AI-Generated Code Is Not Reproducible (Yet): An Empirical Study of Execution Reliability in LLM-Based Coding Agents

Bhanu Prakash Vangala
bv3hz@missouri.edu
Dept. of Electrical Engg. and Computer Science
University of Missouri
Columbia, MO, USA

Ashish Gehani
ashish.gehani@sri.com
Computer Science Laboratory
SRI
Menlo Park, CA, USA

Tanu Malik
tanu@missouri.edu
Dept. of Electrical Engg. and Computer Science
University of Missouri
Columbia, MO, USA

Abstract

Large Language Model (LLM) coding agents promise to accelerate software development, yet their impact on the reproducibility of generated code remains largely unexplored. We present an empirical study of whether complete software artifacts generated by LLM coding agents can be executed in a clean environment using only the code, dependency specifications, and instructions the agent provides. To conduct the empirical study, we introduce two terms: *execution reliability*, which is the fraction of artifacts that execute successfully out-of-the-box with zero manual intervention, and a two-layer dependency framework comparing the dependencies an agent claims with those actually loaded at runtime. We evaluate three state-of-the-art coding agents across 300 artifacts generated from 100 standardized prompts spanning Python, JavaScript, and Java. Our results show that only 68.3% of artifacts meet this bar, with substantial variation by language (Python 89.2%, JavaScript 61.9%, Java 44.0%) and by agent. Using this dependency framework, we further show a 15× average expansion from declared to actual runtime dependencies, revealing a transparency gap between what agents communicate about their software's requirements and what actually gets loaded at execution time. Finally, analyzing the 31.7% of artifacts that fail via a structured error taxonomy, we find that failures split almost evenly between code errors, where the generated code is itself faulty (49.5%), and environment errors, where the dependency, configuration, or runtime context is incomplete (50.5%), indicating that clean-environment executability, not dependency completeness alone, is the binding constraint on reproducibility. These findings suggest that coding-agent evaluation should treat clean-environment executability as a first-class metric alongside functional correctness.

CCS Concepts

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

Keywords

reproducibility, large language models, code generation, dependency management, software engineering

ACM Reference Format:
Bhanu Prakash Vangala, Ashish Gehani, and Tanu Malik. 2026. AI-Generated Code Is Not Reproducible (Yet): An Empirical Study of Execution Reliability in LLM-Based Coding Agents. In *Proceedings of ACM Conference on Reproducibility and Replicability (ACM REP '26).* ACM, New York, NY, USA, 14 pages.

1 Introduction

Reproducibility forms the cornerstone of computational science. When researchers share code, others must be able to execute it, verify results, and build upon the work [7]. This fundamental principle ensures reliability of results, validates them, and ensures scientific progress. Yet as we increasingly rely on Large Language Models (LLMs) to generate code, a critical question emerges: Is AI-generated code reproducible?

The computational research community already faces a reproducibility crisis. Studies show that a significant portion of published AI/ML research cannot be reproduced due to missing dependencies, incomplete environment specifications, and undocumented requirements [8, 9, 23, 25]. A large-scale study found that 74% of R files from Harvard Dataverse failed initial execution, with missing packages and documentation errors as leading causes [30]. Now, coding agents such as Claude Code, OpenAI Codex, and Gemini Code Assist promise to accelerate development by generating complete artifacts, entire applications with multiple files, configurations, and dependency specifications. But if these coding agents inherit or amplify reproducibility problems, we risk compounding the crisis rather than solving it.

Consider a developer who uses a coding agent to build a sentiment analysis web application in Python. The agent generates an artifact comprising frontend code, backend API files, and a requirements file listing dependencies. While seemingly complete functionally, from a reproducibility perspective, this artifact is reproducible only if the provided files and specifications, when set up in a clean environment, install the declared dependencies, and run the application without further intervention.

We term this property of the agent as *execution reliability*, which requires that the artifacts an agent produces, namely the source code, dependency specification, and usage instructions, to be jointly sufficient to result in an execution without failure. For an agent to satisfy this property, two conditions must both hold. First, the generated code must itself be correct. Second, the dependency specification must be complete and precise, *i.e.,* if it omits requirements the code relies on implicitly, such as an unlisted package, or if it declares packages without pinning the versions needed to reconstruct the environment, execution can fail even when the underlying source code is correct. Execution reliability therefore depends not merely on whether an agent writes working code, but also on whether it accurately and completely declares the environment that code depends on.

To assess this property of coding agents, in this paper we determine *to what extent do current LLM coding agents produce artifacts that execute in a clean environment using only what the agent provides, without manual intervention?* Existing benchmarks for LLM code generation, such as HumanEval [4] and MBPP [3], evaluate code generation failures under the assumption that a working execution environment already exists, and typically test short, single-file, dependency-free snippets. Prior work on environment configuration as a bottleneck in code evaluation and repair benchmarks [13, 32] has observed related challenges in setting up execution environments for existing repositories, but does not isolate agent-declared dependency specification.

To assess coding agents, we must distinguish two possible sources of failure that are often conflated in existing LLM evaluation practice: the agent may generate code that is itself incorrect *i.e.,* a code-generation failure, or it may generate correct code accompanied by an incomplete specification of its execution environment, *i.e.,* a specification failure.

We evaluate three coding agents, Claude Code [2], OpenAI Codex [20], and Gemini Code Assist [6], across 300 generated artifacts spanning Python, JavaScript, and Java. Each artifact is evaluated in an isolated environment provisioned with only operating system packages, using exclusively the code, dependency specification, and instructions the agent provides. To characterize *why* artifacts fail beyond a binary success/failure outcome, we introduce a two-layer dependency framework of *claimed* and *transitive* dependencies. The *claimed* dependencies are the dependencies that the agent declares, and the *runtime* dependencies are the full transitive closure loaded at execution time. From these two layers we derive two metrics that determine the quality of agent's specification: incomplete specification, i.e., environment specification the agent should have declared but omitted, and the hidden expansion from declared to runtime dependencies, also termed as the undisclosed transitive complexity.

Our evaluation finds that 68.3% of the 300 generated artifacts execute successfully without manual intervention, with substantial variation by language (Python 89.2%, JavaScript 61.9%, Java 44.0%). Among the artifacts that fail, incomplete dependency declarations account for only 7.4% of failures; the failures instead split almost evenly between code errors, where the generated code is itself faulty (49.5%), and environment errors, where the dependency, configuration, or runtime context is incomplete (50.5%), indicating that execution reliability is constrained by both code correctness and environment completeness rather than dependency declaration alone. Even among artifacts that execute successfully, we observe a substantial gap between claimed and runtime dependencies: an average of 3 declared packages corresponds to roughly 45 packages loaded at runtime, a 15× expansion. Because package managers resolve transitive dependencies automatically, this expansion does not itself cause execution failures; rather, it reflects a transparency gap between what agents communicate about an artifact's requirements and its actual runtime footprint.

This paper makes four contributions. *First*, we formalize *execution reliability* as a metric for evaluating whether LLM-generated artifacts execute successfully in isolated environments without manual intervention. *Second*, we introduce a two-layer dependency framework that distinguishes incomplete specification from undisclosed transitive complexity, which helps to assess the agents specification. *Third*, we conduct an empirical evaluation of three state-of-the-art coding agents across 300 artifacts in three languages, reporting execution reliability, dependency completeness, and runtime dependency expansion. *Finally*, we provide a failure taxonomy characterizing the distribution of error types underlying execution failures, informing where future agent improvements would have the greatest impact on reproducibility.

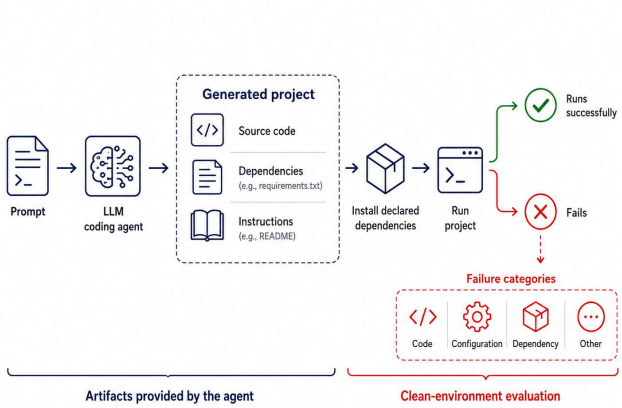

Figure 1: Execution reliability in practice. An LLM coding agent turns a user request into an artifact consisting of source code, a dependency specification, and usage instructions. Reproducibility requires that these agent-provided artifacts alone be sufficient to install dependencies and run the software in a clean environment; when the specification is incomplete or the code is faulty, execution fails.

The rest of the paper is organized as follows: Section 2 describes execution reliability in coding agents and the dependency framework. Section 3 describes the dataset construction and experimental setup. We outline our comprehensive results in Section 4. We discuss the scope of our methodology in Section 5. We describe related work in Section 6 and conclude in Section 7.

2 Agent Evaluation

To evaluate whether agents based on large language models (LLMs) can generate executable software within isolated, minimalist environments, we formalize a property of the agent termed *execution reliability* (Figure 1).

Let $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$ denote a set of standardized prompts, and let $\mathcal{L} = \{L_1, L_2, \ldots, L_M\}$ denote the set of coding agents under evaluation. Each prompt $p_i \in \mathcal{P}$ specifies the requirements for a functionally complete software artifact. For a given agent $L_j \in \mathcal{L}$ responding to prompt p_i, the generated output G is modeled as a tuple:

$$G(L_j, p_i) = \langle C_i^j, D_i^j, I_i^j \rangle \tag{1}$$

where C_i^j is the primary source code artifact that the agent generates, D_i^j is the explicit dependency specification file (*e.g.,* requirements.txt for Python artifacts), and I_i^j contains any usage instructions the agent provides. Given a well-formed prompt, this output triple captures the entirety of the information a user or automated pipeline receives from the generative agent. Here i indexes the prompt and j the agent, so each artifact is identified by the pair (i, j).

Using this representation, the execution reliability $\mathcal{E}(L_j)$ of an agent L_j is defined as the proportion of prompts that yield a successfully executable artifact without external human intervention:

$$\mathcal{E}(L_j) = \frac{\left|\left\{i \in \{1, \ldots, N\} : \text{exec}(C_i^j, D_i^j) = \text{success}\right\}\right|}{|\mathcal{P}|} \tag{2}$$

The evaluation function exec$(\cdot)$ returns *success* if and only if the source code compiles or executes to completion using exclusively the declared dependencies and instructions; otherwise it returns *failure*. The success of the execution is determined by executing the code in a standardized, sandboxed container image provisioned only with default operating system utilities and no network access to fetch unclaimed dependencies. Within this sandbox, any artifact that requires manual debugging, source code modification, or the resolution of unlisted dependencies returns failure.

Within the failure set we additionally distinguish a *partial* functional subset: artifacts that initialize and run using only the declared dependencies but cannot complete their intended function because a required external service (*e.g.,* a database, message broker, or third-party API) is unavailable in the sandbox. A partial functional artifact is still a failure for the purposes of execution reliability, since it cannot be reproduced end-to-end from the agent's output alone; we tag it separately only because it is distinguished by an absent external service rather than by any defect in the code or dependency specification.

2.1 Two-Layer Dependency Framework

We now formalize the dependency requirements of an artifact. We adopt a broad definition of dependency: any resource required for successful execution beyond the generated source code itself. This includes manifest dependencies (packages declared in requirements.txt, package.json, or pom.xml), but also non-manifest requirements such as system libraries, external services (databases, message brokers, third-party APIs), environment variables, and configuration files. Based on this definition, we classify dependencies into two layers: the dependencies an agent *claims* as sufficient for execution, and the dependencies actually loaded at *runtime* for a successful execution. To describe this framework, we define per artifact and agent metrics but suppress the (i, j) indices for readability.

Layer 1: Claimed Dependencies. Let D_c denote the set of packages explicitly declared by agent L_j in the dependency specification D_i^j:

$$D_c = \text{parse}(D_i^j) \tag{3}$$

PROMPT TO LLM Agents:

```
[Task Description]
IMPORTANT: Provide EVERYTHING needed for reproduction in
[Environment]:
1. Complete working [Language] code
2. Complete requirements.txt/package.json/pom.xml
with ALL dependencies and versions
3. Brief usage instructions
Make it 100% reproducible in a clean [environment].
```

Figure 2: Standardized prompt template used for all code generation requests. Placeholders [Task Description], [Environment], and [Language] are instantiated per prompt.

where parse$(\cdot)$ extracts the set of named package identifiers from a structured manifest. These are the dependencies the agent asserts are sufficient for execution.

Missing Dependencies. Beyond what the agent declares, some directly-imported packages required for execution may be absent from D_c. Let D_m denote this set of *missing dependencies*, determined by iterative import resolution:

$$D_m = \left\{d \notin D_c \ : \ \text{exec}(C_i^j,\, D_c) \text{ raises ImportError requiring } d\right\} \tag{4}$$

Installing $D_c \cup D_m$ yields a first working configuration; D_m is empty if and only if the agent's specification is complete with respect to its direct imports.

Layer 2: Runtime Dependencies. Let closure(d) denote the reflexive transitive closure of package d in the dependency graph, i.e., d itself together with every package it directly or indirectly requires as resolved by the package manager. The full runtime dependency set is:

$$D_r = \bigcup_{d \in D_c \cup D_m} \text{closure}(d) \tag{5}$$

Since closure is extensive ($d \in \text{closure}(d)$), we obtain the containment chain:

$$D_c \ \subseteq \ D_r \tag{6}$$

The two layers, together with the missing set D_m, give rise to two metrics that characterize distinct failure modes in agent-generated dependency specifications.

The *completeness gap* quantifies the number of dependencies omitted by the agent in its initial specification:

$$\Delta_{\text{completeness}} = |D_m| \tag{7}$$

The *runtime multiplier* quantifies the ratio of installed packages to declared packages for an artifact:

$$\rho_{\text{runtime}} = \frac{|D_r|}{|D_c|} \tag{8}$$

This metric is defined only when $|D_c| > 0$; artifacts for which $D_c = \emptyset$ are classified as specification failures and excluded from this computation. In our evaluation, no evaluated artifact produced $D_c = \emptyset$; this guard is included for completeness. The per-language multipliers reported in Section 4 are the ratio of the mean runtime count to the mean claimed count over all artifacts in a language.

2.2 Prompt Design

Each prompt is instantiated from the standardized template shown in Figure 2, which requires the agent to produce: (1) complete executable source code, (2) a fully specified dependency manifest with explicit version pins, and (3) usage instructions. The template is held constant across all agents and languages; only the task description, target environment, and language identifier are varied per prompt. Tasks are deliberately selected to require external dependencies, as dependency-free programs trivially satisfy exec(·) and do not exercise dependency specification. The complete prompt catalog is provided in Appendix A.

2.3 Iterative Resolution Protocol

To characterize the failure modes of agent-generated artifacts, we define a structured post-failure resolution protocol. This protocol is applied exclusively to artifacts that fail primary evaluation under exec(·); it plays no role in computing execution reliability $\mathcal{E}(L_j)$. All resolution steps are applied to a copy of the original artifact; C_i^j and D_i^j are never modified for the purposes of reliability reporting.

For each generated triple $G(L_j, p_i) = \langle C_i^j, D_i^j, I_i^j \rangle$, we install $D_c = \text{parse}(D_i^j)$ into a clean sandboxed environment and invoke $\text{exec}(C_i^j, D_c)$. An artifact is marked *reproducible* if and only if this call returns success with zero manual intervention. Artifacts that fail, including the partial subset defined in Section 2, proceed to post-failure analysis.

Algorithm 1 processes each failed artifact to determine its resolved dependency set $D_c \cup D_m$ and to assign it to one of the error categories reported in Section 4. The algorithm attempts an automated fix only for two categories: missing dependencies (install the absent package) and simple code defects (apply a small edit); it iterates for at most $k = 5$ rounds, matching the 2–3 resolution steps typical in practice with margin for slower cases. For configuration errors, absent-service or missing-data conditions, and unprocessable output, it records the category and terminates immediately without attempting a fix, since these are not resolvable by adding a package or a one-line code edit. The algorithm is thus a classifier with limited repair, not a general repair loop, and it plays no role in computing execution reliability.

Auxiliary definitions. In the algorithm, Parseable(C) returns false when the generated output is too malformed to compile or interpret at all (MalformedOutput). The function Classify(e) maps an execution error to a category, such as ImportError ↦ Dependency; SyntaxError, NameError ↦ SyntaxError; ConfigError ↦ Configuration; ServiceUnavailable, BuildRequired, missing input data ↦ SystemError; all other errors (version conflicts, credential failures) ↦ Other. ExtractPackage(m) parses the missing module identifier from an ImportError or ModuleNotFoundError traceback (*e.g.,* the token X in "No module named 'X'").

ApplyMinimalFix(C, e) denotes a human-applied, scope-minimal edit that resolves error e without altering program logic or structure; permissible edits are restricted to correcting import paths, resolving undefined names, and repairing surface-level syntax errors. Any edit requiring semantic restructuring leaves the artifact as Unfixable in the SyntaxError category. Only the Dependency and SyntaxError branches attempt a fix; the Configuration, SystemError, Other, and MalformedOutput branches record the category and terminate without modification, since these failures cannot be resolved by installing a package or applying a one-line edit. The resulting per-category counts are reported in Section 4.

Algorithm 1 Failure Classification with Limited Repair

Input: Code C, Claimed dependencies D_c, bound $k \in \mathbb{Z}^+$
Output: Resolved dependencies $D_c \cup D_m$;
category $\kappa \in$ {Dependency, SyntaxError, Configuration, MalformedOutput, SystemError, Other};
status $S \in$ {Resolved, Unfixable}

if ¬ Parseable(C) **then**
  return D_c, MalformedOutput, Unfixable
end if
$D_{\text{curr}} \leftarrow D_c$; Install($D_{\text{curr}}$)
for $t \leftarrow 1$ **to** k **do**
  $r \leftarrow$ Execute(C)
  if r = success **then**
    return D_{curr}, κ, Resolved
  end if
  $\kappa \leftarrow$ Classify(r.error)
  if κ = Dependency **then**
    $d \leftarrow$ ExtractPackage(r.message)
    $D_{\text{curr}} \leftarrow D_{\text{curr}} \cup \{d\}$
    Install($\{d\}$)
  else if κ = SyntaxError **then**
    ApplyMinimalFix(C, r.error)
  else
    return D_{curr}, κ, Unfixable {recorded, not repaired}
  end if
end for
return D_{curr}, κ, Unfixable

3 Dataset Construction and Experimental Setup

We describe the prompts, the software artifacts, and the two-layer dependency framework constructed from the prompts.

3.1 Dataset Construction

We construct a benchmark dataset $\mathcal{P}$ of 100 prompts, each specifying the requirements for a functionally complete software artifact. To evaluate dependency specification behavior across heterogeneous language ecosystems, the prompts are partitioned by target programming language:

$$\mathcal{P} = \mathcal{P}_{\text{Py}} \uplus \mathcal{P}_{\text{JS}} \uplus \mathcal{P}_{\text{Java}} \tag{9}$$

where ⊎ denotes disjoint union, with $|\mathcal{P}_{\text{Py}}| = 40$, $|\mathcal{P}_{\text{JS}}| = 35$, and $|\mathcal{P}_{\text{Java}}| = 25$. This distribution reflects the relative prevalence of these languages in professional software development as reported in recent industry surveys [12, 28].

Prompts were constructed in two stages. First, candidate task descriptions were generated by querying GPT-4o (OpenAI, version 2024-08-06) with the instruction to enumerate common dependency-requiring development tasks for each target language. Second, the candidate set was manually curated to ensure: (i) coverage across five application domains: data processing, web services, security

Table 1: Agent configurations used in evaluation. All agents were invoked through their command-line interfaces with default tools and permissions. Claude Code and Gemini Code Assist are identified by their underlying model; a CLI/agent version (Codex 0.52.0) over an OpenAI GPT-4-class code model.

Agent	Underlying Model	CLI / Version
Claude Code	claude-opus-4-1	Claude Code
Codex	OpenAI GPT-4 (code)	Codex 0.52.0
Gemini Code Assist	gemini-2.5-pro	Gemini CA

tooling, DevOps automation, and machine learning, and (ii) variation in dependency complexity, ranging from single-library utilities to multi-framework applications. Prompts that could be satisfied without external packages, or that were functionally redundant within a domain, were removed. The complete prompt catalog is provided in Appendix A.

Each of the three agents under evaluation ($|\mathcal{L}| = 3$) responds to all 100 prompts, yielding a corpus of $|\mathcal{L}| \times |\mathcal{P}| = 300$ generated artifacts $G(L_j, p_i)$ as defined in Equation 1.

3.2 Experimental Infrastructure

Each generated artifact is evaluated in an isolated, stateless execution environment. We provisioned AWS EC2 instances (type t2.xlarge; 4 vCPUs, 16 GB RAM) running Ubuntu 22.04 LTS with the following language runtimes: Python 3.10.12, Node.js 18.17.1 with npm 9.6.7, and OpenJDK 17.0.8 with Apache Maven 3.6.3. Each language uses a public base image (python:3.10-slim, node:18-slim, and maven:3.9-eclipse-temurin-17 for Python, JavaScript, and Java, respectively). The baseline environment comprises 71 Ubuntu system packages and 20 pre-installed user packages. After each artifact evaluation, the environment is restored to this exact state by reverting to a fixed snapshot, guaranteeing that no residual packages from one artifact can influence the evaluation of any subsequent artifact.

Table 1 summarizes the configuration of each agent evaluated. All agents were invoked with network access enabled during code generation. Evaluation and execution were performed separately on the standardized environment described above, which had no network access beyond package-index resolution (i.e., PyPI, npm registry, Maven Central).

Results reported in this paper reflect statistically grounded observations for the configurations in Table 1; however, because coding agents are rapidly evolving systems whose behavior may vary with model version, agent release, and tool permissions, generalization beyond the evaluated configurations should be made with caution.

To empirically instantiate D_r for each evaluated artifact, we employ language-specific tooling to intercept the full set of packages resolved during execution. For Python, we use Sciunit [29], which traces system calls via strace to record every module loaded during execution. For JavaScript, we extract the full transitive dependency tree from npm after installation using npm list --all --json, which enumerates all resolved package identifiers in the dependency graph. For Java, we extract the equivalent information from Maven's resolved dependency graph using mvn dependency:tree. In all three cases, the resulting set corresponds to the empirical observation of D_r as defined in Equation 5, under the package versions available at evaluation time.

Table 2: Execution reliability across three LLM coding agents. Partial indicates artifacts that execute but require, unavailable, external services (databases, third-party APIs) for full functionality; these are excluded from the success count.

Agent	Total	Success	Failed (partial)	Rate
Claude	100	73	27 (4)	73.0%
Gemini	100	72	28 (7)	72.0%
Codex	100	60	40 (3)	60.0%
Overall	300	205	95 (14)	68.3%

4 Experimental Results

We present experimental results measuring overall execution reliability across 300 artifacts, then evaluate reliability by languages and agents, and finally report dependency gap analysis.

4.1 Overall Execution Reliability

Table 2 reports execution reliability across all 300 evaluated artifacts. Of the 300 artifacts, 205 (68.3%) execute successfully out-of-the-box. Of the remaining 95 artifacts, 81 were outright failures and 14 were partial executions. In Table 2, the failed count is written as total (partial); for example, Claude's "27 (4)" means 27 failures of which 4 are partial. The partial executions were subjected to post-failure analysis via Algorithm 1 to characterize failure modes; none of the partial executions are counted as successes regardless of fixability under that protocol. This is because they cannot be reproduced end-to-end from the agent's output alone, and reflects an absent runtime service rather than any defect in the generated code or dependency specification.

4.2 Language-Specific Execution Reliability

Table 3 reports execution reliability success rates by programming language with Wilson 95% confidence intervals. Python achieves the highest rate (89.2%), followed by JavaScript (61.9%) and Java (44.0%).

The confidence intervals are non-overlapping across all three languages, indicating that the observed hierarchy is statistically robust within this evaluation. We note that this hierarchy likely reflects a combination of factors including ecosystem dependency complexity and the relative representation of each language in agent training data. However, currently our analysis does not isolate these effects.

4.3 Agent-Language Specialization

Table 4 drills down further and reports the success rates per agent, language pair, with Wilson 95% confidence intervals. Given per-cell sample sizes of $n = 25$–40, point estimates in the Table 4 are to be interpreted alongside the reported intervals rather than as precise rankings.

Table 3: Language-specific execution success rates with Wilson 95% confidence intervals.

Language	Total	Success	Rate	95% CI
Python	120	107	89.2%	[82.4, 93.6]
JavaScript	105	65	61.9%	[52.3, 70.6]
Java	75	33	44.0%	[33.2, 55.3]

Several patterns emerge from Table 4. All three agents perform comparably on Python, with overlapping confidence intervals at the upper end of the range (80.0–100.0%). This suggests that Python is the most manageable target. We believe that this likely reflects not only Python's flat dependency model but also an indication of prevalence of Python in the agents' training data; however, we still cannot cleanly separate the two effects here.

Java performance diverges substantially: Claude achieves 80.0% [60.9, 91.1] while Gemini and Codex reach only 28.0% [14.3, 47.6] and 24.0% [11.5, 43.4] respectively. This 3× performance gap for the same language is rather surprising. The Claude–Java interval and the Gemini–Java and Codex–Java intervals are non-overlapping, indicating that Claude's advantage on Java is robust to sampling uncertainty at these cell sizes.

Figure 3 visualizes the full success rate matrix between coding agents and languages. Gemini's row tells a story of extreme specialization, achieving perfect Python reproduction (1.000) but failing dramatically with Java (0.280). Meanwhile, Claude's row shows the most balanced performance across all three languages (0.800, 0.600, 0.800), making it the only agent that handles enterprise Java effectively. Codex shows consistent Python strength (35/40) but poor Java performance (6/25). These specializations were not advertised by vendors but emerge clearly through systematic evaluation.

4.4 Error Classification

We classify errors according to the iterative resolution protocol (Algorithm 1) into two top-level groups: *Code Errors*, where the fault lies in the code the agent generated, and *Environment errors*, where the code is well-formed but the surrounding dependency, configuration, or runtime context is incomplete. Failures split almost evenly between the two top-level groups: Code Errors account for 49.5% (47 artifacts) and Environment errors for 50.5% (48 artifacts). Table 5 reports the full sub-category breakdown, and Figure 4 shows the composition by agent.

Within Code Errors, Syntax Errors (32.6%, 31 artifacts) are the larger sub-category, comprising defects such as import paths broken by compressing multi-file artifacts into a single file, incorrect file location assumptions, uninitialized variables, mishandled JavaScript async/await patterns, and malformed Maven XML. These require understanding the code's intent to correct. Malformed Output (16.8%, 16 artifacts), code too malformed to parse or execute at all, occurred exclusively in Codex and Gemini outputs (8 artifacts each); Claude produced none.

Within Environment errors, Configuration (18.9%, 18 artifacts) is the largest sub-category, covering correct code paired with incomplete or malformed configuration. This includes missing build

Table 4: Execution success rates by agent and language with Wilson 95% confidence intervals.

Agent	Language	n	Success	Rate	95% CI
Claude	Python	40	32	80.0%	[65.2, 89.5]
Claude	JavaScript	35	21	60.0%	[43.6, 74.4]
Claude	Java	25	20	80.0%	[60.9, 91.1]
Gemini	Python	40	40	100.0%	[91.2, 100.0]
Gemini	JavaScript	35	25	71.4%	[54.9, 83.7]
Gemini	Java	25	7	28.0%	[14.3, 47.6]
Codex	Python	40	35	87.5%	[73.9, 94.5]
Codex	JavaScript	35	19	54.3%	[38.2, 69.5]
Codex	Java	25	6	24.0%	[11.5, 43.4]

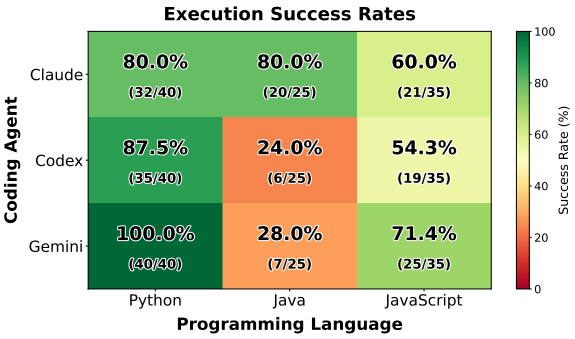

Figure 3: Success rate heatmap reveals agent specializations. Claude excels at Java (80%), Gemini achieves perfect Python (100%), while all agents struggle with Java except Claude.

scripts, incorrect entry points, or partial artifact setups. System Error (10.5%, 10 artifacts) covers runtime-context failures independent of configuration, such as missing sample input the code expects but the agent did not supply (e.g., a referenced PDF, audio file, or image), missing system libraries outside the package manager's scope (e.g., libgl1), and build- or service-level requirements unavailable in the clean sandbox. Dependency (7.4%, 7 artifacts) covers missing package specifications: ModuleNotFoundError in Python, Cannot find module in JavaScript, ClassNotFoundException in Java, where the agent imported a package in code but omitted it from the dependency file. Finally, Other (13.7%, 13 artifacts) collects remaining heterogeneous cases, including missing credentials and version conflicts (e.g., Package A requiring B v2.0 while Package C requires B v3.0).

Figure 4 shows agent-specific patterns within this taxonomy. Codex has the most Code Errors overall (21), driven by both the most Syntax Errors (13) and a joint-highest share of Malformed Output(8). The Malformed Output split is the starkest agent difference: 8 each for Codex and Gemini, none for Claude, indicating Claude reliably produced parseable output even when an artifact failed for other reasons. Configuration errors similarly concentrate in Codex (11, versus 5 for Claude and 2 for Gemini). Dependency errors are uniformly low across all three agents (Claude 2, Codex 3, Gemini 2), consistent with dependency completeness being a minor

Table 5: Aggregated error distribution across all 95 non-successful artifacts, grouped into Code Errors (faults in the generated code) and Environment errors (incomplete dependency, configuration, or runtime context).

Group	Sub-category	Count (%)
Code Errors	Syntax Errors	31 (32.6%)
Code Errors	Malformed Output	16 (16.8%)
Code Errors	**Subtotal**	**47 (49.5%)**
Environment	Configuration	18 (18.9%)
Environment	Other	13 (13.7%)
Environment	System Error	10 (10.5%)
Environment	Dependency	7 (7.4%)
Environment	**Subtotal**	**48 (50.5%)**
Total		95 (100%)

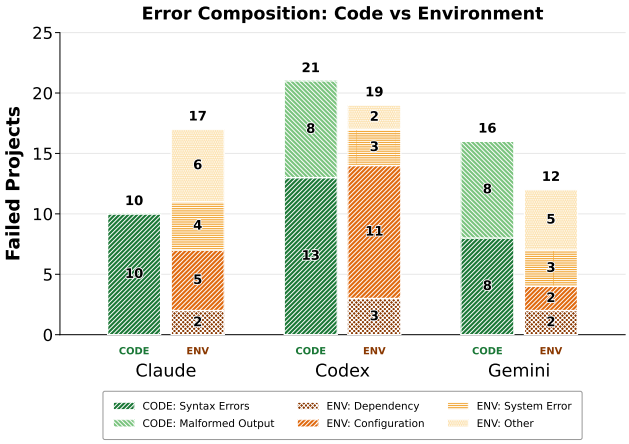

Figure 4: Error composition by agent. Each agent has two stacked bars: Code Errors (Syntax Errors + Malformed Output) and Environment (Dependency + Configuration + System Error + Other).

contributor to non-executability relative to code and configuration correctness. Claude's failures lean toward the Environment group (17 of its 27), while Codex splits nearly evenly between Code (21) and Environment (19).

4.5 Analysis of LLM Self-Correction

A natural question is whether LLM agents can fix their own reproducibility failures when presented with the resulting error output. To investigate this, we conducted a self-correction experiment on the 95 non-successful artifacts. For each failure, we fed the complete error output (stack traces, error messages, and the relevant code) back to the same LLM agent that generated the artifact, asking it to diagnose and fix the issue. We repeated this feedback loop for up to 5 iterations per artifact, re-executing after each correction attempt and feeding any new errors back to the agent.

The results reveal that self-correction has limited effectiveness for reproducibility failures. Dependency errors (the 7 artifacts that failed due to missing package declarations) were the most amenable to self-correction: agents could typically identify and add the missing package within 1–2 iterations when presented with a clear ModuleNotFoundError or ClassNotFoundException. However, Syntax Errors (the 31 artifacts with structural or logic defects) proved far more resistant. Agents frequently introduced new errors while attempting to fix existing ones, or applied superficial patches that shifted the failure to a different location without resolving the underlying structural problem. Configuration errors (18 artifacts) showed mixed results, with some fixable through iterative feedback but many requiring understanding of the intended artifact structure. The 16 Malformed Output artifacts, which were too malformed to parse, remained unfixable even after 5 iterations in all cases.

These findings suggest that while LLM self-correction [31] is a promising direction, current agents lack the ability to reliably diagnose and repair their own multi-file artifact outputs. The feedback loop helps with simple, well-localized errors (such as a missing import), but struggles with the structural and deep dependency configuration errors that constitute the majority of reproducibility failures.

4.6 Dependency Gap Analysis

We report the completeness gap across all 300 artifacts to measure how often agents fail to declare a package their own code imports:

$$P(\Delta_{\text{completeness}} = k) = \begin{cases} 0.94 & \text{if } k = 0 \\ 0.06 & \text{if } k = 1 \end{cases} \tag{10}$$

Of the 300 artifacts, 282 (94%) declare all their direct imports ($k = 0$), while 18 (6%) leave out exactly one imported package ($k = 1$); no artifact left undeclared more than one (Figure 5). These gaps surface only through execution: a missing package is invisible in the declaration and appears solely as an import error when the code runs. Not every undeclared package is equally consequential: some block execution outright, while others are optional or supplied transitively by another declared package, so the artifact still runs. The completeness gap therefore captures a declaration defect that may or may not manifest as a runtime failure.

The undeclared packages reveal ecosystem-specific oversights. In Python, they included lxml (parsing), python-dotenv (configuration), and bcrypt (authentication); in JavaScript, body-parser for Express applications, ws for WebSocket implementations, and dotenv for environment-variable handling; and in Java, test framework specifications (particularly JUnit) and logging implementations (such as SLF4J). These are aggregate examples across all artifacts in each language; no single artifact left more than one package undeclared, consistent with the maximum completeness gap of one shown in Figure 5.

The runtime multiplier reveals a hidden property of agent-generated dependencies: the number of packages that actually load at runtime is far larger than the number the agent declares. Figure 6 visualizes this expansion across the three languages.

The multiplier is an aggregate for the whole language, not the value of any single artifact and not the largest observed case. For each language we average the number of packages an agent declares across all its artifacts ($\overline{|D_c|}$) and the number actually loaded at runtime ($\overline{|D_r|}$); the multiplier $\rho = \overline{|D_r|}/\overline{|D_c|}$ is the ratio of these

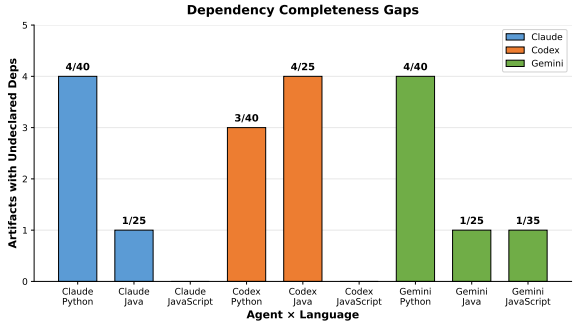

Figure 5: Dependency completeness gaps by agent and language. Each bar shows the number of artifacts that left at least one direct import undeclared, out of the total for that agent–language pair.

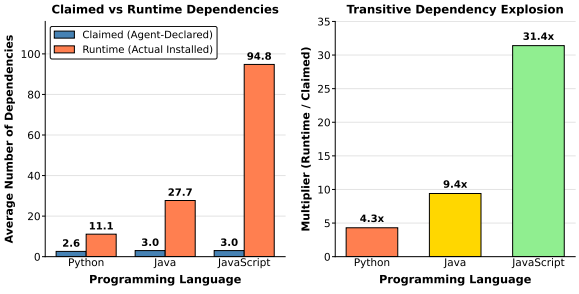

Figure 6: Runtime dependency explosion: the gap between claimed (agent-declared) and runtime (actually installed) dependencies, averaged per language. JavaScript expands the most (31.4×), followed by Java (9.4×) and Python (4.3×).

two averages, describing the typical declaration-to-runtime gap. Averaged this way, JavaScript expands the most: agents declare about 3 packages but 95 actually load ($\overline{|D_c|} = 3.0 \rightarrow \overline{|D_r|} = 94.8$), a 31.4× multiplier, reflecting `npm`'s deep transitive dependency trees where each library pulls in numerous others. Java expands 9.4× (3.0 → 27.7) under `Maven`'s transitive resolution, and Python 4.3× (2.6 → 11.1), the most contained but still hiding substantial complexity. Across all three languages the artifact-weighted average expansion is 15×.

Much of this runtime footprint is packages the code never directly uses. Package managers resolve dependencies conservatively: when a declared package lists its own dependencies, the manager installs all of them, and their dependencies in turn, whether or not the artifact's code imports them. A single declared library may pull in packages for optional features, alternative backends, or platform-specific code paths that a given artifact never exercises. The runtime count $\overline{|D_r|}$ therefore reflects the full installed footprint, not the smaller set the code genuinely requires.

This over-installation is normal in the `pip`, `npm`, and `Maven` ecosystems and does not by itself constitute a reproducibility failure: package managers resolve these dependencies deterministically, provided the resolved versions are pinned, locked, or containerized. The reproducibility-relevant question is therefore not the size of the closure but whether the agent captures it. None of the evaluated agents emitted lock files (`requirements.lock`, `package-lock.json`, or a Maven dependency lock) or pinned transitive versions; they declared only top-level packages, often without exact versions. We therefore treat the multiplier as a transparency gap between what the agent communicates and what the artifact actually loads, rather than as a direct execution failure. Developers inherit complex dependency chains with no visibility into the true scope of their software supply chain.

5 Discussion

In this paper we have adopted an empirical approach toward measuring execution reliability. We have shown that agents fail on both code errors (49.5%) and environment errors (50.5%), with environment errors slightly exceeding code errors. We treat code errors and environment errors as reproducibility failures because execution reliability asks whether a recipient can run the artifact *as delivered*; under this definition an artifact that was never correct is, *a fortiori*, not reproducible. Efforts to improve execution reliability should therefore equally prioritize code correctness and a complete execution environment; a coding agent that declares dependencies perfectly but generates code with structural defects will fail the majority of the time under our metric, and vice versa.

Execution reliability varies substantially by language and by agent within language, most notably in Java, where Claude's success rate (80.0%) is markedly higher than Gemini's (28.0%) or Codex's (24.0%), with non-overlapping confidence intervals as reported in Table 4. Plausible contributing factors include differences in ecosystem dependency structure (e.g., Maven's nested XML configuration and multiple dependency scopes versus a flat requirements.txt), and differences in the prevalence of each language, and of enterprise-style Java patterns specifically, in each agent's training data. We report this variation as an empirical finding relevant to agent selection rather than as evidence for either explanation.

Our self-correction experiment (Section 4.5) tests whether agents can resolve their own failures when given error feedback. Results there indicate that self-correction is more effective for dependency errors, which are typically resolved by identifying and adding a single missing package, than for code errors or configuration errors, which more often require understanding the intended structure of the artifact rather than reacting to a single error message. This suggests that automated feedback loops are a partial remedy at best: they may close part of the 7.4% dependency-attributable gap, but are unlikely by themselves to close the larger code-error and environment gaps that account for most failures.

Several factors limit the scope of these conclusions. Our metric scores out-of-the-box execution without weighting per-task difficulty, so an agent that succeeds on simple tasks and fails on hard ones is scored identically to the reverse; we mitigate this by releasing all 100 prompts so that difficulty is inspectable. Failure categorization and the minimal-fix step were performed by a single developer, introducing labeling subjectivity. We again mitigate this subjectivity, along with the difficulty-weighting limitation, by releasing all prompts, generated artifacts, execution logs, and failure labels for independent re-labeling and verification. Finally, we use a

single zero-shot prompt template, chosen to measure the default behavior a typical developer would encounter rather than the ceiling reachable with prompt engineering; few-shot exemplars, chain-of-thought, or explicit dependency-pinning instructions might narrow the gaps we observe, and we view characterizing that headroom as future work.

From a practical perspective, our results have implications for software engineering teams. For teams adopting LLM coding agents, our results suggest that execution reliability should be checked empirically against the target language and technology stack rather than assumed uniform across agents. For teams working with generated Python artifacts, our results suggest a comparatively high baseline execution reliability, though not a guarantee. Because top-level dependency declarations do not capture the full runtime closure, we recommend that generated artifacts be paired with a locked or pinned dependency file (*e.g.*, pip freeze, package-lock.json, or a Maven dependency lock) before being treated as reproducible artifacts, regardless of agent or language.

6 Related Work

Environment Reproducibility. Computational reproducibility, defined as the ability to regenerate consistent empirical results using the exact code, data, and digital artifacts published by the original authors is essential to the integrity of modern scientific and algorithmic workflows [18]. However, achieving this standard is heavily throttled by failures in environment reproducibility, where software execution states decay or drift over time [16]. For instance, in a large-scale study of over 9,000 R scripts archived in the Harvard Dataverse, Trisovic et al. [30] discovered that 74% of the files failed to execute on the initial attempt, primarily due to missing or incorrectly specified dependencies. A mirroring analyses of over 860,000 unique Jupyter Notebooks mined from GitHub revealed that a mere 24.1% could be successfully re-executed to completion due to unrecorded libraries and environmental volatility [22]. A community survey [24] highlighted environmental reproducibility as a key issue in large-scale artifact evaluation. To mitigate these environmental discrepancies, research has yielded application virtualization and lightweight provenance capture tools [1, 29], which encapsulate active execution environments along with their underlying binaries, data dependencies, and system configurations to ensure transportable runtime contexts.

However, these traditional frameworks assume a reference execution that has been established by humans. LLM coding agents can synthesize source code, build configurations, and dependency declarations using prompts. In this paper we examine generative composition errors and systemic misalignment between the generated source code, its accompanying configuration files, and the target runtime environment.

Benchmarking Constraints in Code Generation. Evaluating LLM coding capabilities remains a highly active domain, yet existing evaluation protocols largely decouple functional correctness from environment reproducibility. Foundations benchmarks such as HumanEval [4] and MBPP [3] isolate individual code snippets, executing them within pre-configured, monolithic runtime environments that abstract away dependency management entirely.

More recent, agent-centric evaluation suites attempt to evaluate real-world software engineering complexities but introduce distinct environmental assumptions. SWE-bench [13] requires models to resolve issues within established, fully provisioned repositories where the baseline dependency tree is already fixed. Similarly, frameworks like DevBench [14] and LiveCodeBench [10] assess multi-file generation and repository-level engineering, yet their evaluation pipelines automatically supply the target execution container or default runtime matrices. These benchmarks assess whether a code patch or script satisfies static functional unit tests, rather than testing whether the artifact generated by the LLM contains the necessary explicit configuration to independently reproduce that environment.

Dependency Complexity and Container Decay. Modern software engineering is characterized by an escalating reliance on deeply nested, transitive third-party dependencies. Studies of package-manager ecosystems have documented how dependency networks grow and evolve over time [5], and Zimmermann et al. [33] show that npm packages reach large numbers of transitive dependencies, creating a "small world" network topography where minor upstream changes or omission cascades can collapse downstream systems. This problem is compounded by dependency bloat [26] and data bloat [17, 19], wherein unnecessary dependencies and data gets downloaded along with a software.

Containers do not permanently halt environmental decay. Empirical studies tracking containerized builds over multi-year horizons demonstrate that a significant percentage of Dockerfiles become non-reproducible over time due to unpinned base images, broken package repository mirrors, and evolving transitive requirements [15]. Because autoregressive models lack inherent systemic awareness of these shifting runtime states, they predictably fail to generate the strict semantic versioning constraints required to guarantee execution.

Quantitative Evaluation Gaps in Generative Software. While substantial research examines LLM code defects [11] and security vulnerabilities [21], the formal verification of generated deployment configurations remains an overlooked paradigm. Existing literature heavily documents semantic flaws or specific linguistic anomalies, such as package hallucination, wherein LLMs invoke non-existent library names or obsolete APIs [27].

However, prior studies typically observe these flaws through post-hoc manual debugging or within pre-provisioned sandboxes that mask environmental fragility. To our knowledge, no existing work systematically measures whether multi-file artifacts generated by autonomous coding agents can successfully assemble and initialize in a pristine container using solely their own self-contained metadata. This paper evaluates the end-to-end deployability of agent-generated systems across multiple language ecosystems (Python, JavaScript, and Java). By mapping the delta between an agent's explicit declarations and its actual runtime requirements through a two-layer dependency framework, we shift evaluation from a narrow analysis of isolated algorithmic correctness ("does the function work?") to a systemic validation of clean-environment executability ("can the artifact initialize itself out-of-the-box?").

7 Conclusion

Across 300 artifacts generated by three coding agents, 68.3% execute successfully out-of-the-box, with substantial variation by language (Python 89.2%, Java 44.0%). Missing dependency declarations account for only 7.4% of failures; instead, failures split almost evenly between code errors, where the generated code is itself faulty (49.5%), and environment errors, where the dependency, configuration, or runtime context is incomplete (50.5%), indicating that execution reliability is constrained by both code correctness and environment completeness rather than dependency declaration alone. Our two-layer framework further shows a 15× average expansion from claimed to runtime dependencies among successful artifacts, not itself a failure, but a transparency gap, since none of the evaluated agents emit pinned or locked dependency specifications. These findings suggest that evaluating LLM coding agents requires execution reliability as a metric distinct from functional correctness, and that agent selection should be informed by language-specific and configuration-specific evidence rather than assumed uniform performance. We release our prompts, artifacts, and labels to support further study.

Acknowledgments

This work is supported by NASA under grant NASA-AIST-21-0095 and National Science Foundation under grants CNS-1846418.

References

[1] Raza Ahmad, Naga Nithin Manne, and Tanu Malik. 2022. Reproducible Notebook Containers using Application Virtualization. In *2022 IEEE 18th International Conference on e-Science (e-Science)*. 1–10. doi:10.1109/eScience55777.2022.00015

[2] Anthropic. 2025. Claude Code. https://www.anthropic.com/claude-code. Accessed: July 2026.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021).

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[5] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems. *Empirical Software Engineering* 24, 1 (2019), 381–416. doi:10.1007/s10664-017-9589-y

[6] Google. 2025. Gemini Code Assist. https://codeassist.google/. Accessed: July 2026.

[7] Odd Erik Gundersen, Yolanda Gil, and David W. Aha. 2018. On Reproducible AI: Towards Reproducible Research, Open Science, and Digital Scholarship in AI Publications. *AI Magazine* 39, 3 (2018), 56–68. doi:10.1609/aimag.v39i3.2816

[8] Benjamin Haibe-Kains, George Alexandru Adam, Ahmed Hosny, Farnoosh Khodakarami, Levi Waldron, Bo Wang, Chris McIntosh, Anna Goldenberg, Anshul Kundaje, Casey S. Greene, et al. 2020. Transparency and Reproducibility in Artificial Intelligence. *Nature* 586, 7829 (2020), E14–E16. doi:10.1038/s41586-020-2766-y

[9] Matthew Hutson. 2018. Artificial Intelligence Faces Reproducibility Crisis. *Science* 359, 6377 (2018), 725–726. doi:10.1126/science.359.6377.725

[10] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *arXiv preprint arXiv:2403.07974* (2024).

[11] Kevin Jesse, Toufique Ahmed, Premkumar T. Devanbu, and Emily Morgan. 2023. Large Language Models and Simple, Stupid Bugs. In *Proceedings of the 20th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. 563–575. doi:10.1109/MSR59073.2023.00082

[12] JetBrains. 2024. The State of Developer Ecosystem 2024. https://www.jetbrains.com/lp/devecosystem-2024/.

[13] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=VTF8yNQM66

[14] Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, et al. 2024. DevBench: A Comprehensive Benchmark for Software Development. *arXiv preprint arXiv:2403.08604* (2024).

[15] Julien Malka, Stefano Zacchiroli, and Théo Zimmermann. 2026. Docker Does Not Guarantee Reproducibility. *arXiv preprint arXiv:2601.12811* (2026).

[16] Haiyan Meng, Rupa Kommineni, Quan Pham, Robert Gardner, Tanu Malik, and Douglas Thain. 2015. An Invariant Framework for Conducting Reproducible Computational Science. *Journal of Computational Science* 9 (2015), 137–142. doi:10.1016/j.jocs.2015.04.012

[17] Aniket Modi, Rohan Tikmany, Tanu Malik, Raghavan Komondoor, Ashish Gehani, and Deepak D'Souza. 2024. Kondo: Efficient Provenance-Driven Data Debloating. In *Proceedings of the 40th IEEE International Conference on Data Engineering (ICDE)*. 4965–4978. doi:10.1109/ICDE60146.2024.00377

[18] National Academies of Sciences, Engineering, and Medicine. 2019. *Reproducibility and Replicability in Science*. The National Academies Press, Washington, DC. doi:10.17226/25303

[19] Chaitra Niddodi, Ashish Gehani, Tanu Malik, Sibin Mohan, and Michael Lee Rilee. 2023. IOSPReD: I/O Specialized Packaging of Reduced Datasets and Data-Intensive Applications for Efficient Reproducibility. *IEEE Access* 11 (2023), 1718–1731. doi:10.1109/ACCESS.2022.3233787

[20] OpenAI. 2025. Codex. https://openai.com/codex/. Accessed: July 2026.

[21] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P)*. 754–768. doi:10.1109/SP46214.2022.9833571

[22] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *Proceedings of the 16th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. 507–517. doi:10.1109/MSR.2019.00077

[23] Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché Buc, Emily Fox, and Hugo Larochelle. 2021. Improving Reproducibility in Machine Learning Research (A Report from the NeurIPS 2019 Reproducibility Program). *Journal of Machine Learning Research* 22, 164 (2021), 1–20.

[24] Beth A. Plale, Tanu Malik, and Line C. Pouchard. 2021. Reproducibility Practice in High-Performance Computing: Community Survey Results. *Computing in Science & Engineering* 23, 5 (2021), 55–60. doi:10.1109/MCSE.2021.3096678

[25] Edward Raff. 2019. A Step Toward Quantifying Independently Reproducible Machine Learning Research. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 32.

[26] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem. *Empirical Software Engineering* 26, 3 (2021), 45. doi:10.1007/s10664-020-09914-8

[27] Joseph Spracklen, Raveen Wijewickrama, A H M Nazmus Sakib, Anindya Maiti, Bimal Viswanath, and Murtuza Jadliwala. 2025. We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs. In *34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association, Seattle, WA, 3687–3706. https://www.usenix.org/conference/usenixsecurity25/presentation/spracklen

[28] Stack Overflow. 2025. Stack Overflow Developer Survey 2025. https://survey.stackoverflow.co/2025/.

[29] Dai Hai Ton That, Gabriel Fils, Zhihao Yuan, and Tanu Malik. 2017. Sciunits: Reusable Research Objects. In *Proceedings of the 13th IEEE International Conference on e-Science (e-Science)*. 374–383. doi:10.1109/eScience.2017.51

[30] Ana Trisovic, Matthew K. Lau, Thomas Pasquier, and Mercè Crosas. 2022. A Large-Scale Study on Research Code Quality and Execution. *Scientific Data* 9, 1 (2022), 60. doi:10.1038/s41597-022-01143-6

[31] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. 1482–1494. doi:10.1109/ICSE48619.2023.00129

[32] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 37.

[33] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 995–1010.

A Prompt Catalog

The 100 prompts used in this study were designed to cover the breadth of tasks developers commonly delegate to LLM coding

agents. We selected tasks that require external dependencies beyond the standard library (since dependency-free scripts would trivially succeed), represent real-world development patterns observed in open-source repositories and industry workflows, and span multiple complexity levels from single-purpose utilities to multi-component applications. The prompts were initially generated by asking an LLM to suggest common developer tasks for each language, then manually curated to ensure diversity across application domains and dependency complexity. Each prompt was given to all three agents in the same language, producing 300 total artifacts. Tables 6, 7, and 8 list all prompts with their domain categories.

Table 6: Python prompts (p_1 – p_40). Python was allocated 40 prompts reflecting its position as the most popular language for data science and general-purpose development [12, 28]. Tasks exercise common Python ecosystems: scientific computing (numpy, scipy, scikit-learn), web frameworks (Flask, FastAPI), data processing (pandas, lxml), and system utilities.

ID	Task Description	Domain	Key Dependencies Expected
p_1	Web scraper for product information	Web Scraping	requests, beautifulsoup4, lxml
p_2	Sentiment analysis for customer reviews	ML/NLP	nltk, textblob, pandas
p_3	K-means clustering with visualization	ML	scikit-learn, matplotlib, numpy
p_4	REST API with JWT authentication	Web	flask, pyjwt, bcrypt
p_5	Image classifier (pre-trained model)	ML	torch/tensorflow, pillow
p_6	ETL pipeline (CSV to SQLite)	Data	pandas, sqlalchemy
p_7	Async web crawler	Web Scraping	aiohttp, beautifulsoup4
p_8	Time series forecasting (ARIMA)	ML	statsmodels, pandas, matplotlib
p_9	PDF text extraction and analysis	Data	PyPDF2/pdfplumber
p_10	Real-time dashboard (WebSocket)	Web	flask-socketio, websockets
p_11	Email automation with attachments	Utility	smtplib (stdlib), jinja2
p_12	Server log analyzer	Data	pandas, regex
p_13	Twitter bot for hashtag monitoring	API	tweepy, python-dotenv
p_14	File encryption/decryption (AES)	Security	cryptography, pycryptodome
p_15	QR code generator and reader	Utility	qrcode, pillow, pyzbar
p_16	Recommendation system (collaborative filtering)	ML	scikit-learn, scipy, pandas
p_17	CSV data validation against schemas	Data	pandas, cerberus/jsonschema
p_18	Stock price tracker with alerts	API	yfinance, requests
p_19	Markdown to HTML converter	Utility	markdown, pygments
p_20	File deduplication tool	Utility	hashlib (stdlib)
p_21	URL shortener service	Web	flask, sqlite3
p_22	Password strength checker and generator	Security	zxcvbn, secrets (stdlib)
p_23	Audio transcription (speech recognition)	ML	speechrecognition, pyaudio
p_24	Network port scanner	DevOps	socket (stdlib), scapy
p_25	Weather data aggregator (multiple APIs)	API	requests, python-dotenv
p_26	JSON to CSV converter (nested flattening)	Data	pandas, json (stdlib)
p_27	Face detection with OpenCV	ML	opencv-python, numpy
p_28	Redis-based API caching layer	Web	redis, flask
p_29	Command-line task manager (SQLite)	Utility	sqlite3 (stdlib), click
p_30	Credit card validator (Luhn algorithm)	Utility	(minimal deps)
p_31	File compression (multiple formats)	Utility	zipfile (stdlib), gzip
p_32	GitHub repository analyzer	API	requests, PyGithub
p_33	Spell checker with auto-correction	NLP	pyspellchecker, textblob
p_34	Database migration tool	DevOps	alembic, sqlalchemy
p_35	Text summarization (NLP)	ML/NLP	transformers, torch, nltk
p_36	Image resize and optimization	Utility	pillow, numpy
p_37	Cryptocurrency price tracker	API	requests, websockets
p_38	Plagiarism detector	NLP	sklearn, nltk, difflib
p_39	System resource monitor with alerts	DevOps	psutil, smtplib
p_40	SQL query builder with parameterization	Data	sqlalchemy

Table 7: JavaScript prompts (p_41 – p_75). JavaScript was allocated 35 prompts targeting server-side web development with Node.js, the dominant runtime for JavaScript backend development [28]. Tasks exercise the npm ecosystem, where packages exhibit complex interdependency patterns and reach large numbers of transitive dependencies [33], including web frameworks (Express, Fastify), real-time communication (Socket.IO, WebSocket), cloud services (AWS SDK), and middleware patterns common in production backends.

ID	Task Description	Domain	Key Dependencies Expected
p_41	Express REST API with MongoDB	Web	express, mongoose, mongodb
p_42	React component library with state	Frontend	react, redux/zustand
p_43	WebSocket chat server with rooms	Real-time	ws, socket.io
p_44	File upload service (AWS S3)	Cloud	aws-sdk, multer, express
p_45	GraphQL API with authentication	Web	@apollo/server, graphql, jwt
p_46	Real-time data visualization dashboard	Real-time	socket.io, chart.js, express
p_47	Email automation service with templates	Utility	nodemailer, handlebars
p_48	PDF generation from HTML	Utility	puppeteer, html-pdf
p_49	OAuth2 authentication server	Security	passport, express-session
p_50	Serverless image processing	Cloud	sharp, aws-lambda
p_51	URL shortener API with analytics	Web	express, mongoose, nanoid
p_52	Rate limiter middleware	Web	express-rate-limit, redis
p_53	Web scraper (headless browser)	Scraping	puppeteer, cheerio
p_54	File watcher with action triggers	DevOps	chokidar, fs-extra
p_55	Command-line task runner	Utility	commander, chalk, shelljs
p_56	Markdown blog engine	Web	marked, express, highlight.js
p_57	Webhook receiver and processor	Web	express, crypto (stdlib)
p_58	JSON API mock server	Testing	json-server, faker
p_59	Log aggregator service	DevOps	winston, express, morgan
p_60	Real-time notification system	Real-time	socket.io, express, redis
p_61	Job queue processor (Bull)	DevOps	bull, redis, express
p_62	Currency converter API	API	axios, express
p_63	Password hashing service (bcrypt)	Security	bcryptjs, express
p_64	CSV to JSON converter API	Data	csv-parser, express
p_65	Twitter bot (scheduled tweets)	API	twitter-api-v2, node-cron
p_66	Proxy server with request logging	DevOps	http-proxy, morgan, express
p_67	Session management (Redis)	Web	express-session, connect-redis
p_68	Content delivery optimizer	Web	express, compression, helmet
p_69	Health check endpoint for microservices	DevOps	express, os (stdlib)
p_70	Geolocation service (IP-based)	API	geoip-lite, express
p_71	Data validation middleware (Joi)	Web	joi, express
p_72	Distributed cache (Redis cluster)	DevOps	ioredis, express
p_73	Markdown preview server (live reload)	Web	marked, express, livereload
p_74	JWT token manager with refresh tokens	Security	jsonwebtoken, express
p_75	API gateway with routing/load balancing	DevOps	http-proxy-middleware, express

Table 8: Java prompts (p_76 – p_100). Java was allocated 25 prompts focused on enterprise development patterns. Tasks exercise the Maven build system and the Spring ecosystem, which dominate Java enterprise development [12]. These prompts deliberately target dependency-heavy patterns (Spring Boot, Kafka, Elasticsearch) where 75.1% of Maven dependencies have been shown to be bloated [26], stress-testing LLM capabilities with complex transitive dependency graphs.

ID	Task Description	Domain	Key Dependencies Expected
p_76	Spring Boot REST API with JPA and MySQL	Web	spring-boot, spring-data-jpa, mysql
p_77	Microservice with Kafka integration	Messaging	spring-kafka, spring-boot
p_78	JWT authentication (Spring Security)	Security	spring-security, jjwt
p_79	Batch processing (Spring Batch)	Data	spring-batch, spring-boot
p_80	WebSocket chat (STOMP protocol)	Real-time	spring-websocket, stomp
p_81	File processing service (AWS S3)	Cloud	aws-sdk-java, spring-boot
p_82	Scheduled task runner (Quartz)	DevOps	quartz-scheduler, spring-boot
p_83	Cache management (Redis)	DevOps	spring-data-redis, jedis
p_84	Email service (Spring Mail)	Utility	spring-boot-starter-mail
p_85	REST API client (RestTemplate)	Web	spring-web, jackson
p_86	File upload with validation	Web	spring-web, commons-fileupload
p_87	PDF generation from templates	Utility	itext/openpdf, thymeleaf
p_88	Logging framework (Log4j2)	DevOps	log4j2, slf4j
p_89	Data validation (Hibernate Validator)	Web	hibernate-validator, spring-boot
p_90	URL shortener (Spring Boot + MongoDB)	Web	spring-data-mongodb, spring-boot
p_91	Search service (Elasticsearch)	Data	spring-data-elasticsearch
p_92	Notification system (RabbitMQ)	Messaging	spring-amqp, rabbitmq
p_93	Rate limiter (Bucket4j)	Web	bucket4j, spring-boot
p_94	JSON parser and validator	Utility	jackson, gson
p_95	Database connection pool manager	Data	HikariCP, spring-boot
p_96	CSV to database importer	Data	opencsv, spring-data-jpa
p_97	Health check endpoint (Spring Boot)	DevOps	spring-boot-actuator
p_98	Distributed lock manager (Redisson)	DevOps	redisson, spring-boot
p_99	API versioning for REST endpoints	Web	spring-web, spring-boot
p_100	Password encryption (BCrypt)	Security	spring-security, bcrypt