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

Anonymous Authors
Anonymous Institution
Anonymous City, Country
anonymous@example.com

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

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

I. Introduction

A. The Production Reliability Gap

While AI coding agents demonstrate impressive capabilities on standard benchmarks of isolated tasks like HumanEval [1] and MBPP [2], relying on them to build production-ready applications without human supervision remains infeasible. Recent repository-level systems such as Devin [3] and SWE-agent [4] represent significant advances, yet their performance on real-world software engineering tasks reveals a substantial gap between research benchmarks and production requirements.

This gap manifests across multiple dimensions. Function-level benchmarks like HumanEval evaluate isolated code generation but fail to capture system-level concerns including error handling, integration complexity, and production constraints [5]. Even state-of-the-art systems like AutoCodeRover, achieving 19% efficacy on SWE-bench at $0.43 per issue [6], demonstrate that raw model capability alone is insufficient for reliable automated software development.

The core challenge lies in treating LLMs as standalone systems rather than components requiring structured environments. Current approaches predominantly focus on making models "smarter" via either training or prompt engineering, but this paradigm fails to address fundamental reliability issues inherent in probabilistic generation. Recent surveys [7], [8] 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) Principles.:

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

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

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.

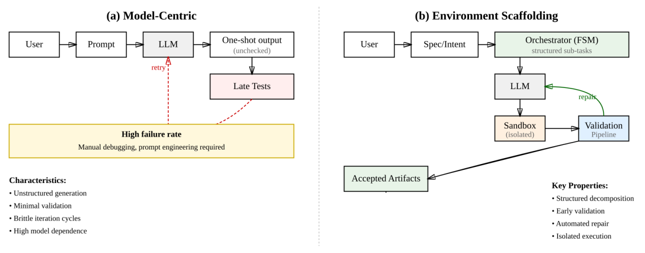

Fig. 1. **Environment scaffolding vs. model-centric generation.** ES wraps the model with a finite, validated workflow that catches errors early and repairs them before proceeding.

TABLE I
ENVIRONMENT SCAFFOLDING (ES) VS. MODEL-CENTRIC GENERATION

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

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

II. BACKGROUND AND RELATED WORK

A. Agentic Software Engineering

The evolution of AI coding agents has progressed from simple code completion to autonomous software engineering systems capable of repository-level modifications. **SWE-bench** [9] established the gold standard for evaluating repository-level understanding with 2,294 real GitHub issues from 12 Python projects. The accompanying **SWE-agent** [4] demonstrated that custom agent-computer interfaces significantly enhance performance, achieving 12.5% pass@1 through careful interface design rather than model improvements.

Repository-level agents have emerged as a distinct research direction. **WebArena** [10] revealed that even GPT-4 achieves only 14.41% success versus 78.24% human performance in realistic environments, demonstrating that environment design matters more than model capability. **GAIA** [11] reinforces this with 92% human versus 15% GPT-4 performance on practical tasks. **AutoCodeRover** [6] combines LLMs with spectrum-based fault localization, achieving 19% efficacy on SWE-bench at $0.43 per issue. More recently, **Agentless** [12] challenged complex agent architectures with a simple three-phase process (localization, repair, validation) achieving 32% on SWE-bench Lite at $0.70 cost, suggesting that sophisticated architectures may not always improve performance.

Multi-agent systems have consistently outperformed single-agent approaches. **AgentCoder** [13] employs a three-agent architecture (Programmer, Test Designer, Test Executor) achieving 96.3% pass@1 on HumanEval with GPT-4, compared to 71.3% for single-agent approaches. **MapCoder** [14] extends this with four specialized agents replicating human programming cycles, achieving 93.9% pass@1 on HumanEval and 22.0% on the challenging APPS benchmark. **MetaGPT** [15] demonstrates role-based agents communicating through structured documents, achieving 85.9% pass@1 on HumanEval with 100% task completion on software development tasks.

B. Production Quality in Generated Code

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

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

C. Tree Search

Tree search enhances LLM-based solutions and serves as a way to increase compute budget beyond internal model reasoning token budget. The closest approach is used by Li et al. in S* Scaling [16] by combining iterative feedback with parallel branches taking different paths toward solving the problem. Sampling more trajectories increases success rate significantly, which is evident by difference in pass@1 and pass@3 often by 30% or more.

D. Runtime Isolation and Scaling

Sandboxing is a cornerstone due to web applications requiring much more elaborate testing than running unit tests.

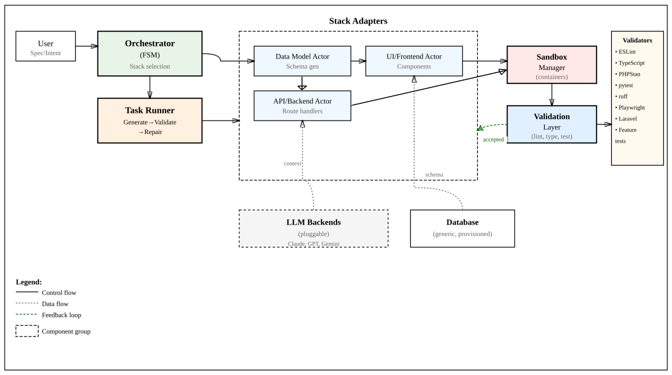

Fig. 2. **app.build architecture** expressed through environment scaffolding. The orchestrator plans stages per stack; each sub-task runs in a sandbox, is validated, and only then merged. CI/CD and DB provisioning are integrated.

It includes setup and teardown of databases and browser emulation. For parallel scaling, we use Dagger.io for its caching capabilities and Docker compatibility.

III. Problem Setup and Method

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.

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). Per-stage validators are stack-aware (e.g., ESLint+TypeScript and Playwright for tRPC; PHPStan and feature tests for Laravel; `pytest`/`ruff`/`pyright` for Python), and the platform provisions managed Postgres databases and CI/CD hooks.

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

IV. 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 A.

C. Metrics

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

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

D. 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 [17] and GPT OSS 120B [18] 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.

E. 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).

TABLE II
CHECK WEIGHTS AND DEFINITIONS USED IN SCORING (SEE RUBRIC IN SECTION IV-E). ALL CHECKS SHARE EQUAL WEIGHT AFTER NA RE-NORMALIZATION; AB-01 AND AB-02 ARE HARD GATES FOR VIABILITY V.

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

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

TABLE III
AGGREGATED EVALUATION RESULTS FOR TYPESCRIPT/TRPC ($n = 30$ PROMPTS). VIABILITY V AND QUALITY Q ARE DEFINED IN SECTION IV-E. "PERFECT QUALITY" DENOTES $Q = 10$ (ALL APPLICABLE CHECKS PASS). "NON-VIABLE" DENOTES $V = 0$ (AB-01 OR AB-02 = FAIL). MEAN QUALITY IS COMPUTED OVER VIABLE APPS ONLY ($V = 1$).

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

Note. Scoring rubric and check definitions in Section IV-E.

Viability (binary):

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

Quality (0–10):

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

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

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

V. RESULTS

A. Environment Scaffolding Impact (TypeScript/tRPC only)

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

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

TABLE IV
CHECK-SPECIFIC OUTCOMES ACROSS $n = 30$ TYPESCRIPT/TRPC TASKS. SEE SECTION IV-E FOR CHECK DEFINITIONS, PASS/WARN/FAIL GRADING, AND THE VIABILITY RULE. NA INDICATES THE CHECK WAS NOT APPLICABLE TO A PROMPT (E.G., AB-04 WHEN NO VIEW/EDIT FLOWS ARE REQUIRED). "PASS RATE (EXCL. NA)" IS COMPUTED OVER APPLICABLE CASES ONLY.

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

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

B. Open vs Closed Model Performance

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

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

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

C. 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.

D. 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.

E. 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.

F. 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.

VI. Discussion

A. 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.

B. 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 [19], 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.

VII. Conclusion

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

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

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

References

[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] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[3] C. Labs, "Swe-bench technical report," https://cognition.ai/blog/swe-bench-technical-report, 2024.

[4] 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

[5] 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

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

[7] 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

[8] 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

[9] 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

[10] S. Zhou, F. F. Xu, H. Zhu, X. Zhou, R. Lo, A. Sridhar, X. Cheng, T. Ou, Y. Bisk, D. Fried, U. Alon, and G. Neubig, "Webarena: A realistic web environment for building autonomous agents," 2024. [Online]. Available: https://arxiv.org/abs/2307.13854

[11] G. Mialon, C. Fourrier, C. Swift, T. Wolf, Y. LeCun, and T. Scialom, "Gaia: a benchmark for general ai assistants," 2023. [Online]. Available: https://arxiv.org/abs/2311.12983

[12] 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

[13] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, and H. Cui, "Agentcoder: Multi-agent-based code generation with iterative testing and optimisation," 2024. [Online]. Available: https://arxiv.org/abs/2312.13010

[14] M. A. Islam, M. E. Ali, and M. R. Parvez, "Mapcoder: Multi-agent code generation for competitive problem solving," 2024. [Online]. Available: https://arxiv.org/abs/2405.11403

[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] 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

[17] 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

[18] 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.

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

Appendix

TABLE V

COMPLETE PROMPT DATASET USED IN EVALUATION ($n = 30$). DATASET CONSTRUCTION DETAILS IN SECTION IV-B. COMPLEXITY LABELS FOLLOW THE RUBRIC IN SECTION V-F: *Low* (STATIC/SINGLE-PAGE UI), *Medium* (SINGLE-ENTITY CRUD), *High* (MULTI-ENTITY/CUSTOM LOGIC).

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