app.build: A Production Framework for Scaling Agentic Prompt-to-App Generation with Environment Scaffolding

1st Evgenii Kniazev*
Databricks
Email: eng-appbuild@databricks.com

2nd Arseny Kravchenko*
Databricks
Email: eng-appbuild@databricks.com

3rd Igor Rekun*
Databricks
Email: eng-appbuild@databricks.com

4th James Broadhead
Databricks
Email: eng-appbuild@databricks.com

5th Nikita Shamgunov
Databricks
Email: eng-appbuild@databricks.com

6th Pranav Sah
THWS University of Applied Sciences, Würzburg-Schweinfurt (CAIRO)

7th Pratik Nichite
THWS University of Applied Sciences, Würzburg-Schweinfurt (CAIRO)

8th Ivan Yamshchikov
THWS University of Applied Sciences, Würzburg-Schweinfurt (CAIRO)

***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.

***Index Terms*—software engineering, code generation, LLM agents, validation, environment scaffolding**

Code available at https://github.com/appdotbuild/agent/
*These authors contributed equally to this work.

A. The Production Reliability Gap

Engineering teams increasingly experiment with LLM agents to synthesize full-stack web applications, yet production deployment remains blocked by fundamental reliability issues. While research systems demonstrate impressive capabilities on isolated benchmarks—HumanEval [1] leaders achieve 90%+ pass rates on function-level tasks, and LiveCodeBench [2] 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 [3].

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 4.5 achieve high success rates but at significant costs [TODO:proof!]. 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 [4], [5] note the field requires a shift from model-centric to environment-centric design.

B. 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.

a) 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 → API → 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.

b) 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 I illustrate this architectural difference.

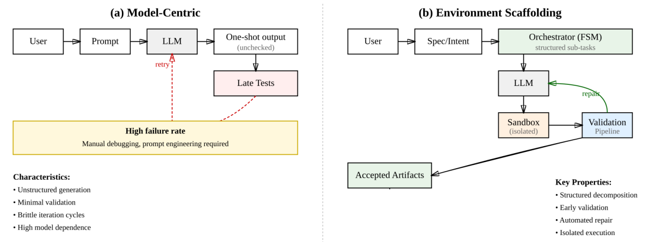

Fig. 1. **Environment scaffolding vs. model-centric generation.** ES wraps the model with a finite, validated workflow that catches errors early and repairs them before proceeding.

TABLE I
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

C. 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.
- **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 balances statistical power with nuanced quality assessment.
- **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 E2E browser tests introduce brittleness.

D. Background and 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** [6] established the evaluation standard with 2,294 real GitHub issues from 12 Python projects. Recent agents demonstrate that environment design rivals model capability: **OpenHands** [7], published at ICLR 2025, achieves 53% on SWE-bench Verified through an open platform for generalist agents with agent-computer interfaces. **SWE-agent** [8] showed 12.5% pass@1 through careful interface design rather than model improvements. Contemporary 2024 agents include **AutoCodeRover** [9], which combines LLMs with spectrum-based fault localization (19% on SWE-bench, $0.43 per issue), and **Agentless** [10], 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 [11], modern frameworks now integrate validation at multiple layers. Test-driven approaches [12] achieve 45.97% absolute improvement in pass@1 through interactive generation with dynamic test feedback. **AST-based validation** [13] provides structural guarantees, with AST-T5 outperforming CodeT5 by 2–3 points through structure-aware pretraining. Tree search methods [14] demonstrate that scaling compute through iterative refinement and parallel branches can significantly improve success rates. Multi-agent systems [15] 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.

I. Industrial Context & System

A. 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.

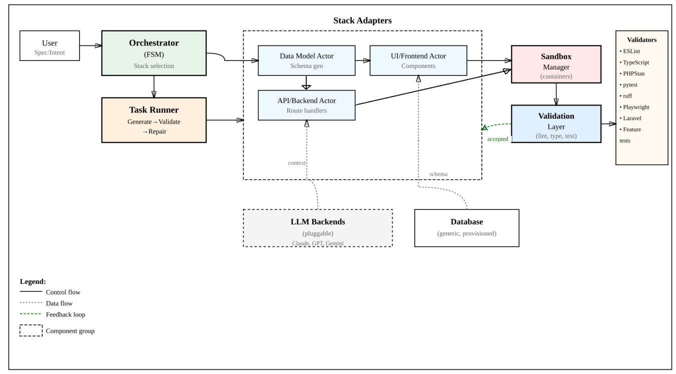

Fig. 2. **app.build architecture** expressed through environment scaffolding. The orchestrator plans stages per stack; each sub-task runs in a sandbox, is validated, and only then merged. CI/CD and DB provisioning are integrated.

B. 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.

II. Experimental Setup

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

A. Evaluation Framework

B. 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 III-L.

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 II-D
- Model/cost comparisons where applicable

C. 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 [16] and GPT OSS 120B [17] 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.

D. 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):

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

Quality (0–10):

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

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 $[0, 1]$

TABLE II
CHECK WEIGHTS AND DEFINITIONS USED IN SCORING

Check ID	Description	Weight	Notes
AB-01	Boot & Home	1/6	Hard gate for V
AB-02	Prompt Corr.	1/6	Hard gate for V
AB-03	Create Func.	1/6	
AB-04	View/Edit Ops	1/6	
AB-06	Clickable Sweep	1/6	
AB-07	Performance	1/6	Normalized to $[0, 1]$

See Section II-D for rubric details. All weights equal after NA re-normalization.

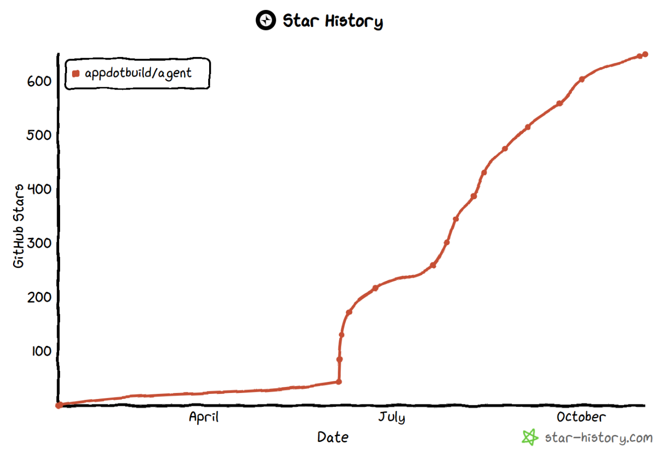

Fig. 3. GitHub star growth trajectory for appdotbuild/agent repository showing 13x growth over 5 months (May-October 2025), with inflection point in June 2025 coinciding with production deployment launch. The sustained upward trajectory through October 2025 indicates genuine practitioner adoption rather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

III. Results

A. 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/appdotbuild/agent/) 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 concurrent user growth spike (Figure 4, right panel)

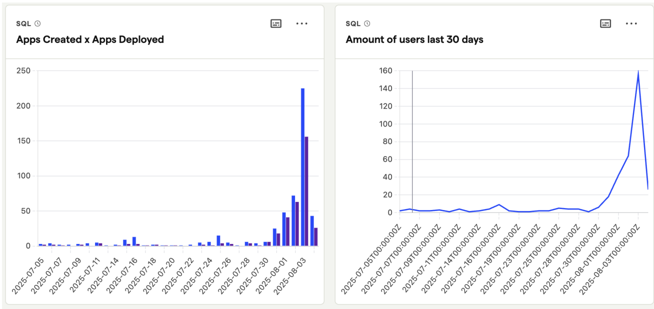

Fig. 4. Production usage metrics demonstrating real-world deployment scale. Left: Daily application creation and deployment activity showing peak usage of 220+ apps/day in early August 2025. Right: User growth trajectory over 30 days showing rapid adoption spike coinciding with peak usage period, reaching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

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.

B. Two-Tier Evaluation Methodology

Our evaluation combines large-scale automated validation with detailed human quality 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. This automated tier provides statistical power and cost-effectiveness analysis. For quality validation, we performed **detailed human evaluation on 30 representative prompts** using the AB-check rubric (Section II-D), providing nuanced assessment of viability and functional correctness that automated metrics cannot capture.

This two-tier approach balances scale with depth: automated metrics (n=300) establish broad patterns and enable rigorous ablation studies, while human evaluation (n=30) validates that automated success correlates with actual application quality. The methodology reflects industrial practice where automated gates filter candidates before human review.

C. Automated Validation Results at Scale (n=300)

Table III presents aggregated results from 300 automated experiments across all conditions. The baseline configuration

TABLE III
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 + template validation passed. HC Pass = healthcheck only. Cost = total for cohort. Dur. = mean per-app duration. Open model experiments used simplified validation (smoke tests only).

TABLE IV
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.

(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.

D. 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 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.

TABLE V
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 II-D. Perfect = all checks PASS; non-viable = AB-01 or AB-02 FAIL.

TABLE VI
CHECK-SPECIFIC OUTCOMES ACROSS $n = 30$ TASKS

Check	Pass	Warn	Fail	NA
AB-01 (Boot)	25	2	3	0
AB-02 (Prompt)	19	3	5	3
AB-03 (Create)	22	2	0	6
AB-04 (View/Edit)	17	1	1	11
AB-06 (Clickable)	20	4	1	5
AB-07 (Perf.)	23	3	0	4

See Section II-D for grading criteria. NA = not applicable. Pass rates (excl. NA): AB-01: 83.3%, AB-02: 70.4%, AB-03: 91.7%, AB-04: 89.5%, AB-06: 80.0%, AB-07: 88.5%.

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 87%). 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.

E. 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 ($V = 1$, $n = 21$), quality averaged 8.78 with 77.3% achieving $Q \geq 9$. Non-viability ($V = 0$) arises from smoke test failures or missing artifacts.

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

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.

G. 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.

H. 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.

I. 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.

J. 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.

K. 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 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.

L. 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 [18], 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.

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

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[2] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, "Livecodebench: Holistic and contamination free evaluation of large language models for code," 2024.

[3] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," 2023. [Online]. Available: https://arxiv.org/abs/2305.01210

[4] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," 2024. [Online]. Available: https://arxiv.org/abs/2406.00515

[5] D. G. Paul, H. Zhu, and I. Bayley, "Benchmarks and metrics for evaluations of code generation: A critical review," 2024. [Online]. Available: https://arxiv.org/abs/2406.12655

[6] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" 2024. [Online]. Available: https://arxiv.org/abs/2310.06770

[7] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig, "Openhands: An open platform for ai software developers as generalist agents," 2024, published at ICLR 2025. [Online]. Available: https://arxiv.org/abs/2407.16741

[8] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," 2024. [Online]. Available: https://arxiv.org/abs/2405.15793

[9] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," 2024. [Online]. Available: https://arxiv.org/abs/2404.05427

[10] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying llm-based software engineering agents," 2024. [Online]. Available: https://arxiv.org/abs/2407.01489

[11] E. 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.

[12] S. Fakhoury, S. Chakraborty, M. Allamanis, and S. K. Lahiri, "Llm-based test-driven interactive code generation: User study and empirical evaluation," 2024. [Online]. Available: https://arxiv.org/abs/2404.10100

[13] L. Gong, S. Wang, M. Elhoushi, and A. Cheung, "Ast-t5: Structure-aware pretraining for code generation and understanding," 2024. [Online]. Available: https://arxiv.org/abs/2401.03003

[14] D. Li, S. Cao, C. Cao, X. Li, S. Tan, K. Keutzer, J. Xing, J. E. Gonzalez, and I. Stoica, "S*: Test time scaling for code generation," 2025. [Online]. Available: https://arxiv.org/abs/2502.14382

[15] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "Metagpt: Meta programming for a multi-agent collaborative framework," 2024. [Online]. Available: https://arxiv.org/abs/2308.00352

[16] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, C. Zheng, D. Liu, F. Zhou, F. Huang, F. Hu, H. Ge, H. Wei, H. Lin, J. Tang, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Zhou, J. Lin, K. Dang, K. Bao, K. Yang, L. Yu, L. Deng, M. Li, M. Xue, M. Li, P. Zhang, P. Wang, Q. Zhu, R. Men, R. Gao, S. Liu, S. Luo, T. Li, T. Tang, W. Yin, X. Ren, X. Wang, X. Zhang, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Zhang, Y. Wan, Y. Liu, Z. Wang, Z. Cui, Z. Zhang, Z. Zhou, and Z. Qiu, "Qwen3 technical report," 2025. [Online]. Available: https://arxiv.org/abs/2505.09388

[17] OpenAI, :, S. Agarwal, L. Ahmad, J. Ai, S. Altman, A. Applebaum, E. Arbus, R. K. Arora, Y. Bai, B. Baker, H. Bao, B. Barak, A. Bennett, T. Bertao, N. Brett, E. Brevdo, G. Brockman, S. Bubeck, C. Chang, K. Chen, M. Chen, E. Cheung, A. Clark, D. Cook, M. Dukhan, C. Dvorak, K. Fives, V. Fomenko, T. Garipov, K. Georgiev, M. Glaese, T. Gogineni, A. Goucher, L. Gross, K. G. Guzman, J. Hallman, J. Hehir, J. Heidecke, A. Helyar, H. Hu, R. Huet, J. Huh, S. Jain, Z. Johnson, C. Koch, I. Kofman, D. Kundel, J. Kwon, V. Kyrylov, E. Y. Le, G. Leclerc, J. P. Lennon, S. Lessans, M. Lezcano-Casado, Y. Li, Z. Li, J. Lin, J. Liss, Lily, Liu, J. Liu, K. Lu, C. Lu, Z. Martinovic, L. McCallum, J. McGrath, S. McKinney, A. McLaughlin, S. Mei, S. Mostovoy, T. Mu, G. Myles, A. Neitz, A. Nichol, J. Pachocki, A. Paino, D. Palmie, A. Pantuliano, G. Parascandolo, J. Park, L. Pathak, C. Paz, L. Peran, D. Pimenov, M. Pokrass, E. Proehl, H. Qiu, G. Raila, F. Raso, H. Ren, K. Richardson, D. Robinson, B. Rotsted, H. Salman, S. Sanjeev, M. Schwarzer, D. Sculley, H. Sikchi, K. Simon, K. Singhal, Y. Song, D. Stuckey, Z. Sun, P. Tillet, S. Toizer, F. Tsimpourlas, N. Vyas, E. Wallace, X. Wang, M. Wang, O. Watkins, K. Weil, A. Wendling, K. Whinnery, C. Whitney, H. Wong, L. Yang, Y. Yang, M. Yasunaga, K. Ying, W. Zaremba, W. Zhan, C. Zhang, B. Zhang, E. Zhang, and S. Zhao, "gpt-oss-120b & gpt-oss-20b model card," 2025.

[18] V. Babushkin and A. Kravchenko, *Machine Learning System Design with End-to-End Examples*. Manning Publications, 2025.

APPENDIX: PROMPT DATASET

TABLE VII
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 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 II-B. Complexity rubric in Section III-J: Low (static/single-page UI), Medium (single-entity CRUD), High (multi-entity/custom logic).