-page UI tasks (e.g., landing pages, counters)
- **Medium complexity**: single-entity CRUD without advanced flows or auth
- **High complexity**: multi-entity workflows, custom logic, or complex UI interactions

Medium-complexity CRUD prompts achieve the highest quality ($Q = 9$–10), reflecting strong scaffolding for data models and handlers. Low-complexity UI prompts are not uniformly easy: several failed prompt correspondence (AB-02) with generic templates. High-complexity prompts show lower viability rates due to interaction wiring and state-consistency issues surfaced by AB-04/AB-06.

6 Discussion

6.1 Limitations

Our current framework is limited to CRUD-oriented data applications, focusing on structured workflows with well-defined input-output expectations. While effective for common web application patterns, it does not yet support complex systems or advanced integrations. The validation pipeline, though comprehensive, relies on domain-specific heuristics and expert-defined anti-patterns, which may not generalize to novel or edge-case designs. Additionally, our human evaluation protocol, while rigorous, is poorly scalable and constrained by subjectivity in assessing maintainability and user experience nuances.

6.2 Broader Impact

The AI agent boom is accelerating, but real industry deployments often fail silently. Without environment scaffolding, we risk massive overengineering of AI models while ignoring the real bottleneck. App.build represents a shift from model-centric to system-centric AI engineering—a critical step toward scaling reliable agent environments. As practitioners emphasize [Babushkin and Kravchenko, 2025], production AI systems only become effective when development integrates not just model performance, but core software engineering principles. By open-sourcing both the framework and evaluation protocol, we provide a reproducible, transparent foundation for building and benchmarking agent environments at scale.

7 Conclusion

Our results demonstrate that raw model capability alone cannot bridge the gap between AI potential and production reality. Through systematic environment scaffolding, multi-layered validation, and stack-specific orchestration, app.build transforms probabilistic language models into dependable software engineering agents.

Ablations reveal clear trade-offs: removing unit tests increases apparent viability but reduces CRUD correctness; removing linting yields small gains with modest regressions; removing Playwright tests improves outcomes by eliminating flaky UI checks. These results support retaining minimal smoke tests for boot and primary flows, structural checks for UI/code consistency, and scoped E2E tests for critical paths only.

The path to reliable AI agents lies not in better prompts or bigger models, but in principled environment engineering with validation layers tuned to maximize value while minimizing brittleness.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

Valerii Babushkin and Arseny Kravchenko. *Machine Learning System Design with End-to-End Examples*. Manning Publications, 2025.

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. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework, 2024. URL https://arxiv.org/abs/2308.00352.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation, 2024. URL https://arxiv.org/abs/2312.13010.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving, 2024. URL https://arxiv.org/abs/2405.11403.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation, 2024. URL https://arxiv.org/abs/2406.00515.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Cognition Labs. Swe-bench technical report. https://cognition.ai/blog/swe-bench-technical-report, 2024.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E. Gonzalez, and Ion Stoica. S*: Test time scaling for code generation, 2025. URL https://arxiv.org/abs/2502.14382.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation, 2023. URL https://arxiv.org/abs/2305.01210.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants, 2023. URL https://arxiv.org/abs/2311.12983.

OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park

Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily, Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. gpt-oss-120b & gpt-oss-20b model card, 2025.

Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. Benchmarks and metrics for evaluations of code generation: A critical review, 2024. URL `https://arxiv.org/abs/2406.12655`.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents, 2024. URL `https://arxiv.org/abs/2407.01489`.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL `https://arxiv.org/abs/2505.09388`.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL `https://arxiv.org/abs/2405.15793`.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement, 2024. URL `https://arxiv.org/abs/2404.05427`.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents, 2024. URL `https://arxiv.org/abs/2307.13854`.

A Prompt Dataset (Full List)

Table 5: Complete prompt dataset used in evaluation ($n = 30$). Dataset construction details in Section 4.2. Complexity labels follow the rubric in Section 5.6: *Low* (static/single-page UI), *Medium* (single-entity CRUD), *High* (multi-entity/custom logic).

ID	Prompt (summary)	Complexity
plant-care-tracker	Track plant conditions using moods with custom rule-based logic. No AI/ML/APIs.	Medium
roommate-chore-wheel	Randomly assigns chores weekly and tracks completion.	Medium
car-maintenance-dashboard	Monitor car maintenance history and upcoming service dates.	Medium
city-trip-advisor	Suggest tomorrow's trip viability based on weather forecast API.	High
currency-converter	Convert currency amounts using Frankfurter API.	Low
book-library-manager	Manage book library with CRUD operations, search, and filters.	Medium
wellness-score-tracker	Input health metrics, get daily wellness score with trends.	High
event-tracker	Basic event tracker with add, view, delete functionality.	Low
daily-pattern-visualizer	Log and visualize daily patterns (sleep, work, social time).	High
pantry-inventory-app	Track pantry items, expiry notifications, AI recipe suggestions.	High
home-lab-inventory	Catalog home lab infrastructure (hardware, VMs, IP allocations).	High
basic-inventory-system	Small business inventory with stock in/out transactions.	Medium
pastel-blue-notes-app	Notes app with pastel theme, folders, user accounts.	Medium
teacher-question-bank	Question bank with quiz generation and export features.	High
beer-counter-app	Single-page beer counter with local storage.	Low
plumbing-business-landing-page	Professional landing page for lead generation.	Low
kanji-flashcards	Kanji learning with SRS, progress tracking, JLPT levels.	High
bookmark-management-app	Save, tag, organize links with search and sync.	Medium
personal-expense-tracker	Log expenses, categories, budgets, spending visualization.	Medium
gym-crm	Gym CRM for class reservations with admin interface.	High
todo-list-with-mood	To-do list combined with mood tracker.	Medium
birthday-wish-app	Static birthday card with message and animation.	Low
pc-gaming-niche-site	Budget gaming peripherals review site with CMS.	Medium
tennis-enthusiast-platform	Social platform for finding tennis partners.	High
engineering-job-board	Niche job board for engineering positions.	High
indonesian-inventory-app	Inventory management app in Indonesian language.	Medium
habit-tracker-app	Track habits, daily progress, visualize streaks.	Medium
recipe-sharing-platform	Community platform for sharing recipes.	High
pomodoro-study-timer	Minimalistic Pomodoro timer with session logging.	Low
cat-conspiracy-tracker	Humorous app tracking cat suspicious activities.	Low

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer **Yes**, **No**, or **N/A**.
- **N/A** means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for N/A).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: **Yes**

   Justification: The abstract and introduction clearly state our contributions regarding environment scaffolding for code generation agents, with specific claims supported by experimental results on 30 generation tasks.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: **Yes**

Justification: Section 6.1 explicitly discusses limitations including restriction to CRUD applications, reliance on domain-specific heuristics, and scalability challenges of human evaluation.

3. **Theory Assumptions and Proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: **N/A**

   Justification: This paper focuses on empirical evaluation of a practical system rather than theoretical contributions requiring formal proofs.

4. **Experimental Result Reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: **Yes**

   Justification: We provide detailed experimental configurations, evaluation protocols, and the complete prompt dataset. The framework is open-source with reference implementations.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: **Yes**

   Justification: The app.build framework is open-source, and we provide the complete evaluation dataset and protocols in Appendix A.

6. **Experimental Setting/Details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: **Yes**

   Justification: Section 4 provides comprehensive experimental setup including configurations, model choices, evaluation metrics, and detailed scoring protocols.

7. **Experiment Statistical Significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: **No**

   Justification: We report success rates and quality scores but do not include statistical significance tests due to the limited sample size (30 prompts) and focus on practical system evaluation rather than statistical inference.

8. **Experiments Compute Resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: **Partial**

   Justification: We mention Dagger.io infrastructure and Docker-based sandboxing but do not provide detailed compute resource specifications for reproduction.

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics?

   Answer: **Yes**

   Justification: Our research focuses on improving software development tools and does not involve human subjects, sensitive data, or potential for misuse.

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: **Yes**

    Justification: Section 6.2 discusses the broader impact of shifting from model-centric to environment-centric AI engineering and the importance of production-ready agent systems.

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse?

    Answer: **N/A**

    Justification: Our work involves a software development framework rather than models or datasets with high misuse potential.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: **Yes**

    Justification: We properly cite all referenced models, benchmarks, and tools. The app.build framework is released as open-source with appropriate licensing.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: **Yes**

    Justification: The app.build framework and evaluation dataset are well-documented with detailed protocols provided in the appendix and open-source repository.

14. **Crowdsourcing and Research with Human Subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: **N/A**

    Justification: Our evaluation involved human assessors following standardized protocols but did not constitute formal human subjects research requiring IRB approval.

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals were obtained?

    Answer: **N/A**

    Justification: The human evaluation did not involve study participants but rather standard software testing protocols by technical evaluators.