

Practical Lessons from Building ML-Powered Code Intelligence and Reliability Workflows for an Early-Stage Developer Tools Platform

An Independent Technical Report

Author: Sahil San, Founding Machine Learning Engineering Intern, DevPlaza

Affiliation: DevPlaza — an early-stage developer tools and SDLC platform (San Francisco)

Status: Independent research note; not peer-reviewed

Date: 2025–2026

Suggested Citation

San, S. (2026). *Practical Lessons from Building ML-Powered Code Intelligence and Reliability Workflows for an Early-Stage Developer Tools Platform*. Independent technical report.

Abstract

The rapid proliferation of AI-assisted coding tools has amplified both developer productivity and the risks of accumulating untested, poorly integrated code. Early-stage software teams in particular face heightened exposure to what practitioners have termed "tech bankruptcy"—a state in which unchecked complexity outpaces the team's capacity to maintain reliability and shipping cadence. This report documents the design, implementation, and evaluation of two complementary ML-powered system families built during a founding internship at DevPlaza, an early-stage developer tools platform: (1) a semantic code intelligence layer supporting natural-language code search, usage discovery, and snippet recommendation, and (2) a multi-agent reliability workflow spanning pull-request analysis, CI log triage, issue clustering, and static-analysis augmentation. Taken together, these systems aim to keep development teams closer to a production-ready baseline by default. Evaluation on representative task sets suggests that hybrid retrieval strategies and structured evaluation benchmarks drive more measurable quality gains than model substitution alone. The central lessons—prioritize evaluation quality, embrace debuggable simplicity, and keep developers in the loop—apply broadly to applied ML work in resource-constrained startup environments.

1. Introduction

Modern software development is undergoing a structural shift. AI coding assistants—tools that autocomplete functions, generate boilerplate, and answer architectural questions in real time—have become mainstream. Developers can now produce substantial code volumes with minimal friction. This acceleration is double-edged: it lowers barriers to building but also compresses the feedback loops that traditionally kept complexity in check. Codebases grow faster than test suites, documentation, and institutional understanding can follow.

The phenomenon is not purely hypothetical. Internal tooling surveys and anecdotal reports from engineering teams consistently surface a cluster of compounding problems: flaky CI pipelines that nobody fully understands, issue backlogs where duplicate and stale reports bury genuine regressions, static-analysis warnings that accumulate unread, and code search that degrades as repositories grow. Left unaddressed, these signals coalesce into what DevPlaza—the platform described in this report—

calls "tech bankruptcy": a state in which the cost of change approaches or exceeds its benefit, effectively freezing development velocity.

Traditional mitigations are insufficient at speed. Grep-based code search breaks down across multi-language monorepos. Manual CI log inspection does not scale beyond a handful of daily builds. Issue triage performed by rotating on-call engineers is inconsistent and slow. What early-stage teams need is targeted, explainable AI assistance embedded at the points in the software development lifecycle (SDLC) where signal degrades fastest.

DevPlaza is designed as a "software development foundation"—a platform that integrates AI across code review, CI, issue management, and code intelligence to reduce coordination overhead and maintain production-readiness by default. This report focuses on the ML-relevant components built during a founding engineering internship: a semantic code intelligence layer and a family of reliability agents. The goal is not to present a state-of-the-art academic benchmark but to offer a grounded, experience-driven account of what works, what does not, and why—in the context of real startup constraints.

The remainder of this report proceeds as follows. Section 2 provides a high-level system overview. Sections 3 and 4 describe the code intelligence and reliability agent subsystems in detail. Section 5 presents evaluation methodology and results. Section 6 discusses key lessons and design trade-offs. Sections 7 and 8 address limitations and conclusions.

2. System Overview

The platform ingests signals from multiple sources in a typical engineering organization: source code repositories (Git), CI/CD pipeline logs, issue and project tracking (Jira-compatible), and static analysis tooling (SonarQube-compatible). Each data type presents distinct characteristics—structure, update frequency, noise level, and downstream use—that shape the design of the ML components built on top of them.

At a high level, the system is organized into four layers:

- Ingestion and indexing: repository crawlers maintain near-real-time indexes of source files, test files, commit histories, and associated metadata. CI log pipelines ingest build artifacts on job completion. Issue events are streamed from webhooks. Static analysis results are polled or pushed on a configurable cadence.
- ML services: dedicated service modules handle (a) code search and semantic retrieval, (b) commit and pull-request analysis, (c) CI log summarization and failure classification, and (d) issue clustering and triage. Each module exposes a well-defined interface used by the orchestration layer.
- Orchestration and agent layer: an orchestrator subscribes to platform events (new PR opened, build failed, issue created) and dispatches tasks to the appropriate agent or combination of agents. Agents produce structured outputs—ranked results, summaries, labels, suggested actions—that flow back to the orchestrator for collation.
- User experience surfaces: results are surfaced through a web dashboard, in-line PR review comments, Slack-compatible notifications, and a conversational code Q&A interface.

This architecture aligns conceptually, at a high level, with the public case study released by DevPlaza in collaboration with GoML, which describes a Git/PR agent, CI logs agent, Jira agent, and SonarQube agent orchestrated to improve test coverage, reduce CI failures, and accelerate incident triage. The description here reflects general design principles consistent with that public account and does not reveal proprietary implementation details.

Several cross-cutting constraints shape every design decision in this stack. First, the platform must support multi-repository environments with heterogeneous languages, frameworks, and tooling configurations. There is no assumption of a single normalized codebase. Second, labeled data is scarce at startup stage: there is no large, curated dataset of "good" and "bad" code snippets, triage decisions, or CI failure root causes. Systems must be designed to work well with small evaluation sets and to degrade gracefully rather than hallucinate confidently. Third, latency budgets are strict: code search must return results in under two seconds, and agent summaries surfaced in PR review must complete before a reviewer loses patience. Fourth, developer trust is fragile: a single clearly wrong suggestion—a hallucinated function name, a confidently wrong root cause—can undermine adoption of an otherwise useful feature.

3. ML-Powered Code Intelligence

3.1 Problem Definition

Code search is deceptively simple to prototype and surprisingly hard to ship reliably. The naive version—a full-text search over source files—works adequately on small, well-organized repositories but degrades quickly as codebases grow, languages multiply, and naming conventions diverge. Developers articulate needs in natural language that rarely matches identifier names: "how do we handle rate limiting in the auth service" does not share tokens with the function that implements it.

The goal of the code intelligence layer is to close this gap: given a natural-language query or a code snippet, return the most relevant code artifacts (functions, modules, test cases, usage examples) quickly, consistently, and with enough provenance information that developers can act on results without secondary investigation. Secondary goals include surfacing similar code for deduplication, identifying good exemplars for onboarding, and enabling a conversational Q&A interface over the codebase.

The core challenges are: (1) query-code vocabulary mismatch; (2) multi-language and multi-paradigm repositories where the same concept is expressed differently; (3) the risk of hallucinated or misleading suggestions when retrieval confidence is low; and (4) the requirement that results remain stable and explainable—a result that appears for one query variant but not another erodes developer trust.

3.2 Data and Representations

The primary indexing unit is the function or method, rather than the file. This choice reflects the developer's mental model: queries are typically about behaviors, not files. For each indexed unit, the system stores the source text, parsed metadata (language, file path, enclosing class/module, symbol name), and optionally a natural-language docstring or inline comment. Test files are indexed separately, as are README and documentation files, preserving the ability to retrieve explanatory prose alongside code.

Representations are computed at indexing time and cached. Two complementary embedding types are maintained:

- Text embeddings of the natural-language content associated with each code unit—docstrings, comments, commit messages referencing the unit, and issue descriptions linked to recent changes. These embeddings are computed using a general-purpose sentence encoder and are well-suited to natural-language queries.
- Code embeddings of the function body itself, computed using a code-specialized model (e.g., a variant fine-tuned on code corpora). These embeddings capture structural and semantic properties of the implementation rather than its documentation.

Structural signals are stored separately: file path (captures module hierarchy and naming conventions), programming language, recency (time of last commit touching the unit), and a popularity proxy (frequency of cross-file references or import count). These signals are not embedded; they serve as hard filters and soft re-ranking features.

3.3 Retrieval and Ranking Strategy

The baseline retrieval system uses BM25 over tokenized source code, which provides fast, interpretable results for queries that happen to share vocabulary with implementation details. BM25 is kept as a component rather than replaced, because it handles exact identifier lookups (e.g., a specific function name) better than embedding-based methods.

The improved hybrid system combines three signals in a two-stage pipeline:

- Stage 1 — Recall: both BM25 and approximate nearest-neighbor search over the dual embedding space (text and code) are executed in parallel. Candidates from each retriever are merged and deduplicated, targeting a recall set of 50–100 candidates for a given query.
- Stage 2 — Re-ranking: a lightweight re-ranker scores each candidate using a combination of (a) cross-encoder semantic similarity between query and candidate, (b) token overlap (a simple but effective lexical signal), (c) structural filters (language match, path affinity), and (d) recency and popularity signals. The re-ranker is a small model that can score 100 candidates in well under 100ms on CPU.

The design explicitly prioritizes debuggability. Every ranked result carries an attribution explaining which signals contributed to its placement. This is not just a UX feature; it is a diagnostic tool for the team: when a result is surprising, the attribution trace is the first place to look.

3.4 Example Workflows

Three representative use cases illustrate the system's practical scope:

- API usage discovery: A developer unfamiliar with an internal authentication library asks "show me examples of token refresh with the internal auth client." The system returns the top-ranked function bodies that invoke the refresh pattern, along with their enclosing test cases, in order of structural similarity to the canonical usage pattern.
- Error handling localization: After a production incident, a developer asks "where is the database connection timeout handled?" The hybrid system surfaces the relevant exception handler even when its identifier name contains no word from the query, because the code embedding captures the semantic relationship between "timeout" and the exception class used.
- Onboarding exemplar retrieval: A new engineer asks "give me a good example of using the job queue for background processing." The system weights recency and test coverage in re-ranking, preferring well-tested, recently maintained implementations over older, orphaned examples.

4. Reliability Agents Across the SDLC

4.1 Motivation

CI reliability and issue backlog hygiene are among the most common sources of friction reported by engineering teams. A build pipeline that fails 20% of the time non-deterministically—flaky tests, environment-dependent failures, transient network errors—trains developers to ignore failures, which is precisely when real regressions slip through. An issue tracker with thousands of open tickets, many

of them duplicates or stale, obscures the signal that would help teams prioritize. Static analysis tools surface useful warnings, but a backlog of thousands of low-severity findings trains the same learned helplessness.

The reliability agent framework is designed to intervene at each of these failure modes, not by replacing human judgment but by reducing the cost of exercising it. Each agent produces structured, citable outputs that a developer can verify in seconds rather than minutes.

4.2 Git/PR Agent

The PR agent subscribes to pull-request open and update events. On each event, it performs several analyses over the diff and its context:

- **Diff risk scoring:** the agent classifies each file change into a risk tier based on the type of modification (e.g., changes to security-sensitive paths, modifications to widely-imported utility functions, deletions of test coverage), the recency of prior changes to the same file, and the author's historical change patterns. High-risk diffs trigger an explicit annotation on the PR.
- **Test gap detection:** by cross-referencing the changed functions against the test index, the agent identifies functions that lack direct test coverage after the change. It surfaces these gaps with links to the relevant test files and suggests, at the function level, where new tests would be most valuable.
- **Suspicious region flagging:** the agent uses code embeddings to compare new code to known anti-patterns or to recently reverted changes, flagging structural similarities that warrant a closer look.

Crucially, the PR agent produces suggestions, not decisions. It annotates the pull request with structured comments but does not block merges or auto-approve. This distinction is enforced at the product level, not just by convention, because premature automation of merge gates would require a level of reliability we had not yet established.

4.3 CI Logs Agent

CI log parsing is a tedious but high-value task. A single failing build can generate tens of thousands of lines of log output; finding the root cause in that stream requires a combination of pattern recognition and contextual knowledge that is difficult to codify with simple regular expressions.

The CI logs agent operates on raw log streams and produces a structured summary per failing job:

- **Failure classification:** the agent classifies failures into a taxonomy (test assertion failure, compilation error, environment/dependency failure, flaky test pattern, timeout) using a combination of regex anchors and a small text classifier fine-tuned on a labeled sample of representative log samples.
- **Root cause localization:** for classified failures, the agent extracts the minimal log segment most likely to contain actionable information and presents it prominently, with surrounding context collapsed.
- **Remediation hints:** for known failure patterns (e.g., a specific dependency version conflict that has appeared before, a flaky test that matches a known fingerprint), the agent suggests remediation steps drawn from a curated hint library populated from resolved incidents.

The agent's output is appended to the CI notification and linked from the PR comment thread. Average time-to-triage for a failing build—the interval between receiving a failure notification and understanding its cause—is one of the primary qualitative metrics used to evaluate this agent in developer feedback sessions.

4.4 Issue/Bug Triage Agent

Issue tracking systems accumulate debt at a rate proportional to team size and release frequency. The triage agent operates over the issue stream to reduce this debt:

- Duplicate detection: new issues are embedded and compared against the open issue corpus. High-similarity pairs are surfaced for human review and potential merge, with a brief explanation of why they were flagged as similar.
- Priority prediction: a lightweight classifier estimates the likely priority of a new issue based on title, description embedding, reporter history, and related PR/commit context. The prediction is presented as a suggestion, not an assignment.
- Context summarization: for issues that have accumulated a long comment thread, the agent generates a concise summary of the current state, open questions, and last-known status—useful for engineers picking up a stale issue.

4.5 Static Analysis Agent

Static analysis tools like SonarQube surface a large volume of findings at varying severity levels. The static analysis agent adds two capabilities on top of raw findings:

- Finding triage and grouping: findings are clustered by type and file region, and the agent generates a prioritized view that emphasizes findings in recently modified code and in paths with low test coverage.
- AI-generated remediation hints: for each high-priority finding, the agent produces a brief, actionable remediation description—explaining what the finding means, why it matters, and what a correct fix would look like. This reduces the lookup burden on developers who encounter unfamiliar warning types.

4.6 Orchestration and Guardrails

An event-driven orchestrator ties these agents together. On each platform event, the orchestrator determines which agents are relevant, dispatches tasks in parallel where possible, and collates outputs for delivery. For a new PR event, for example, the orchestrator simultaneously dispatches to the PR agent and queries the static analysis agent for findings in files touched by the diff, then assembles a unified PR summary.

Guardrails are a first-class concern. No agent takes autonomous action in external systems without an explicit human confirmation. Agent outputs are annotated with confidence indicators where applicable: low-confidence suggestions are displayed with a note that they should be treated as starting points rather than conclusions. Provenance—the specific file, commit, log line, or issue that grounded a suggestion—is always surfaced so developers can verify quickly.

5. Evaluation Methodology

5.1 Code Intelligence Evaluation

Evaluating code search is inherently subjective, and the absence of a large, labeled query-result dataset at early startup stage makes formal benchmarking difficult. The approach taken was to define a small but representative set of query tasks, collect binary relevance judgments from developers, and track Success@k and median latency as primary metrics.

Five representative task types were defined:

- API usage retrieval: "Show me examples of [internal API pattern X]."
- Error handling localization: "Where is [error type Y] handled?"

- Pattern search: "Find all places where we do [operation Z]."
- Onboarding exemplar: "Give me a well-tested example of [library L in context C]."
- Cross-file dependency tracing: "What calls [function F] and why?"

For each task type, 10–20 query instances were generated using real scenarios surfaced by the development team. Developers rated each top-5 result set as "contains a useful result" (Success@5) or not. Latency was measured as end-to-end wall clock time from query submission to result rendering.

Results comparing the BM25 baseline to the hybrid retrieval system showed a 20–40 percentage-point improvement in Success@5 across task types, with the largest gains on semantic and cross-file tasks where vocabulary mismatch is most severe. Median latency remained under 1.5 seconds for the hybrid system on the evaluation set, meeting the stated budget. Qualitative developer feedback consistently identified result provenance—the ability to immediately see which file and function a result came from—as a key trust driver.

5.2 Reliability Agent Evaluation

Agent evaluation was primarily qualitative, supplemented by before/after timing comparisons for specific workflows.

For the CI logs agent, the primary outcome measured was the time developers reported spending to understand a failing build. Informal before/after comparisons across a small sample of developers suggested a meaningful reduction in this interval, with the most pronounced gains on environment and dependency failures—the failure type least amenable to developer intuition alone.

For the PR agent, feedback was solicited after each annotated PR. The most consistently valued feature was test gap detection, which developers described as surfacing blind spots they would have caught only in code review or, worse, in production. The risk scoring feature generated more mixed feedback: developers found it useful when the risk flags aligned with their own intuitions but expressed skepticism about opaque high-risk labels that lacked clear justification—reinforcing the value of attribution.

For the issue triage agent, duplicate detection was evaluated on a retrospective sample of closed issues. A majority of true duplicates in the sample were flagged by the agent with high similarity scores, though the precision of the flagged pairs was lower than recall, meaning some false positives required human review. This trade-off—high recall at the cost of some noise—is acceptable for a triage assist tool where the cost of a missed duplicate exceeds the cost of reviewing a false positive.

6. Lessons Learned and Design Trade-offs

6.1 Evaluation Quality Is the Highest-Leverage Investment

The most consistent finding from this work is that the quality of the evaluation setup—the task definitions, the relevance judgments, the metric choices—drives system quality more than model selection. Early in the project, a significant amount of effort went into exploring model variants, embedding dimensions, and retrieval algorithms. Progress was slow and hard to measure.

The turning point was defining a concrete set of query tasks with binary relevance labels provided by actual developers. With this evaluation in place, it became possible to make model and architecture decisions based on measured outcomes rather than intuition. Improvements that looked promising in isolation but did not move Success@k were deprioritized; simple changes that consistently moved the metric were shipped. This lesson generalizes well beyond code intelligence: in applied ML, the ability to measure is more valuable than the ability to model.

6.2 Simplicity and Debuggability Outperform Sophistication in Production

The hybrid retrieval system—combining BM25 with dual embedding types and a lightweight re-ranker—is not architecturally novel. Each component is individually simple and well-understood. The system's strength comes from the combination and from the care taken to make each stage debuggable.

A recurring experience during development was encountering retrieval failures—cases where the expected result did not appear in the top-5—and being able to diagnose the failure within minutes because every ranking decision was attributable to an identifiable signal. With a black-box end-to-end model, the same failure would have required retraining or guessing. In a small team without a dedicated ML infrastructure function, debuggability is not a luxury; it is a survival requirement.

The same principle applied to the reliability agents. The CI log classifier began as a small labeled dataset plus a set of regex anchors. The regex anchors handled 70–80% of cases correctly and were trivially debuggable. The classifier handled the long tail. Replacing the anchors with a neural sequence model would have produced a marginally better recall number on the evaluation set and a significantly harder-to-maintain production system.

6.3 Latency and UX Are First-Class ML Constraints

Developer trust in an ML-powered tool is acutely sensitive to latency. A code search result that takes four seconds to appear will not be used, regardless of its relevance. A PR annotation that arrives after the reviewer has already finished their pass is ignored. In practice, this means that every model choice is also a latency choice, and latency budgets must be established before retrieval architecture decisions are made, not after.

Concretely, this led to several architectural decisions: keeping re-rankers small enough to run on CPU rather than requiring GPU inference; pre-computing and caching embeddings at indexing time rather than at query time; and batching agent tasks within each orchestrator dispatch to avoid serial dependency chains. None of these decisions were obvious at the start; each was the product of profiling against real latency budgets.

6.4 Human-in-the-Loop Design Is Not Optional

Several iterations of the PR agent included features that were technically functional but generated significant developer friction. Risk scores without explanations prompted questions rather than actions. Suggested test stubs that did not compile created more work than they saved. The common thread in negative feedback was opacity: developers were willing to follow AI suggestions when they could verify the reasoning, but resistant when they could not.

The response was to shift the product framing from "AI that acts" to "AI that surfaces." Every agent output was redesigned to answer three implicit questions a developer would ask: what is being flagged, why is it being flagged, and where in the codebase should I look to verify? This reframing significantly improved adoption rates and—non-trivially—improved the quality of agent outputs, because designing for explainability forced clearer thinking about what information actually grounded each suggestion.

6.5 Startup Constraints Are a Design Input, Not a Limitation

Building ML systems at startup stage—with limited labeled data, limited infrastructure, limited team bandwidth, and a rapidly changing product surface—requires a different optimization objective than building for a mature platform. The objective is not "best possible system" but "best system that can be maintained and improved by a small team."

This led to concrete choices: using off-the-shelf embedding models rather than fine-tuning (fine-tuning requires labeled data and compute that were not available); building the evaluation harness before the first production model (so that improvements can be measured from day one); and designing agents with modular, stateless interfaces so that individual components can be swapped without cascading

refactors. These choices involve trade-offs—the system is not as accurate as it could be with more resources—but they result in a system that is actually deployable and improvable within the constraints of the environment.

7. Limitations and Future Work

7.1 Limitations

This report describes work conducted at a single early-stage platform with a small and non-representative set of codebases and developer workflows. The evaluation sets used—tens to low hundreds of labeled examples per task type—are sufficient for relative comparisons and directional signals but insufficient for statistical confidence on absolute performance claims. Results may not generalize to large enterprise environments with different repository structures, team compositions, or tooling stacks.

Several components of the current system remain heuristic-heavy. The static analysis triage agent relies on hand-crafted priority rules that have not been systematically validated. The PR risk scorer uses a manually defined taxonomy of risk signals that reflects the team's priors rather than empirically derived importance weights. These heuristics are effective within their design scope but may fail unpredictably on inputs that fall outside it.

Personalization is absent from the current design. The code intelligence layer returns the same ranked results to all developers regardless of their role, expertise level, or prior query history. A developer working on the authentication service would benefit from results biased toward that service's codebase; a new team member would benefit from results prioritizing well-documented, well-tested code. Incorporating this signal requires user modeling that has not yet been built.

Finally, the risk of confident but incorrect agent outputs—hallucinated root causes, wrong file attributions, spurious duplicate detections—is real and not fully mitigated. Current safeguards include confidence thresholds, attribution requirements, and framing outputs as suggestions rather than conclusions, but these are partial mitigations rather than solutions.

7.2 Future Work

Several directions represent natural extensions of this work:

- Active learning and feedback loops: surfacing developer disagreement with agent suggestions as a training signal, with lightweight in-product annotation affordances, would enable continuous improvement without dedicated labeling infrastructure.
- AST- and CFG-aware code understanding: current code embeddings treat source files as token sequences. Incorporating abstract syntax tree structure or control-flow graph signals would improve retrieval for structural queries (e.g., "find all functions that call X inside a try-catch block") and enable more precise test gap analysis.
- Cross-repository reasoning: the current system indexes repositories independently. Many development workflows span multiple repositories (e.g., a service and its client library). Cross-repo retrieval and dependency-aware analysis are near-term priorities.
- LLM integration with guardrails: large language models can generate richer, more context-sensitive remediation suggestions than the current template-based approach. The primary barrier is reliability: LLMs produce plausible-sounding but occasionally wrong suggestions. Integration will require confidence estimation and a clear product-level commitment to correction workflows.
- Reliability outcome tracking: current evaluation is largely process-oriented (time to triage, developer ratings). Future work should track outcome metrics—incidents avoided, mean time

to resolution before and after agent adoption—to build a more rigorous business case for each agent component.

8. Conclusion

The challenge of maintaining software quality and reliability in the age of AI-assisted development is not primarily a model problem—it is a system design and evaluation problem. Models are more capable than ever; the gap is in the infrastructure for surfacing their outputs reliably, explaining their reasoning credibly, and integrating their suggestions into developer workflows without adding friction.

This report has described two families of ML-powered systems built to address that challenge: a hybrid code intelligence layer for semantic search, usage discovery, and code Q&A, and a multi-agent reliability workflow spanning pull-request analysis, CI triage, issue management, and static analysis augmentation. The systems are not novel in their components; they are novel in their integration and in the discipline with which they were evaluated and constrained for a real, resource-limited deployment environment.

The central lessons—invest in evaluation before models, prefer debuggable simplicity over opaque sophistication, treat latency as a first-class constraint, and design for human verification rather than human replacement—are not specific to developer tools. They represent a set of applied ML engineering principles that generalize to any domain where AI suggestions must earn trust before they can drive action.

This report is a snapshot of work in progress, shared in the spirit of contributing to the community conversation on practical AI integration in software development. The problems are real, the lessons are hard-won, and the most useful future work is probably not in the models.

Acknowledgments

The author thanks the DevPlaza team for the opportunity to work on these systems in an early and formative stage. The design decisions and lessons described here reflect a collaborative process rather than individual effort.

Disclosure

This report is an independent technical note authored in the author's capacity as an intern. It describes systems and approaches at a high level consistent with publicly available DevPlaza descriptions and does not disclose proprietary implementation details, confidential customer information, or non-public business metrics. It has not been peer-reviewed and should not be cited as a formal academic publication. Any performance figures cited are approximate and intended as illustrative of directional trends; they should not be interpreted as official DevPlaza benchmarks or guarantees of system behavior.