APP.BUILD: A PRODUCTION FRAMEWORK FOR SCALING AGENTIC PROMPT-TO-APP GENERATION WITH ENVIRONMENT SCAFFOLDING[*]

A PREPRINT

Evgenii Kniazev[1,†] Arseny Kravchenko[1,†] Igor Rekun[1,†] James Broadhead[1]
Nikita Shamgunov[1] Pranav Sah[2] Pratik Nichite[2]
Ivan Yamshchikov[2]

[1]Databricks
[2]THWS University of Applied Sciences Würzburg-Schweinfurt (CAIRO)
[†]Equal contribution

eng-appbuild@databricks.com

ABSTRACT

Engineering teams increasingly experiment with LLM agents to synthesize full-stack web applications, yet production reliability and code generation reproducibility remain the blocking issues. Ongoing improvements of foundational models alone do not reliably translate into deployable software; what matters in practice is the environment that constrains, validates, and repairs model outputs.

We present the app.build framework and report our industrial experience using environment scaffolding (stack-aware generate→validate→repair loops, sandboxed execution, and policy gates) to turn prompt-to-app generation into a dependable workflow. We conducted 300 end-to-end generation experiments with automated validation metrics, complemented by detailed human quality assessment on 30 representative prompts. The framework has been deployed in production and generated 3000+ user applications during 4 months of operation.

Across end-to-end app-building tasks, structured validators and code execution isolation improve the rate of viable apps (viability = pass boot + prompt-correspondence smoke checks) to 73.3% in our tests, while generic end-to-end browser tests introduce brittleness. Large-scale automated metrics (n=300) reveal that open-weights models achieve 80.8% performance of top closed model at 8.2× lower cost per viable app, with validation ablations showing lightweight smoke checks and backend contract tests deliver most reliability lift, whereas broad end-to-end suites often reject working apps.

This paper frames the problem as a software engineering challenge (reliability, maintainability, and cost in agentic development), provides a reproducible evaluation protocol validated at production scale, and distills lessons for practitioners deploying LLM agents. We release the open-source framework (650+ stars) and an artifact to reproduce the main tables.

Keywords software engineering · code generation · LLM agents · validation · environment scaffolding

1 Introduction

1.1 The Production Reliability Gap

The promise of Large Language Model (LLM) agents for automated software development has attracted significant industrial interest, yet a critical gap persists between research benchmarks and production requirements. While research systems demonstrate impressive capabilities on isolated benchmarks—HumanEval [Chen et al., 2021] leaders

[*] Accepted to SANER 2026 Industrial Track. Code: `https://github.com/neondatabase/appdotbuild-agent`

achieve 90%+ pass rates on function-level tasks, and LiveCodeBench [Jain et al., 2024] reaches over 80% on real GitHub issues—these metrics do not translate to deployable software in industrial contexts.

The gap manifests in three critical dimensions that affect practitioner adoption:

Reliability under constraints. Production systems must operate within fixed time and cost budgets while maintaining deterministic quality gates. LLMs generate probabilistically, producing syntactically correct code that fails integration tests, violates security policies, or exhibits subtle runtime defects [Liu et al., 2023].

Reproducibility and debugging. When generation fails, practitioners need actionable diagnostics. Model-centric approaches offer little guidance for iterative refinement. Practitioners need structured validation feedback that pinpoints specific failure modes, so that repairs can be targeted effectively.

Economic viability. At scale, token costs and iteration cycles determine feasibility. Closed frontier models like Claude Sonnet 4 [Anthropic, 2024] achieve high success rates but at significant costs. For teams generating hundreds of applications, these costs compound rapidly. The industry needs cost-performance tradeoffs: where can open-weights models substitute for frontier models? What validation overhead is justified by reliability gains?

We claim that leaning on model-only improvements is insufficient. The prevailing approach treats reliability as a model capability problem—scale parameters, improve training data, refine prompts. However, our production experience generating thousands of applications reveals that environment design matters more than model selection for industrial deployment. A frontier model without validation produces unreliable apps; industry needs explicit tradeoffs between cost, speed, and correctness. Production-ready systems require frameworks that integrate validation, isolation, and repair as first-class concerns—not post-hoc additions to model outputs. Recent surveys [Jiang et al., 2024, Paul et al., 2024] note the field requires a shift from model-centric to environment-centric design.

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.

How environment scaffolding works in practice Environment scaffolding structures LLM-based code generation around four core practices that address production reliability requirements:

Structured task decomposition. Rather than asking the model to generate an entire application at once, we break work into explicit stages (schema → Application Programming Interface (API) → User Interface (UI)) with defined inputs, outputs, and success criteria. This matches how developers actually build software and makes failures easier to diagnose.

Multi-layered validation. After every generation step, deterministic checks run automatically: linters catch syntax errors, type-checkers verify contracts, unit tests validate logic, and smoke tests ensure the app boots. Failures trigger immediate repair loops before moving forward, preventing error accumulation.

Runtime isolation. Every generation and test runs in an isolated sandbox with ephemeral state. If the model generates code that crashes or corrupts data, the container resets cleanly. This enables aggressive trial-and-error without risk to production systems.

Model-agnostic design. The scaffolding layer sits between your workflow and the LLM, allowing you to swap models (e.g., from Claude to Qwen) without rewriting validation logic. This protects against vendor lock-in and enables cost-performance optimization.

Why this differs from model-centric approaches Most existing systems prompt an LLM to generate code and then validate the complete output. This works for simple scripts but fails for full-stack applications where a single integration error can invalidate hours of generation work. Environment scaffolding instead enforces generate→validate→repair at each step, catching errors early when they are cheap to fix. Figure 1 and Table 1 illustrate this architectural difference.

1.3 Contributions

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

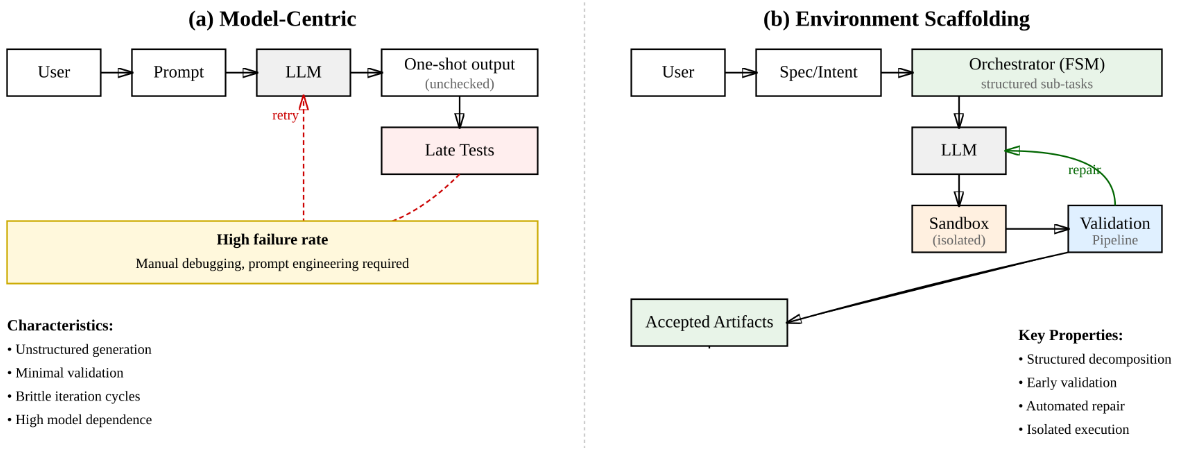

Figure 1: **Environment scaffolding vs. model-centric generation.** ES wraps the model with a finite, validated workflow that catches errors early and repairs them before proceeding.

Table 1: Environment Scaffolding (ES) vs. Model-Centric Generation

Aspect	**Model-Centric**	**ES (Ours)**
Task decomp.	Single/loosely guided; no fixed structure	Explicit FSM: schema → API → UI
Validation	Late or ad-hoc	Per-step: linters, types, tests
Error recovery	Manual/ad-hoc	Auto repair loop w/ feedback
Execution	Often on host	Isolated containers
Model dep.	Strong (prompt-specific)	Model-agnostic
Observability	Limited logs	Per-step metrics, artifacts

- **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.
- **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. The framework has gained 650+ GitHub stars and 89 forks, demonstrating practitioner adoption.
- **Two-Tier Empirical Evaluation.** We conduct 300 end-to-end generation experiments with automated metrics (success rate, cost, tokens, duration) plus detailed human evaluation on 30 representative prompts with 6-criteria quality rubric. This methodology combines automated consistency checks at scale with manual viability assessment on representative samples.
- **Production-Scale Validation.** The framework has been deployed in production since June 2025, generating over 3000 user applications during the first 4 months with hundreds of applications generated daily at peak usage, providing ecological validity beyond controlled experiments.
- **Cost-Performance Analysis.** We quantify validation overhead through token usage and cost-per-viable-app metrics, showing open-weights models (Qwen3) achieve 70% success at 8.2x lower cost ($0.61 vs $5.01 per viable app), while validation ablations reveal that comprehensive testing increases costs by $40 per cohort but catches real defects.
- **Methodological Insight.** We find that improving the *environment* (constraints, tests, repair loops) often matters more than scaling the model for production reliability, with lightweight smoke tests and backend validation providing most gains while End-to-End (E2E) browser tests introduce brittleness.

2 Related Work

Repository-level agentic SE (2024-2025). The evolution of AI coding agents has progressed from code completion to autonomous software engineering systems. **SWE-bench** [Jimenez et al., 2024] established the evaluation standard with 2,294 real GitHub issues from 12 Python projects. Recent agents demonstrate that environment design rivals model capability: **OpenHands** [Wang et al., 2024], published at ICLR 2025, achieves 53% on SWE-bench Verified through an open platform for generalist agents with agent-computer interfaces. **SWE-agent** [Yang et al., 2024] showed 12.5% pass@1 through careful interface design rather than model improvements. Contemporary 2024 agents include **AutoCodeRover** [Zhang et al., 2024], which combines LLMs with spectrum-based fault localization (19% on SWE-bench, $0.43 per issue), and **Agentless** [Xia et al., 2024], challenging architectural complexity with a simple three-phase process (localization, repair, validation) achieving 32% on SWE-bench Lite.

Validation and environment scaffolding. Production-ready code generation requires validation beyond correctness testing. While early explorations in this space focused on code change classification [Kniazev, 2008], modern frameworks now integrate validation at multiple layers. Test-driven approaches [Fakhoury et al., 2024] achieve 45.97% absolute improvement in pass@1 through interactive generation with dynamic test feedback. **AST-based validation** [Gong et al., 2024] provides structural guarantees, with AST-T5 outperforming CodeT5 by 2–3 points through structure-aware pretraining. Tree search methods [Li et al., 2025] demonstrate that scaling compute through iterative refinement and parallel branches can significantly improve success rates. Multi-agent systems [Hong et al., 2024] show that role-based collaboration with structured validation outperforms single-agent approaches, achieving 85.9% pass@1 on HumanEval with 100% task completion on development tasks. For web application generation, sandboxed execution with database provisioning and browser emulation is essential for isolating and validating complex multi-tier systems.

3 Industrial Context & System

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), selected for their deterministic scaffolding patterns and comprehensive validator availability (TypeScript/ESLint/Playwright, PHPStan/Laravel feature tests, pytest/ruff/pyright). Per-stage validators are stack-aware, 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.

4.1 Evaluation Framework

Our evaluation methodology addresses the core challenge of assessing prompt-to-app generation systems: validating that generated applications not only compile and boot, but actually implement the requested functionality correctly. We combine manual-heavy viability assessment with automated checks across 300 end-to-end experiments, where every generated application undergoes deterministic validation followed by human review on a representative subset.

Two-tier evaluation design. We conduct 300 end-to-end generation runs with automated checks measuring boot success, smoke test passage, cost, token usage, and duration. These automated gates provide fast, deterministic feedback

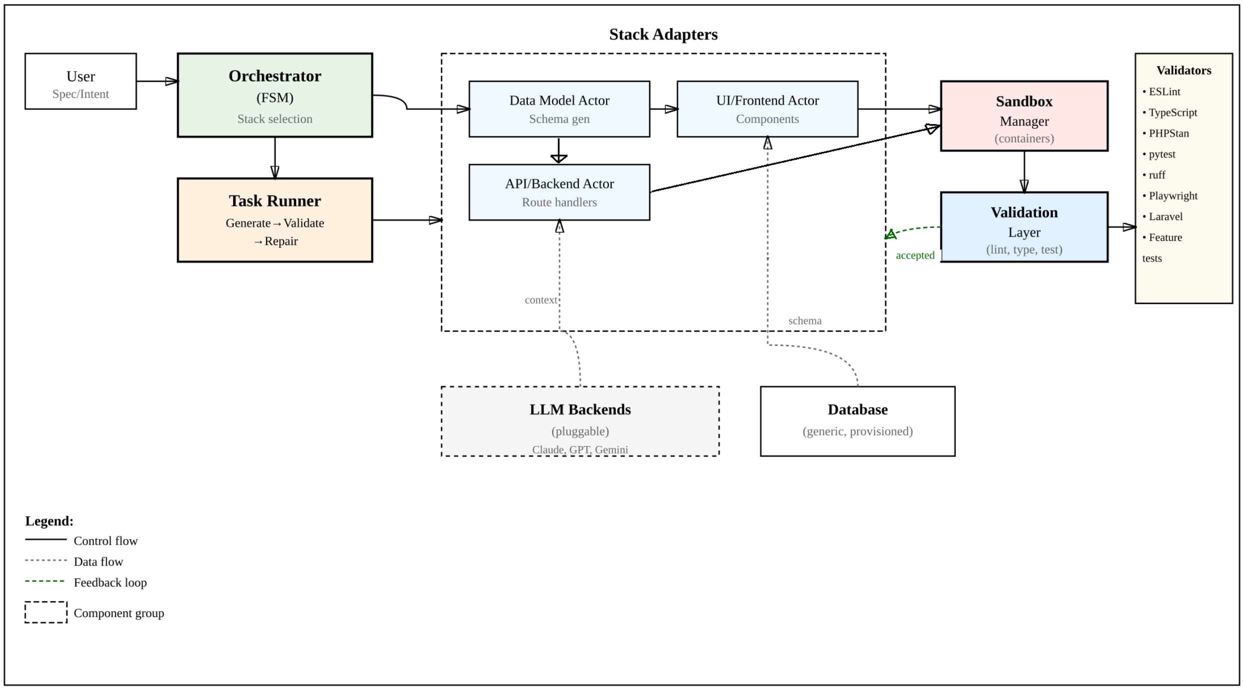

Figure 2: **app.build architecture** expressed through environment scaffolding. The orchestrator plans stages per stack; each sub-task runs in a sandbox, is validated, and only then merged. Continuous Integration/Continuous Deployment (CI/CD) and database provisioning are integrated.

but cannot assess functional correctness or usability. For viability assessment, human evaluators systematically test 30 representative applications using a structured 6-criteria rubric (Section 4.4), verifying that generated apps correctly implement requested functionality, handle edge cases, and meet production quality standards. This manual assessment is essential because automated metrics alone cannot detect integration bugs, incorrect business logic, or poor UX that would block production deployment.

Viability vs. quality distinction. We separate binary viability ($V \in \{0, 1\}$) from continuous quality ($Q \in [0, 10]$). Viability requires only that an application boots successfully and demonstrates basic prompt correspondence—a minimal threshold for deployment consideration. Quality scoring evaluates correctness, completeness, error handling, and code maintainability through systematic human assessment. This distinction reflects industrial practice: automated gates filter non-viable candidates before human review evaluates deployment readiness.

Production-scale validation. Beyond controlled experiments, we validate the framework through production deployment metrics. Since June 2025, the system has generated over 3000 user applications with peak usage exceeding 220 apps/day (Section 5). Production data provides ecological validity showing the framework operates reliably with diverse real-world requirements beyond our curated test set.

Cost-performance tradeoffs. All experiments track token consumption (input/output) and API costs to quantify validation overhead. We report both total cost per generation cohort and cost-per-viable-app to reveal true economics: validation may increase upfront costs but reduces the effective cost of successful outcomes by filtering failures early.

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 Create, Read, Update, Delete (CRUD); high: multi-entity/custom logic). See the full list of prompts in Appendix A.

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 through AB-06, defined in Section 4.4)
- Quality scores (Q, 0–10) using the rubric in Section 4.4
- Model/cost comparisons where applicable

4.3 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.

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.

4.4 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). The complete assessor handbook with detailed grading criteria and example graded applications is publicly available [Kniazev et al., 2025].

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 (Create Functionality), AB-04 (View/Edit Operations), AB-05 (Clickable Sweep): PASS = 1.0, WARN = 0.5, FAIL = 0.0
- AB-06 (Performance): continuous metric normalized to $[0, 1]$

5 Results

5.1 Production Deployment and Community Adoption

The app.build framework has been deployed in production since June 2025, demonstrating real-world viability beyond controlled experiments. The open-source repository (`https://github.com/neondatabase/appdotbuild-age`

Table 2: Check Weights and Definitions Used in Scoring

Check ID	Description	Weight	Notes
AB-01	Boot	1/6	Hard gate for V
AB-02	Prompt Correspondence	1/6	Hard gate for V
AB-03	Create Functionality	1/6	
AB-04	View/Edit Operations	1/6	
AB-05	Clickable Sweep	1/6	
AB-06	Performance	1/6	Normalized to $[0, 1]$

See Section 4.4 for rubric details. All weights equal after NA renormalization. AB-01 and AB-02 are hard gates for viability (V).

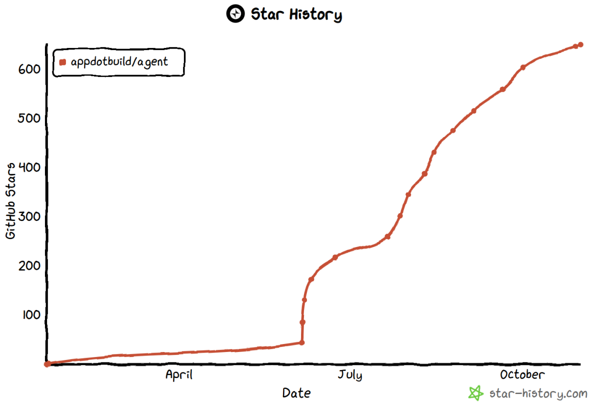

Figure 3: GitHub star growth trajectory for appdotbuild/agent repository showing 13x growth over 5 months (May-October 2025), with inflection point in June 2025 coinciding with production deployment launch. The sustained upward trajectory through October 2025 indicates genuine practitioner adoption rather than transient interest. Data from star-history.com.

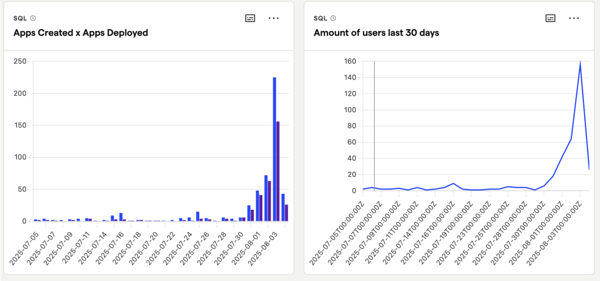

Figure 4: Production usage metrics demonstrating real-world deployment scale. Left: Daily application creation and deployment activity showing peak usage of 220+ apps/day in early August 2025. Right: User growth trajectory over 30 days showing rapid adoption spike coinciding with peak usage period, reaching 160+ active users. Data from production database analytics.

nt) has gained significant community traction with 650 stars and 89 forks as of October 2025, indicating strong practitioner interest in environment-first approaches to agentic code generation.

Figure 3 shows the repository's star growth trajectory, revealing an inflection point in June 2025 when the framework reached production maturity. The repository grew from approximately 50 stars to 650+ stars over five months, representing 13x growth with peak velocity exceeding 100 stars per month during August-September 2025. This organic adoption pattern—characterized by sustained acceleration rather than a single viral spike—suggests the framework addresses genuine practitioner needs.

At peak usage, the platform generated hundreds of applications daily (Figure 4, left panel shows peak of 220+ apps/day in early August 2025), with over 3000 user applications generated during the first 4 months (June-October 2025). The

Table 3: Large-Scale Automated Results Across 300 Experiments

Configuration	n	Success	HC Pass	Cost	Dur.(s)
Baseline (Claude)	30	86.7%	96.7%	$110.20	478
No Lint	30	93.3%	96.7%	$70.49	496
No Playwright	30	83.3%	93.3%	$86.17	463
No Tests	30	93.3%	100%	$71.05	373
Qwen3-480B	90	70.0%	86.7%	$12.68	629
GPT-OSS-120B	90	30.0%	43.3%	$4.55	628

Success = automated healthcheck (AB-01) + template validation (AB-02) passed; excludes template-only apps with zero functionality. HC Pass = healthcheck (AB-01) only; includes both functional apps and non-functional templates. Cost = total for cohort. Dur. = mean per-app duration. Open model experiments used simplified validation pipeline (AB-01 + AB-02 only).

concurrent user growth spike (Figure 4, right panel) demonstrates sustained platform adoption beyond initial experimentation. This production-scale validation complements our controlled experiments: while our systematic evaluation uses 30 prompts with detailed human assessment and 300 experiments with automated metrics, the production deployment provides ecological validity showing the framework operates reliably in uncontrolled real-world conditions with diverse user requirements.

The community adoption metrics (650+ stars, 89 forks) position app.build among actively-used open-source agent frameworks, demonstrating that practitioners value systematic environment scaffolding for production reliability over model-only approaches. The correlation between production deployment launch (June 2025) and rapid community growth validates the industrial relevance of our environment-first approach.

5.2 Two-Tier Evaluation Methodology

Our evaluation combines automated checks with manual viability assessment. We conducted **300 end-to-end generation experiments** across baseline and ablation conditions, collecting objective metrics (success rate, healthcheck pass rate, cost, duration, token usage) for each run. The automated "healthcheck" corresponds to AB-01 (Boot), while full viability assessment requires both AB-01 and AB-02 (Prompt Correspondence) as defined in Section 4.4. These automated checks enable consistent measurement and cost-effectiveness analysis across configurations. For viability validation, we performed **detailed human evaluation on 30 representative prompts** using the AB-check rubric (Section 4.4), providing nuanced assessment of functional correctness and production readiness that automated metrics cannot capture.

This two-tier approach reflects industrial practice: automated metrics enable systematic comparison across model architectures and validation strategies, while human evaluation validates that generated applications actually work for their intended purpose. Automated gates filter obviously broken candidates, but human assessors determine whether apps meet production deployment criteria.

5.3 Automated Validation Results at Scale (n=300)

Table 3 presents aggregated results from 300 automated experiments across all conditions. The baseline configuration (Claude Sonnet 4 with full validation) achieved 86.7% automated success rate at $110.20 total cost for 30 apps. Open-weights models show cost-performance tradeoffs: Qwen3-Coder-480B achieved 70% success at $12.68 total cost, delivering an 8.2x cost reduction per viable app ($0.61 vs $5.01), while validation ablations reveal systematic patterns discussed in subsequent sections.

Key findings from automated metrics: (1) Removing comprehensive validation (no_lint, no_tests) increases automated success by +6.7% but reduces costs by $40, suggesting validators catch real issues at measurable expense. (2) Playwright removal has minimal impact on automated success (-3.3%) while saving $24, indicating E2E brittleness. (3) Open models achieve viable cost-performance tradeoffs for less critical applications.

5.4 Cost and Token Usage Analysis

Detailed telemetry from 300 experiments reveals systematic resource consumption patterns. The baseline configuration (Claude Sonnet 4, full validation) consumed 27.7M input tokens and 1.8M output tokens across 30 apps, averaging

Table 4: Resource Consumption Breakdown by Configuration

Config	In Tok /App	Out Tok /App	Cost /App	Viable Cost
Baseline	923K	60K	$3.67	$5.01
No Lint	531K	50K	$2.35	$2.52
No Playwright	694K	53K	$2.87	$3.45
No Tests	531K	52K	$2.37	$2.54
Qwen3-480B	728K	26K	$0.42	$0.61
GPT-OSS-120B	732K	26K	$0.15	$0.51

Tok/App = tokens per application (K = thousands). Viable Cost = cost per viable app (total cost / viable count). Open models via OpenRouter at reduced rates.

Table 5: Aggregated Evaluation Results for TypeScript/tRPC ($n = 30$)

Metric	Value	Note
Total Apps	30	tRPC stack only
Viability ($V = 1$)	73.3%	22/30 viable
Perfect ($Q = 10$)	30.0%	9/30 perfect
Non-viable ($V = 0$)	26.7%	8/30 failed
Mean Quality	8.78	$V = 1$ apps only

Viability V and quality Q defined in Section 4.4. Perfect = all checks PASS; non-viable = AB-01 (Boot) or AB-02 (Prompt Correspondence) FAIL.

923K input and 60K output tokens per app. This translates to $3.67 per app at standard API rates ($3/M input, $15/M output).

The cost-per-viable-app metric reveals validation overhead: baseline achieves viability at $5.01 per app (22/30 viable), while removing unit tests reduces this to $2.54 (24/30 viable) despite similar per-generation costs. This indicates that comprehensive validation both increases initial costs and filters marginal cases, raising the effective cost per successful outcome.

Open-weights models demonstrate dramatic cost advantages: Qwen3-Coder-480B generates viable apps at $0.61 each (8.2x cheaper than Claude baseline), though at reduced success rates (70% vs 86.7%). For large-scale deployment or less critical applications, this represents a viable engineering tradeoff.

Token efficiency varies by validation configuration: linting and unit tests consume substantial input tokens through multi-round validation cycles (baseline: 923K vs no_tests: 531K), suggesting that validation rigor directly impacts computational cost. The output token counts remain relatively stable (50K-60K), indicating that validation affects iteration count more than generation verbosity.

5.5 Detailed Quality Assessment (Human Evaluation, n=30)

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 ($V = 1$, $n = 22$), quality averaged 8.78 with 77.3% achieving $Q \geq 9$. Non-viability ($V = 0$) arises from smoke test failures or missing artifacts.

5.6 Open vs Closed Model Performance

We evaluated Claude Sonnet 4 against two open-weights models using the TypeScript/tRPC stack with simplified validation pipeline (AB-01 Boot + AB-02 Prompt Correspondence only) 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.

Table 6: Check-Specific Outcomes Across $n = 30$ Tasks

Check	Pass	Warn	Fail	NA
AB-01 (Boot)	25	2	3	0
AB-02 (Prompt Correspondence)	19	3	5	3
AB-03 (Create Functionality)	22	2	0	6
AB-04 (View/Edit Operations)	17	1	1	11
AB-05 (Clickable Sweep)	20	4	1	5
AB-06 (Performance)	23	3	0	4

See Section 4.4 for grading criteria. NA = not applicable for prompt type. Pass rates (excl. NA): AB-01 (Boot): 83.3%, AB-02 (Prompt Correspondence): 70.4%, AB-03 (Create Functionality): 91.7%, AB-04 (View/Edit Operations): 89.5%, AB-05 (Clickable Sweep): 80.0%, AB-06 (Performance): 88.5%.

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 8.2x cost reduction per viable app for 19% performance loss represents a viable tradeoff for many production scenarios.

Template detection reveals inflated success rates. Our evaluation uncovered a critical issue: many apps that pass healthcheck validation are non-functional "Under Construction" templates with zero functionality. For GPT-OSS-120B, 43.3% passed AB-01 (Boot), but only 30.0% passed both AB-01 and AB-02 (Prompt Correspondence), with manual inspection revealing that a substantial portion of bootable apps were generic placeholder scaffolding rather than functional implementations. In contrast, Qwen3-Coder-480B-A35B achieved 86.7% healthcheck pass and 70.0% true success, showing better prompt adherence with fewer template generations. *The Success column in Table 3 reports only viable non-template apps that passed AB-02 validation, while HC Pass includes all bootable apps regardless of functionality*, demonstrating why boot checks alone are insufficient for evaluating code generation quality.

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). AB-01 (Boot) 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.

5.7 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, systematically removing one validation component while keeping others intact. The baseline configuration (all validation layers active) achieved 73.3% viability with mean quality score $Q = 8.06$.

Unit test removal trades quality for apparent viability. Disabling backend handler tests increased viability to 80.0% (+6.7 pp) but reduced mean quality to $Q = 7.78$ (-0.28). This paradox reflects that unit tests catch critical CRUD errors: apps boot successfully without them but fail on data operations. AB-04 (View/Edit Operations) pass rates dropped from 90% to 60%, indicating that backend validation prevents functional regressions that smoke tests cannot detect. The increased viability metric reflects false positives: apps that appear viable but contain latent data integrity bugs.

Linting removal shows mixed effects with modest gains. Removing ESLint checks increased viability to 80.0% (+6.7 pp) and slightly improved quality to $Q = 8.25$ (+0.19), suggesting some lint rules may be overly restrictive. However, AB-03 (Create Functionality) dropped 8.3 pp and AB-04 (View/Edit) dropped 7.6 pp, indicating ESLint does catch legitimate structural issues. The net positive quality score indicates that strict linting can reject valid alternative implementations, though the effect size is small.

E2E test removal significantly improves outcomes (a negative result). Removing Playwright tests produced the strongest effect: viability increased to 90.0% (+16.7 pp) with quality improving to $Q = 8.62$ (+0.56). AB-02 (Prompt Correspondence) improved +11.8 pp and AB-05 (Clickable Sweep) improved +5.7 pp. This counterintuitive result indicates that E2E tests introduce more false rejections than they catch real defects. Manual inspection of failed Playwright runs reveals three root causes of brittleness: (1) *Over-specified selectors*: tests hardcode element IDs or CSS classes that models generate variably across runs, causing spurious failures despite functional correctness; (2) *Race conditions*: E2E assertions check UI state before async data fetching completes, rejecting apps that function correctly under human-realistic interaction delays; (3) *False negatives from implementation variance*: models generate semantically equivalent but structurally different UIs (e.g., modal dialogs vs inline forms) that satisfy requirements but fail brittle assertions expecting specific Document Object Model (DOM) structure. This finding suggests that comprehensive E2E suites designed for deterministic codebases are poorly suited to probabilistic code generation, where implementation details vary across generation runs while functional correctness remains stable.

5.8 Synthesis: Optimal Validation Strategy

Our ablation results reveal systematic trade-offs between validation rigor and success metrics. Unit/handler tests prove essential for data integrity: removing them increases perceived viability (+6.7 pp) but causes real functional regressions, particularly in AB-04 (View/Edit Operations, -30 pp drop). ESLint provides modest value with measurable false positives; the small net quality gain (+0.19) and mixed per-check effects suggest selective application of rules targeting actual errors rather than style preferences. E2E tests currently cause more harm than good, with removal yielding +16.7 pp viability and +0.56 quality improvement, indicating these tests reject too many working applications due to brittleness rather than correctness issues.

These findings suggest a pragmatic validation architecture optimized for probabilistic code generation: *retain* lightweight smoke tests (boot verification and primary route checks) combined with backend unit tests for CRUD operations, which together provide high-confidence quality gates; *refine* static analysis to focus on structural correctness and known anti-patterns while relaxing stylistic rules that reject valid implementations; *replace* comprehensive E2E suites with targeted integration tests covering only critical user paths, avoiding brittle assertions on implementation details. This configuration balances defect detection against false rejection rates. For scenarios where quality requirements dominate cost constraints, comprehensive validation including strict E2E tests remains viable, trading lower automated success rates (-16.7 pp) for stronger guarantees about production correctness.

5.9 Failure Mode Analysis

We systematically categorize observed failure modes and map them to AB check coverage. *Boot/Load failures* (template placeholders, incomplete artifacts) are reliably caught by AB-01 (Boot) during server startup. *Prompt correspondence failures* (generic templates from generation failures) are detected by AB-02 when human evaluators identify that apps render successfully but implement incorrect functionality. *Content Security Policy (CSP) restrictions* (blocked images or media) are partially caught by AB-05 (Clickable Sweep) through visual inspection, though automated checks miss these unless they break critical flows. *UI interaction defects* (unbound event handlers, non-functional controls) are exposed by AB-05 and AB-03 (Create Functionality) through manual interaction; automated E2E tests theoretically detect these but suffer from brittleness as documented in Section G. *State/integration defects* (data persistence failures across refresh, broken filters, authentication issues) are caught by AB-04 (View/Edit Operations), though backend unit tests provide only partial coverage and miss client-side state management bugs. Finally, *component misuse* (runtime exceptions from incorrect composition) is detected by AB-01 or AB-05 depending on timing, with TypeScript catching some violations at compile time while others surface only during manual testing.

Validation blind spots and proposed upgrades. Our AB check suite exhibits systematic coverage gaps across five dimensions. First, *accessibility violations* go undetected; apps may function for able-bodied users but fail Web Content Accessibility Guidelines (WCAG) standards. We propose integrating *axe-core* for automated accessibility auditing. Second, *mobile responsiveness* remains unchecked as all validation uses desktop viewports; extending smoke tests to include mobile viewport validation would address this. Third, *performance degradation* under load is not measured; AB-06 assesses only initial load time, missing runtime performance issues. *Lighthouse* performance budgets could provide continuous monitoring. Fourth, *security vulnerabilities* beyond CSP violations (Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), injection attacks) are not detected; static analysis via *semgrep* with OWASP rulesets would provide defense-in-depth. Fifth, *data consistency* in concurrent scenarios and transaction isolation remain untested; implementing multi-client test scenarios would validate state consistency guarantees. These proposed upgrades would strengthen production readiness while maintaining the environment scaffolding paradigm of deterministic, automated validation.

5.10 Prompt Complexity and Success Rate

We categorize prompts by complexity: *low* (static/single-page UI), *medium* (single-entity CRUD), *high* (multi-entity workflows with custom logic), and analyze success patterns. Medium-complexity CRUD prompts achieve highest quality ($Q = 9$–10), reflecting strong scaffolding alignment with data models and handler patterns. Low-complexity UI prompts prove non-uniformly easy: several failed AB-02 (Prompt Correspondence) by generating generic templates rather than task-specific implementations. High-complexity prompts exhibit lower viability due to interaction wiring and state-consistency issues surfaced by AB-04 (View/Edit Operations) and AB-05 (Clickable Sweep).

Trajectory patterns reveal scaffolding trade-offs. Manual inspection of generation logs exposes distinct model reasoning patterns across complexity levels. For medium-complexity CRUD tasks, environment scaffolding provides effective guidance: structured schema definitions constrain data models to valid patterns, type-safe API contracts prevent interface mismatches, and immediate validator feedback enables targeted repairs within 1–2 iterations. Representative trajectory (`plant-care-tracker`): schema generation passed immediately; API handlers failed unit tests due to missing foreign key constraints, triggering repair with validator feedback; UI generation succeeded on first attempt using type inference from validated API. This demonstrates the intended scaffolding workflow.

Conversely, for low-complexity UI-only tasks, scaffolding introduces bias: models over-engineer solutions using CRUD templates when simple static pages suffice, causing AB-02 failures. The framework's database-first workflow assumes backend requirements even when unnecessary. Failure example (`birthday-wish-app`): model generated full API routes and database schema for a static greeting card, achieving technical correctness but failing prompt correspondence. This reveals a structural limitation where strong scaffolding can constrain creativity for tasks outside the target domain.

High-complexity tasks expose scaffolding's context limits. Multi-entity relationships and custom business logic require models to maintain architectural context across multiple validation cycles spanning frontend-backend boundaries. Failures cluster in AB-04 (state management) when models lose track of data flow requirements. Validation feedback successfully catches syntax errors but provides limited architectural guidance, forcing models into trial-and-error that exhausts retry budgets before convergence. This suggests that scaffolding effectiveness degrades as task complexity increases beyond the structured patterns the environment encodes.

5.11 Threats to Validity & 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 evaluation focused exclusively on the TypeScript/tRPC stack because it is the most mature and well-supported stack in our framework, enabling the most reliable assessment of environment scaffolding effectiveness. We chose web application generation as the evaluation domain because it represents the most widely adopted use case among both closed-source commercial generators and academic end-to-end code generation systems, providing a relevant benchmark for comparison. 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.

Benchmark applicability. We do not evaluate on SWE-bench [Jimenez et al., 2024] or HumanEval [Chen et al., 2021] because these benchmarks target fundamentally different tasks: repository-level bug fixing and function-level code completion, respectively. SWE-bench evaluates patch correctness against existing test suites in mature codebases, while our work generates complete applications requiring multi-layered validation (schema correctness, API contracts, UI functionality, integration testing). The evaluation methodology mismatch is structural: existing benchmarks assume deterministic test oracles, whereas greenfield application generation requires human assessment of prompt correspondence, as our template detection findings demonstrate—53.6% of GPT-OSS-120B outputs passed automated healthchecks but were non-functional placeholders. Our ablation studies provide baseline comparisons that demonstrate environment scaffolding's value: configurations with validation layers disabled show that removing backend tests increases apparent viability (+6.7pp) but causes CRUD correctness to drop 30pp, revealing that comprehensive validation trades automated success metrics for functional correctness.

5.12 Ethics & 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.

Our results suggest that for CRUD-oriented web applications, structured environment scaffolding complements model capability in achieving production reliability. Through systematic validation, stack-specific orchestration, and iterative repair, app.build demonstrates how probabilistic language models can be guided toward dependable software generation within constrained domains.

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.

For production-oriented agent systems in structured domains, environment engineering with targeted validation layers offers a complementary path to scaling model capability, providing measurable improvements in reliability while managing cost. As model capabilities continue to advance, the systematic integration of validation and iterative repair remains essential for bridging the gap between probabilistic generation and deterministic production requirements.

Data and Artifact Availability

To support reproducibility, we release the following artifacts:

- **Framework source code:** Open-source implementation at `https://github.com/neondatabase/appdotbuild-agent` (experiments used commit `e362615` from August 14, 2025)
- **Evaluation dataset:** Generated applications, configuration files, and generation logs at `https://github.com/keugenek/app.build-publications/tree/main/analysis/dataset` (evaluation conducted August 19-20, 2025)
- **Analysis and results:** Experimental results, validation data, and one-command reproduction script (`run_analysis_and_compare.sh`) at `https://github.com/keugenek/app.build-publications/tree/main/analysis/results`
- **Evaluation assessor handbook:** Detailed grading criteria, example graded applications, and raw human assessment scores [Kniazev et al., 2025]
- **Prompt dataset:** Complete set of 30 evaluation prompts with complexity ratings (Appendix A)
- **Docker environment:** Reproducible sandbox configurations included in the framework repository

Acknowledgments

This submission is prepared in collaboration between Databricks (app.build team) and THWS University of Applied Sciences Würzburg-Schweinfurt (CAIRO). We thank the app.build community for their contributions and feedback which have been invaluable in shaping this work. Special thanks to Databricks executive team for supporting the open-source initiative and providing resources for this research. We also thank David Gomes for advocating for the community-centered vision that guided this project.

References

Anthropic. Claude 4 model card. `https://www-cdn.anthropic.com/4263b940cabb546aa0e3283f35b686f4f3b2ff47.pdf`, 2024. Model card for Claude 4.

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`.

Sarah Fakhoury, Saikat Chakraborty, Miltiadis Allamanis, and Shuvendu K. Lahiri. Llm-based test-driven interactive code generation: User study and empirical evaluation, 2024. URL https://arxiv.org/abs/2404.10100.

Linyuan Gong, Sida Wang, Mostafa Elhoushi, and Alvin Cheung. Ast-t5: Structure-aware pretraining for code generation and understanding, 2024. URL https://arxiv.org/abs/2401.03003.

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.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024.

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.

Evgenii Kniazev, Arseny Kravchenko, and Igor Rekun. app.build evaluation assessor handbook. https://docs.google.com/spreadsheets/d/1hvx2IpySdcOP8VfibnWZr3unyDinJ79OhQ5lfqwWfp0/, 2025. Evaluation rubric, grading criteria, and example graded applications.

Evgeny G Kniazev. Automated source code changes classification for effective code review and analysis. In *Proceedings of the Spring/Summer Young Researchers' Colloquium on Software Engineering*. Institute for System Programming of the Russian Academy of Sciences, 2008.

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.

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.

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. Openhands: An open platform for ai software developers as generalist agents, 2024. URL https://arxiv.org/abs/2407.16741. Published at ICLR 2025.

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`.

A Prompt Dataset

This appendix provides the complete specification of evaluation prompts used in our experiments. Table 7 lists all 30 prompts with their complexity classifications, covering low-complexity UI tasks, medium-complexity single-entity CRUD applications, and high-complexity multi-entity workflows with custom business logic.

Table 7: Complete Prompt Dataset Used in Evaluation ($n = 30$)

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 Spaced Repetition System (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

Note. Dataset details in Section 4.2. Complexity rubric in Section 5.10: Low (static/single-page UI), Medium (single-entity CRUD), High (multi-entity/custom logic).