Klaudbiusz: Agent-Agnostic Tooling for Databricks Application Generation with Trajectory-Based Optimization and Composite Evaluation

Anonymous Author(s)

Abstract

We present Klaudbiusz, an open-source, agent-agnostic toolset for autonomous generation of Databricks data applications. Rather than building a specific agent, we provide reusable infrastructure—environment scaffolding, templates, and MCP tool integrations—that any agentic coding system can leverage. To continuously improve this toolset, we introduce a trajectory analyzer that processes agent execution traces via map-reduce LLM analysis, identifying friction points and generating actionable recommendations. To measure outcomes, we develop AppEval-100, a composite evaluation score built on four pillars: Reliability, SQL Quality (inspired by BIRD/Spider), Web Quality (inspired by WebArena), and Agentic DevX. Unlike existing benchmarks that evaluate SQL or web tasks in isolation, AppEval-100 provides the first composite evaluation for full-stack data applications, mapped to industry-standard DORA delivery metrics. We design a 100-prompt benchmark with ±9% confidence intervals across three difficulty tiers. Evaluated on 20 Databricks applications, we achieve 100% build/runtime success with 6–9 minute generation latency at $0.74 per application. We release the complete framework including scaffolding tools, trajectory analyzer, evaluation harness, and MLflow integration.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Natural language processing*.

Keywords

agent-agnostic tooling, agentic code generation, evaluation framework, trajectory optimization, Databricks

ACM Reference Format:
Anonymous Author(s). 2026. Klaudbiusz: Agent-Agnostic Tooling for Databricks Application Generation with Trajectory-Based Optimization and Composite Evaluation. In *Proceedings of the ACM Conference on AI and Agentic Systems (CAIS '26), May 26–29, 2026, San Jose, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The emergence of agentic coding systems—AI agents capable of autonomously generating, testing, and deploying software applications—represents a paradigm shift in software engineering. While benchmarks like HumanEval [4] and SWE-bench [8] have advanced our understanding of code generation quality, they focus primarily on functional correctness rather than production readiness.

The Infrastructure Gap. We observe that agent performance depends heavily on the quality of surrounding infrastructure: environment scaffolding, templates, tool integrations, and validation pipelines. Yet most work focuses on improving agents themselves rather than the reusable tooling that enables them. We argue for an *agent-agnostic* approach: build excellent infrastructure that any agent can leverage.

Core Principle. Our work is guided by a simple axiom: *If an AI agent cannot autonomously deploy code using provided tooling, the tooling needs improvement—not necessarily the agent.*

Three-Pillar Approach. We address this through three interconnected contributions:

(1) **Agent-Agnostic Toolset.** We introduce Klaudbiusz, an open-source infrastructure for Databricks application generation comprising environment scaffolding, TypeScript/tRPC templates, MCP tool integrations, and containerized execution via Dagger. Any agentic system can leverage these tools.
(2) **Trajectory Analyzer.** To continuously improve this toolset, we present a map-reduce LLM approach for analyzing agent execution traces. The analyzer identifies friction points, inefficient patterns, and tool failures, generating actionable recommendations for infrastructure improvement.
(3) **Composite Evaluation (AppEval-100).** To measure outcomes, we develop a 4-pillar evaluation framework combining Reliability, SQL Quality, Web Quality, and Agentic DevX into a single composite score mapped to DORA delivery metrics.

Empirical Validation. Evaluated on 20 Databricks applications, we achieve 100% build/runtime success with 6–9 minute generation latency at $0.74 per application.

2 Related Work

We survey evaluation approaches across four categories: function-level benchmarks, repository-level agent benchmarks, interactive agent environments, and evaluation harnesses. Table 1 summarizes key frameworks and identifies the gap our work addresses.

2.1 Function-Level Code Generation Benchmarks

HumanEval [4] introduced pass@k evaluation on 164 Python functions, becoming the standard metric for code generation. **MBPP** [3] expanded to 974 problems but remains algorithm-focused. Both benchmarks are now saturated—top models achieve >90% on HumanEval—raising concerns about contamination and real-world relevance.

BigCodeBench[1] addresses these limitations with diverse library calls and complex instructions, while **LiveCodeBench**[2] provides dynamic, contamination-resistant evaluation. Recent work on **HumanEval Pro and MBPP Pro** [20] introduces self-invoking code generation to test progressive reasoning.

Gap: Function-level benchmarks do not evaluate deployment, integration, or operational readiness.

2.2 Repository-Level Agent Benchmarks

SWE-bench [8] evaluates agents on 2,294 real GitHub issues, with **SWE-bench Verified**[3] providing 500 human-verified samples where top models achieve ~72%. The harder **SWE-bench Pro**[4] reveals performance drops to ~23% on production-grade issues.

SWE-agent [18] demonstrates that custom agent-computer interfaces significantly enhance performance through Docker-based harnesses and tool augmentation. **REPOCOD**[5] tests complete method implementation with only 6% resolution rate, while **DevQualityEval**[6] evaluates multi-language software engineering tasks on private datasets to avoid contamination.

Gap: Repository-level benchmarks focus on patch correctness, not autonomous deployment capability.

2.3 Text-to-SQL Benchmarks

For data-centric applications, text-to-SQL benchmarks provide critical evaluation capabilities. **Spider** [19] introduced cross-domain evaluation with 10,181 questions across 200 databases, establishing difficulty tiers (easy/medium/hard/extra-hard). **BIRD** [13] scales to 12,751 question-SQL pairs across 95 databases (33.4GB), introducing the *Valid Efficiency Score (VES)* that rewards both correctness and query efficiency.

Spider 2.0 [12] targets enterprise workflows with 632 real-world tasks involving BigQuery and Snowflake, where even o1-preview achieves only 21.3% (vs 91.2% on Spider 1.0). This dramatic performance gap highlights the challenge of production-grade SQL generation.

Gap: Text-to-SQL benchmarks evaluate query correctness in isolation, not within deployed applications with UI, APIs, and DevOps requirements.

2.4 Data Analytics Agent Benchmarks

Recent benchmarks evaluate agents on end-to-end data analysis. **Tapilot-Crossing** [14] provides 1,024 human-machine interactions for interactive data analysis, testing multi-turn reasoning and visualization. **InfiAgent-DABench** [7] offers 311 questions across 55 datasets for data analysis scenarios.

InsightBench [17] evaluates 100 business analytics tasks requiring insight generation—the closest to our target domain. **DS-1000** [11] benchmarks data science code generation across NumPy, Pandas, and other libraries with 1,000 problems.

Gap: Data analytics benchmarks focus on insight generation or code correctness, not full-stack application deployment with Databricks integration.

2.5 Interactive Agent Benchmarks

WebArena [22] evaluates 812 web tasks across e-commerce, forums, and content management, where best agents achieve 61.7% versus 78% human performance. **GAIA** [16] tests general AI assistants on 466 multi-step reasoning questions requiring tool use, with agents reaching 80.7% versus 92% human baseline.

AgentBench [15] spans 8 environments (OS, databases, web) revealing significant gaps between commercial and open-source models. **OSWorld**[7] (NeurIPS 2024) benchmarks multimodal agents in real computer environments, where recent advances have achieved superhuman performance (76% vs 72% human baseline).

Gap: Interactive benchmarks evaluate general agent capabilities, not software deployment pipelines.

2.6 Agent Evaluation Harnesses and Frameworks

Inspect AI[8] from the UK AI Safety Institute provides 100+ pre-built evaluations with sandboxing, MCP tool support, and multi-agent primitives. It has been adopted by frontier labs and safety organizations for standardized agent evaluation.

DeepEval[9] offers CI/CD integration with LLM-as-judge metrics including task completion, tool correctness, and hallucination detection. **Databricks Agent Evaluation** integrates with MLflow for tracking groundedness, correctness, and coherence of agentic applications.

The emerging **AgentOps** paradigm extends DevOps principles to AI agents, addressing observability, tracing, and lifecycle management specific to autonomous systems.

2.7 DevOps and Deployment Metrics

DORA metrics [6]—deployment frequency, lead time, change failure rate, and mean time to restore—provide industry-standard measures of software delivery performance. **MLOps 2.0** architectures integrate CI/CD with Continuous Data Validation (CDV) for reliable ML delivery.

Gap: No existing framework combines code generation evaluation with DORA-mapped deployment metrics and agentic DevX scores (runability, deployability).

[1]https://github.com/bigcode-project/bigcodebench
[2]https://livecodebench.github.io/
[3]https://openai.com/index/introducing-swe-bench-verified/
[4]https://scale.com/blog/swe-bench-pro
[5]https://arxiv.org/abs/2410.21647
[6]https://github.com/symflower/eval-dev-quality
[7]https://os-world.github.io/
[8]https://inspect.aisi.org.uk/
[9]https://github.com/confident-ai/deepeval

Table 1: Comparison of agent evaluation approaches. Klaudbiusz uniquely combines deployment-centric metrics with DORA mapping and trajectory-based optimization.

Framework	Code Gen	Deploy	DORA	DevX	Trajectory
HumanEval/MBPP	✓	–	–	–	–
SWE-bench	✓	–	–	–	–
WebArena/GAIA	–	–	–	–	–
Inspect AI	✓	–	–	–	–
DeepEval	✓	–	–	–	–
Klaudbiusz (Ours)	✓	✓	✓	✓	✓

3 Installable Domain Knowledge

3.1 Motivation

Developers already use capable agents—Claude Code, Cursor, Codex. Rather than asking them to adopt yet another tool, we bring domain knowledge to the agents they already use. A user installs our package into their existing environment; the agent gains domain expertise without workflow changes. However, capable agents often spend excessive steps on fixing issues when domain-specific guidance is missing.

This approach has a practical advantage: we leverage the ecosystem of existing agents rather than competing with them. The alternative—building a custom agent per domain—doesn't scale and, in our view, creates vendor lock-in.

Scope: Our current implementation targets data-centric web applications on Databricks, validating the approach on a concrete domain before generalizing.

3.2 Architecture

The package has three components: context layers that inject domain knowledge progressively, tools exposed via CLI, and a state machine that enforces validation before deployment.

3.2.1 Context Layers (Injected Progressively). Agents have limited context windows. Dumping all domain knowledge upfront wastes tokens and can confuse the model. Instead, we inject context progressively—each layer activates when relevant. This is especially important when context is assembled from multiple parts.

Table 2: Context layer injection strategy

Layer	Content	When Injected
L0: Tools	Tool names/descriptions	Always (protocol-level)
L1: Workflow	Patterns, CLI usage, validation rules	On first discovery call
L2: Target	App vs job vs pipeline constraints	When target type detected
L3: Template	SDK patterns, examples	After scaffolding or from CLAUDE.md

For example, an agent scaffolding a new app receives L0–L2 initially. Only after scaffolding completes does L3 activate—providing SDK-specific patterns like "how to draw charts with Recharts" or "tRPC router conventions". For existing projects, the agent reads CLAUDE.md (placed in the project root) to acquire L3 context.

Implementation: Context layers are implemented as `.mdc` rule files in `.cursor/rules/` directories. Each file specifies glob patterns for activation and contains domain-specific guidance. For example, `database-queries.mdc` provides Drizzle ORM patterns:

```
# database-queries.mdc (excerpt)
- Use proper Drizzle operators: eq(), gte(),
  desc() from 'drizzle-orm'
- Never use direct comparisons like
  table.column === value
- For conditional queries, build step-by-step:
  let query = db.select().from(table);
  const conditions: SQL<unknown>[] = [];
  if (filter) {
    conditions.push(eq(table.field, filter));
  }
  query = query.where(and(...conditions));
```

These rules are *installable*—users add them to their project, and compatible agents (Cursor, Claude Code) automatically load them based on file patterns. Lines of code for the complete scaffolding: ~2,400 across 14 rule files.

3.2.2 Tools (Exposed via CLI). We expose domain functionality through CLI commands—a pattern that Cloudflare ("Code Mode") [5] and Anthropic [2] found effective, reporting that LLMs perform better writing code to call CLI tools than invoking tools directly.

Lifecycle commands: `scaffold` (creates project from template with CLAUDE.md guidance), `validate` (builds in Docker, captures Playwright screenshots), `deploy` (to target platform).

Data exploration: Commands for discovering available data, with agent-friendly additions: batch operations that bundle multiple queries, clearer error messages, and syntax examples for platform-specific SQL variations.

Workspace tools (read/write/edit, grep, glob, bash) are not our contribution—agents already have these. Our package adds domain-specific capabilities on top.

3.2.3 State Machine. In our design, applications cannot deploy unless they pass validation after their most recent modification:

$$\texttt{Scaffolded} \xrightarrow{\text{edit}} \texttt{Modified} \xrightarrow{\text{validate}} \texttt{Validated(checksum)} \xrightarrow{\text{deploy}} \texttt{Deployed}$$

How the checksum works: The checksum captures file state at validation time using MD5 hashes of critical files (e.g., `App.tsx`, `schema.ts`). Any change after validation requires re-validation. This prevents untested code deployment—a common failure mode when agents skip validation. Implementation: `check_template_failed()` in our analysis code computes `hashlib.md5(content).hexdigest()` and compares against known template hashes to detect generation failures.

3.3 Agent Compatibility

To validate agent-agnosticism, we tested the package across multiple backends. The key requirement is function calling capability—any agent that can invoke tools works with our package.

The LiteLLM backend demonstrates that the approach isn't tied to specific vendors—we wrap any model with function calling into our generation pipeline. Production validation with Qwen3-Coder-480B achieved 70% success rate at $0.61/app (vs 86.7% at $5.01/app for Claude Sonnet 4).

Table 3: Agent backend compatibility validation

Backend	Validation	Notes
Claude Agent SDK	Automated	Primary production use
Cursor	Manual	IDE integration
Codex	Manual	Alternative agent
LiteLLM + Qwen3	Automated	Open-source models

4 The Klaudbiusz Toolset

4.1 Agent-Agnostic Design

Klaudbiusz provides reusable infrastructure that any agentic coding system can leverage, rather than a specific agent implementation. The toolset comprises:

- **Environment Scaffolding.** Pre-configured TypeScript + tRPC project templates with Databricks SDK integration, ensuring consistent structure across generated applications. Templates are *pluggable*—users can swap template sets for different stacks.
- **MCP Tool Integrations.** Model Context Protocol tools for file operations, database queries, and deployment actions that agents can invoke.
- **Containerized Execution.** Dagger-based sandboxed builds providing isolation and reproducibility across environments.
- **Validation Pipelines.** Automated checks for build, runtime, type safety, and deployment readiness.

4.2 Evaluation Design Principles

Our evaluation framework is built on two core principles:

Zero-Bias Metrics. All metrics are objective, reproducible, and automatable. We explicitly exclude subjective assessments of code quality, maintainability, or aesthetics.

Tooling-Centric Feedback. When agents fail, we ask "what tooling improvement would help?" rather than "what's wrong with the agent?" This framing drives continuous infrastructure improvement.

4.3 The 13-Metric Rubric

We organize our metrics into four categories spanning core functionality, platform integration, agentic DevX, and generation efficiency.

4.3.1 *Core Functionality (L1–L4, Binary).*

- **L1: Build Success.** Project compiles; `docker build` exits with code 0.
- **L2: Runtime Success.** App starts and serves content; health check responds within 30s.
- **L3: Type Safety.** `npx tsc --noEmit` passes with zero errors.
- **L4: Tests Pass.** Unit/integration tests pass with coverage $\geq$70%.

4.3.2 *Platform Integration (L5–L7, Binary).*

- **L5: DB Connectivity.** Databricks connection works; queries execute without errors.
- **L6: Data Operations.** CRUD operations return correct data from tRPC procedures.
- **L7: UI Validation.** Frontend renders without errors (VLM verification).

4.3.3 *Agentic DevX (D8–D9, 0–5 Score).*

These metrics capture what agents need but humans can work around. Consider an agent that generates a working application. A human developer can run it: they'll figure out missing environment variables, install unlisted dependencies, work around unclear documentation. An agent *can sometimes* do this too, but it's slow and inefficient—spending many turns on trial-and-error that explicit configuration would eliminate. The agent needs explicit `.env.example` files, documented commands, and health endpoints for verification.

- **D8: Runability.** Can a sample AI agent run generated apps locally?
  - 0: install/start fails; missing scripts/env
  - 1–2: starts with manual tweaks
  - 3: starts cleanly with .env.example + documented steps
  - 4: starts with seeds/migrations via scripts
  - 5: + healthcheck endpoint + smoke test succeeds
- **D9: Deployability.** Can a sample AI agent deploy a generated app?
  - 0: no/broken Dockerfile
  - 1–2: image builds; container fails or healthcheck fails
  - 3: healthcheck OK; smoke 2xx
  - 4: + logs/metrics hooks present
  - 5: + automated rollback to prior known-good tag

Why This Matters. Existing metrics miss this distinction. Build success (binary) doesn't capture whether the build process is agent-friendly. We need metrics that ask: can another agent operate this output? This is a novel evaluation dimension absent from existing benchmarks—critical for compound AI systems where one agent's output becomes another's input.

Implementation. D8 and D9 are currently evaluated as *artifact-based proxy metrics*: checklist-style checks for the presence and correctness of artifacts[10] (README with setup instructions, `.env.example`, `npm start` script, Dockerfile, multi-stage build, `HEALTHCHECK`, absence of hardcoded secrets). These are necessary but not sufficient conditions—a README can exist but be incorrect.

To address this gap, we plan to augment proxy checks with *statistical agent simulation*: running the same task (install, start, deploy) with multiple eval agents configured with diverse developer personas (junior frontend, senior backend, data scientist, AI agent with different capability levels). The score becomes the success rate across persona-runs (e.g., 4/5 succeed $\longrightarrow$ 0.8). Variance across personas indicates documentation quality: if only the senior backend persona succeeds, the setup instructions are insufficient. At Haiku-level costs (~\$0.001/attempt), running 15 attempts per app (5 personas $\times$ 3 runs) costs \$0.015/app—negligible compared to generation costs.

4.3.4 *Efficiency Metrics (E10–E13, Numeric).*

- **E10: Tokens Used.** Total tokens (prompt + completion) for generation.
- **E11: Generation Time.** Time spent generating application (seconds).

[10] https://github.com/neondatabase/appdotbuild-agent/blob/main/klaudbiusz/cli/evaluation/evaluate_app.py#L590-L740

- **E12: Agent Turns.** Number of conversation turns during generation.
- **E13: LOC.** Lines of code in generated application.

4.3.5 SQL Quality Pillar (S1–S4). Inspired by BIRD [13] and Spider [19], we evaluate SQL quality *in situ*—within generated full-stack applications rather than against isolated query benchmarks. SQL queries are extracted from generated code[11] (inline from TypeScript for tRPC apps, or from `config/queries/*.sql` for DBX SDK apps) and evaluated:

- **S1: Execution Correctness (EX).** Fraction of extracted SQL queries that execute without error against Databricks and return non-empty results. Range: $[0, 1]$.
- **S2: Valid Efficiency Score (VES).** Adapted from BIRD [13]: rewards both correctness *and* query efficiency relative to a reference solution. VES = correctness $\times \min(1, t_{\text{ref}}/t_{\text{actual}})$. Range: $[0, 1]$.
- **S3: Query Complexity.** Distribution across difficulty tiers (easy/medium/hard/extra-hard) based on SQL AST features (JOINs, subqueries, window functions) via `sqlglot`. Reported as context, not scored.
- **S4: SQL Safety.** Static analysis for destructive operations (`DROP`, `TRUNCATE`, `DELETE` without `WHERE`), parameterization, and injection resistance. Range: $[0, 1]$.

The key distinction from existing text-to-SQL benchmarks is that S1–S4 evaluate queries in their deployment context—connected to real Databricks schemas, embedded in application logic, and subject to runtime constraints. The extraction infrastructure is implemented; automated scoring against Databricks is planned for the 100-prompt benchmark.

4.3.6 Web Quality Pillar (W1–W4). Inspired by WebArena [22] and VisualWebArena [10], we design a *use-case-based* web evaluation framework with accessibility-tagged acceptance criteria:

- **W1: Task Completion Rate.** Per-prompt acceptance criteria generated as testable user stories (e.g., for a "customer churn dashboard": can the user filter by date range? does the table show customer data?). Evaluated via Playwright test scripts. Range: $[0, 1]$.
- **W2: Visual Rendering Correctness.** Extends our current binary VLM check[12] to a structured multi-aspect rubric: charts render with labels, layout is responsive, no overlapping elements, text is readable. Range: $[0, 1]$.
- **W3: Interactive Element Functionality.** Playwright-based interaction testing: click buttons, fill forms, toggle filters, verify DOM changes and network requests. Range: $[0, 1]$.
- **W4: Accessibility Score.** WCAG 2.1 AA compliance via `axe-core` integration with Playwright. Measures: keyboard navigation, ARIA labels, contrast ratios, focus indicators. Score: $1 - (v_{\text{critical}}/n_{\text{elements}})$, clamped to $[0, 1]$.

Acceptance criteria map to WCAG categories (Perceivable, Operable, Understandable, Robust), enabling systematic accessibility tracking across generated applications. The current implementation provides binary UI validation (L7); the full W pillar is designed for the 100-prompt benchmark expansion.

[11]https://github.com/neondatabase/appdotbuild-agent/blob/main/klaudbiusz/cli/evaluation/eval_checks.py#L150-L190

[12]https://github.com/neondatabase/appdotbuild-agent/blob/main/klaudbiusz/cli/evaluation/evaluate_app.py#L509-L588

4.4 AppEval-100 Composite Score

To enable automatic, comparable measurement across prompts and runs, we introduce **AppEval-100**—a single numeric index representing normalized readiness and agentic operability on a 0–100 scale. Unlike existing benchmarks that evaluate SQL correctness (BIRD/Spider) or web task completion (WebArena) in isolation, AppEval-100 provides the first composite evaluation combining all four pillars for full-stack data applications.

Step 1: Reliability Pillar (R). Aggregate core runtime checks:

$$R = \text{GM}(b_{\text{build}}, b_{\text{runtime}}, b_{\text{type}}, b_{\text{tests}})$$

Step 2: SQL Quality Pillar (S). Weighted combination of SQL metrics:

$$S = 0.50 \times S_1 + 0.30 \times S_2 + 0.20 \times S_4$$

where S_1 (execution correctness) dominates, S_2 (efficiency) contributes secondary value, and S_4 (safety) ensures secure queries.

Step 3: Web Quality Pillar (W). Weighted combination of UI/UX metrics:

$$W = 0.40 \times W_1 + 0.30 \times W_2 + 0.20 \times W_3 + 0.10 \times W_4$$

prioritizing task completion and visual correctness.

Step 4: Agentic DevX Pillar (D).

$$D = \text{GM}(\hat{x}_{\text{run}}, \hat{x}_{\text{deploy}})$$

where $\hat{x}$ = score/5 normalizes 0–5 scores to $[0, 1]$.

Step 5: Soft Penalty Gate. Penalize critical outages without collapsing to zero:

$$G = (0.25{+}0.75{\times}b_{\text{build}}){\times}(0.25{+}0.75{\times}b_{\text{runtime}}){\times}(0.50{+}0.50{\times}b_{\text{db}}){\times}(0.50{+}0.50{\times}\mathbf{1}_{S_1 \geq}$$

Step 6: Final Composite.

$$\textbf{AppEval-100} = 100 \times (0.30 \times R + 0.25 \times S + 0.25 \times W + 0.20 \times D) \times G$$

Pillar Weight Rationale: Reliability (30%) ensures the app works; SQL Quality (25%) and Web Quality (25%) capture the core value proposition for Databricks data applications; DevX (20%) measures autonomous operability.

Values near 100 denote near-perfect readiness; 50–70 indicates partial operability; <30 signifies fundamental execution issues.

4.5 DORA Metrics Mapping

We map our metrics to industry-standard DORA [6] measures:

- **Deployment Frequency:** Count of successful D9 events per app per evaluation cohort.
- **Lead Time:** Median time from first model call to successful D9 deployment.
- **Change Failure Rate:** Fraction of deployments that fail healthcheck or rollback within 30 min.
- **MTTR:** Median time from failure detection to restore (prior healthy image running).

Production Gate: L1–L7 pass, D8≥4, D9≥4, type-safety pass, and DORA guardrails (Lead Time P50 ≤10 min, CFR ≤15%, MTTR ≤15 min).

Statistical DevX and DORA. With statistical agent simulation (Section 4.3.3), DORA metrics gain direct empirical grounding:

Change Failure Rate becomes $1 - D9_{\text{statistical}}$ (fraction of persona-runs that fail deployment), and Lead Time becomes the median time across personas from `git clone` to successful health check. Variance across personas measures predictability—a key DORA dimension.

Validation plan. To verify that AppEval-100 predicts real-world delivery performance, we plan to deploy generated applications to staging environments and track actual DORA metrics over 30 days, correlating with evaluation scores. Our hypothesis: AppEval-100 > 70 correlates with DORA "High" performance (deployment frequency: daily, lead time: <1 day, CFR: <15%, MTTR: <1 hour).

4.6 MLflow Integration

We integrate with Databricks Managed MLflow for experiment tracking:

- Automatic metric logging per evaluation run
- Trend analysis across model versions and configurations
- Artifact versioning for reproducibility
- DORA telemetry for delivery performance monitoring

5 Agentic Trajectory Analyzer

5.1 Role in the Feedback Loop

To run trajectory analysis at scale, we built infrastructure for bulk app generation that saves execution traces in a structured format. In production, users work with their own agents (Cursor, Claude Code) which may not save trajectories in our format—but the analyzer works with any trace data that captures tool calls and results.

The analyzer consumes trajectories and recommends package improvements. This closes the loop: agents struggle → we see it in trajectories → we fix the tooling → agents struggle less.

5.2 Why Trajectories, Not Just Outcomes

End-state metrics (build success, test pass) don't reveal causes:

- Model limitations (reasoning, instruction following)?
- Tool problems (unclear descriptions, missing functionality)?
- Template issues (incorrect scaffolding, missing guidance)?
- Prompt issues (underspecified requirements, contradicting constraints)?

Trajectories—the sequence of reasoning, tool calls, and results—show where things went wrong. **Example:** An agent retrying the same malformed SQL five times reveals a missing example in the guidance. An agent calling N tools for N tables reveals a missing batch operation.

5.3 Two-Phase Architecture

We employ a map-reduce approach optimized for cost and quality:

Map Phase (Cheap Model). Each trajectory is processed independently by a fast model (we use Claude Haiku, ~$0.001/trajectory), extracting:

- Errors and retries
- Confusion patterns (agent asking clarifying questions)
- Inefficiency (suboptimal tool sequences)
- **Repetitions** (same action attempted multiple times)

This runs in parallel across all trajectories.

Agentic Synthesis Phase (Reasoning Model). Aggregated patterns go to a reasoning model (we use Claude Opus) with read-only access to:

- Template and CLI tools source code (via Read/Glob/Grep)
- Tool definitions (extracted from MCP server)
- Evaluation metrics (per-app scores, optional)

This is a full agent with up to 50 turns of exploration. If trajectories show SQL confusion, the agent greps templates for SQL examples. If tool descriptions seem unclear, it reads implementations. Context is discovered progressively as patterns demand.

Extensibility. The architecture naturally extends to new context sources. We started with trajectories only, then added template source code access, then tool definitions extracted via the MCP binary. Adding new sources (e.g., user feedback, production logs) requires only pointing the synthesis agent at additional files.

Example trace through synthesis: Given 20 trajectories where agents repeatedly fail on `QUALIFY` syntax:

1. Map phase extracts: "SQL error on QUALIFY clause" (15 occurrences)
2. Synthesis agent searches: `grep -r "QUALIFY" templates/`
3. Finds: no QUALIFY examples in SQL guidance
4. Recommendation: "Add QUALIFY, PIVOT syntax to SQL guidance"

5.4 Concrete Improvements

The analyzer identified issues leading to fixes we implemented:

Table 4: Trajectory-identified improvements (implemented)

Pattern Observed	Diagnosis	Fix Applied
N separate calls for N tables	Missing batch operation	Added `discover_schema` batch command
Agents expecting list, got search	Confusing tool name	Renamed `list_tables` → `find_tables`
Repeated SQL syntax errors	Missing examples	Added QUALIFY, PIVOT syntax to guidance
Retries on malformed errors	Unclear error messages	Added contextual parameter messages

These aren't hypothetical—they're actual fixes derived from trajectory analysis and committed to the codebase. Evidence: commit history in `github.com/neondatabase/appdotbuild-agent`.

5.5 Cost Model

For N trajectories:

- Map: N × ~$0.001 (cheap model)
- Synthesis: 1 × ~$0.5–3 (reasoning model, bounded at 50 turns)

Total scales linearly but remains bounded. For 20 apps, analysis cost was under $15.

5.6 Future Direction

Our current approach is semi-automatic: the analyzer outputs recommendations, but a human reviews them and decides which to implement. This keeps a human in the loop for changes to production tooling.

Recent work on reflective prompt evolution (GEPA [21]) shows prompts can be automatically optimized through self-reflection. DSPy [9] demonstrates similar techniques for prompt tuning. These techniques could close this gap—automatically applying fixes, measuring improvement, and iterating without human intervention. The analyzer's recommendations target tool descriptions, prompts, and examples—artifacts amenable to such techniques.

5.7 Feedback Loop

The trajectory analyzer enables a continuous improvement cycle with optional regression testing:

Generate ⟶ Evaluate ⟶ Analyze Trajectories ⟶ Improve Scaffolding ⟶ Regression Suite ⟶ Repeat

6 Experimental Setup

6.1 AppEval-100 Benchmark Design

We design a 100-prompt benchmark targeting statistical validity for Databricks data application evaluation. Our sample size provides ±9% confidence interval at 95% confidence for binary metrics ($p = 0.7$), matching InsightBench [17] (100 tasks) and exceeding Spider 2.0's [12] focused enterprise subset (632 tasks).

Table 5: Prompt difficulty distribution ($n = 100$)

Tier	Count	Description
Simple	40	Single-entity CRUD, basic dashboards, one data source
Medium	40	Multi-entity JOINs, filters, interactive charts, 2–3 data sources
Hard	20	Complex analytics, multi-step workflows, real-time updates

6.1.1 *Difficulty Distribution.*

6.1.2 Domain Coverage. Prompts span five application domains: Analytics Dashboards (26%), CRUD Applications (22%), Data Visualization (22%), Business Intelligence (20%), and Reporting Tools (10%).

6.1.3 Schema Families. We target six schema families to ensure diversity: TPC-DS (25 prompts, retail/customer analytics), TPC-H (20, supply chain), NYC Taxi (15, trip analytics), Custom Databricks (25, Unity Catalog/ML features), Financial (10, trading/risk), and IoT/Telemetry (5, device metrics).

6.2 Prompt Collection Methodology

Prompts are collected from three sources to ensure realistic ambiguity and domain vocabulary:

(1) **Hackathon recordings** (50 prompts): Extracted from video transcripts of internal Databricks application hackathons, preserving natural language vagueness.
(2) **Production logs** (30 prompts): Anonymized user requests from app.build, capturing real-world requirements.
(3) **Synthetic generation** (20 prompts): LLM-generated from templates to fill coverage gaps.

Quality Criteria: Prompts must exhibit realistic ambiguity, domain-specific vocabulary, implicit requirements (unstated but expected features), and achievability with current templates.

6.3 Current Evaluation Dataset

For this paper, we evaluate on a "Simple 20" subset—20 Databricks data application prompts spanning dashboards, analytics, and business intelligence tools—retained as a fixed regression set for longitudinal comparison.

6.4 Generation Pipeline

Applications are generated using:

- **Claude Agent SDK** with edda MCP for tool orchestration
- **Dagger** containerized execution for isolation and reproducibility
- **Environment scaffolding** with templates and validation pipelines

6.5 Generation–Evaluation Isolation

A critical design requirement is that the generator and evaluator must be *strictly isolated.* Without isolation, the generator agent can exploit evaluation artifacts—a form of reward hacking [1] that inflates metrics without improving application quality. We describe the threat model, our mitigation, and implications for the field.

6.5.1 Threat Model: Metric Gaming Through Code Leakage. Agentic coding systems actively explore their filesystem via tools like `Read`, `Glob`, and `Grep`. When evaluation code resides in the same workspace as generation code, the following attack surfaces emerge:[13]

Scenario 1: Eval check reverse-engineering. Our DB connectivity check (`eval_checks.py:39-70`[14]) verifies connectivity by calling tRPC endpoints and parsing JSON responses for a `result` key. An agent with read access could discover this check and create minimal stubs that pass the binary metric without implementing genuine Databricks queries:

```
// Hypothetical stub (passes L5 check, no real data)
healthcheck: publicProcedure.query(() => ({
  result: { data: [], status: "ok" }
}))
```

Scenario 2: Composite score optimization. The AppEval-100 formula and pillar weights are defined in `eval_metrics.py:44-88`[15]. An agent that reads this file learns that Reliability (30%) and DevX (20%) are weighted separately, and could prioritize boilerplate (`.env.example`, `Dockerfile`) for DevX points while neglecting SQL quality metrics that require genuine implementation.

Scenario 3: Pattern copying from prior runs. If the `app/` directory from prior generations is accessible, an agent could copy file structures verbatim from successful runs rather than reasoning

[13]Examples are hypothetical threat scenarios that motivated our isolation design. We observed agents reading non-generation files during early development, which led to the container-level isolation described in Section 6.5.2.
[14]https://github.com/neondatabase/appdotbuild-agent/blob/main/klaudbiusz/cli/evaluation/eval_checks.py#L39-L70
[15]https://github.com/neondatabase/appdotbuild-agent/blob/main/klaudbiusz/cli/evaluation/eval_metrics.py#L44-L88

about current prompt requirements—producing high-scoring but homogeneous applications that fail on novel prompts.

Scenario 4: Trajectory-informed shortcuts. With access to prior trajectory logs, an agent could identify which tool sequences led to passing evaluations and replay them mechanically, bypassing exploratory reasoning.

6.5.2 Container-Level Isolation via Dagger. We enforce isolation at the container build level. The generation Dockerfile[16] selectively copies only generation-related code, explicitly excluding evaluation:

```
# Exclude evaluation to prevent reward hacking
COPY cli/generation/    ./cli/generation/
COPY cli/utils/         ./cli/utils/
# NOT copied: cli/evaluation/,
#             cli/analyze_trajectories.py
```

The Dagger build context further excludes runtime artifacts[17]:

```
context = client.host().directory(".",
    exclude=["app/", "app-eval/",
             "results/", ".venv/", ".git/"])
```

Each generation starts from a clean workspace. In bulk runs, a pre-generation directory snapshot[18] prevents cross-contamination between parallel generations.

Table 6: Isolation boundaries between generation and evaluation.

Artifact	Generator	Evaluator
Domain package (.mdc rules)	✓	–
MCP tools (Databricks CLI)	✓	–
Installed skills	✓	–
Evaluation checks / metrics	–	✓
Prior generated apps / results	–	✓
Trajectory analyzer	–	✓
Current app code	writes	reads
Current trajectory	writes	reads

This separation mirrors training/test splits in machine learning: the generator learns from the domain package (training signal), while the evaluator measures generalization against unseen criteria. Improvements flow exclusively through the domain package—not through gaming the evaluation protocol.

6.5.3 Implications for Agent Evaluation. Our isolation design addresses a class of vulnerabilities unique to agentic systems: (1) agents actively explore their filesystem and will read evaluation code if accessible; (2) binary metrics are especially vulnerable to trivial stubs that pass without useful output; (3) composite score formulas, when known, enable strategic effort allocation that maximizes score over quality; (4) access to prior outputs enables pattern matching instead of reasoning.

We recommend that agent evaluation frameworks enforce container-level or filesystem-level isolation between generation and evaluation as a baseline requirement.

6.6 Statistical Validity

Our 100-prompt benchmark provides:

- **Confidence Interval:** ±9% at 95% confidence for binary metrics
- **Statistical Power:** 0.80 for medium effect size ($d = 0.5$)
- **Stratification:** Three difficulty tiers adequately sampled (40/40/20)

The "Simple 20" regression set enables longitudinal model comparisons while guarding against prompt inflation effects.

7 Results

7.1 Overall Performance

We evaluate on the "Simple 20" prompt set—20 Databricks data application prompts spanning dashboards, analytics, and business intelligence tools.

Table 7: Aggregate evaluation results (n = 20 applications, Evals 2.0)

ID	Metric	Result	Notes
L1	Build Success	20/20	100% pass
L2	Runtime Success	20/20	100% pass
L3	Type Safety	1/20	5% pass (improvement needed)
L4	Tests Pass	–	Not yet instrumented
L5	DB Connectivity	18/20	90% pass
L6	Data Operations	–	Requires app-specific procedures
L7	UI Validation	–	VLM check in progress
D8	Runability	3.0/5	Average score
D9	Deployability	2.5/5	Average score

Key Finding: 100% of generated applications achieve build and runtime success, with 90% achieving functional Databricks connectivity. However, type safety (5%) and agentic DevX scores (3.0/5, 2.5/5) indicate room for improvement toward production readiness.

7.2 Generation Efficiency Metrics

Table 8: Efficiency metrics (n = 20 applications)

Metric	Value	Notes
E10: Total Tokens	16K/app	Prompt + completion
E11: Generation Time	6–9 min	End-to-end
E12: Agent Turns	93 avg	Conversation turns
E13: LOC	732 avg	Lines of code
Cost per App	$0.74	API cost
Total Cost (20 apps)	$14.81	–
Build Step Time	2.7s avg	Docker build

[16] https://github.com/neondatabase/appdotbuild-agent/blob/main/klaudbiusz/Dockerfile#L58-L64
[17] https://github.com/neondatabase/appdotbuild-agent/blob/main/klaudbiusz/cli/generation/dagger_run.py#L206-L212
[18] https://github.com/neondatabase/appdotbuild-agent/blob/main/klaudbiusz/cli/generation/container_runner.py#L76

7.3 Comparison: Evals 1.0 vs 2.0

Table 9: Evolution from manual (Evals 1.0) to automated (Evals 2.0) evaluation

Aspect	Evals 1.0	Evals 2.0
Viability Rate	73% (30 apps)	100% build/runtime
Time to Deploy	30–60 min	6–9 min
Evaluation Method	Manual rubric	Automated pipeline
Metrics Tracked	Binary viability	13 metrics + AppEval-100
Reproducibility	Low	Full artifact pack

7.4 Production Readiness Assessment

Current status: **below production threshold**. To reach Production Candidate level:

- L3 Type Safety: 5% $\longrightarrow$ target $\geq$90%
- D8 Runability: 3.0 $\longrightarrow$ target $\geq$4
- D9 Deployability: 2.5 $\longrightarrow$ target $\geq$4
- DORA guardrails: Lead Time P50 $\leq$10m, CFR $\leq$15%, MTTR $\leq$15m

7.5 Trajectory Optimizer Insights

Analysis of agent trajectories revealed common friction patterns:

- **SQL Syntax:** Databricks SQL variations causing query failures
- **Error Handling:** Missing error handling in template scaffolding
- **Tool Descriptions:** Unclear MCP tool descriptions leading to incorrect usage
- **Type Inference:** TypeScript strict mode violations in generated code

These insights feed back into template and tool improvements via the optimize $\longrightarrow$ evaluate $\longrightarrow$ analyze cycle.

8 Discussion

8.1 Limitations

Platform Specificity. Our current implementation targets Databricks applications. Extending to other platforms requires platform-specific metrics (e.g., AWS Lambda, Vercel).

Binary Metrics. Several metrics are binary, potentially missing nuanced quality differences. Future work could introduce continuous variants.

Dataset Size. Our evaluation of 20 applications provides initial validation but may not capture edge cases. Scaling to larger datasets is ongoing.

8.2 Broader Impact

By establishing standardized metrics for autonomous deployability, we enable:

- Reproducible benchmarking of agentic code generation systems
- Objective comparison across different approaches
- Systematic improvement through trajectory-based feedback

9 Conclusion

We presented Klaudbiusz, an open-source, agent-agnostic toolset for autonomous Databricks application generation. Rather than building a specific agent, we provide reusable infrastructure—scaffolding, templates, and MCP tools—that any agentic system can leverage. Our trajectory analyzer enables continuous improvement of this toolset by identifying friction points in agent execution traces. AppEval-100 provides composite evaluation combining Reliability, SQL Quality, Web Quality, and Agentic DevX, mapped to industry-standard DORA metrics.

The path to reliable agentic code generation requires not just better models, but better tooling. We release Klaudbiusz to enable the community to build and improve agent-agnostic infrastructure systematically.

Open Source Release. Toolset, trajectory analyzer, and evaluation harness available at: [URL redacted for review]

References

[1] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. 2016. Concrete Problems in AI Safety. *arXiv preprint arXiv:1606.06565* (2016).

[2] Anthropic. 2024. Tool Use with Claude. https://docs.anthropic.com/claude/docs/tool-use.

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

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[5] Cloudflare Inc. 2024. Cloudflare Workers AI: Code Mode for Better Tool Use. https://developers.cloudflare.com/workers-ai/.

[6] Nicole Forsgren, Jez Humble, and Gene Kim. 2018. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution Press.

[7] Xueyu Hu et al. 2024. InfiAgent-DABench: Evaluating Agents on Data Analysis Tasks. *arXiv preprint arXiv:2401.05507* (2024).

[8] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv preprint arXiv:2310.06770* (2024).

[9] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Hruschka, Ankur Sharma, Tushar T Joshi, Nikhil Mober, et al. 2023. DSPy: Compiling Declarative Language Model Calls into State-of-the-Art Pipelines. *arXiv preprint arXiv:2310.03714* (2023).

[10] Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. 2024. VisualWebArena: Evaluating Multimodal Agents on Realistic Visual Web Tasks. *arXiv preprint arXiv:2401.13649* (2024).

[11] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. *arXiv preprint arXiv:2211.11501* (2022).

[12] Fangyu Lei, Tianbao Xie, et al. 2024. Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows. *arXiv preprint arXiv:2411.07763* (2024).

[13] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQL. *Advances in Neural Information Processing Systems* 36 (2023).

[14] Jinyang Li, Nan Huo, Yan Gao, Jiayi Shi, Yingxiu Zhao, Ge Qu, Yurong Wu, Chenhao Ma, Jian-Guang Lou, and Reynold Cheng. 2024. Tapilot-Crossing: Benchmarking and Evolving LLMs Towards Interactive Data Analysis Agents. *arXiv preprint arXiv:2403.05307* (2024).

[15] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. 2023. AgentBench: Evaluating LLMs as Agents. *arXiv preprint arXiv:2308.03688* (2023).

[16] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raez, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. 2023. GAIA: A Benchmark for General AI Assistants. *arXiv preprint arXiv:2311.12983* (2023).

[17] Gaurav Sahu et al. 2024. InsightBench: Evaluating Business Analytics Agents Through Multi-Step Insight Generation. *arXiv preprint arXiv:2407.06423* (2024).

[18] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Liber, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. *arXiv preprint arXiv:2405.15793* (2024).

[19] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. *arXiv preprint arXiv:1809.08887* (2018).

[20] Zhaojian Yu et al. 2024. HumanEval Pro and MBPP Pro: Evaluating Large Language Models on Self-invoking Code Generation. *arXiv preprint arXiv:2412.21199* (2024).

[21] Qi Zhang, Xinwei Liu, Honglin Chen, and Tong Guo. 2024. GEPA: Gradient-based Evolution Prompt Optimization with LLM Agents. *arXiv preprint arXiv:2402.00421* (2024).

[22] Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. 2024. WebArena: A Realistic Web Environment for Building Autonomous Agents. *arXiv preprint arXiv:2307.13854* (2024).