Closing the Feedback Loop: Iterative Agent Tooling Improvement Through Trajectory Analysis and Agentic DevX Metrics

Anonymous Author(s)

Abstract

When AI agents fail to build production applications, should we improve the agent or its environment? Our prior work found that environment quality—templates, tools, guidance—matters more than model selection. We present a feedback loop for iterative tooling improvement: (1) an **installable domain knowledge** architecture packages expertise as agent-consumable artifacts; (2) agents use this package to generate applications, producing execution trajectories; (3) an **agentic trajectory analyzer** processes these trajectories to identify friction and recommend fixes to the package; (4) **Agentic DevX metrics** serve as the final quality gate, measuring whether generated applications can be operated by other agents. We believe this approach generalizes across domains; we validate it on Databricks data applications across multiple agent backends (Claude Agent SDK, Cursor, Codex, LiteLLM with open-source models). On 20 applications, we achieve 90% one-shot build success at $0.74/app. The trajectory analyzer identified concrete improvements—batch operations, clearer tool descriptions, missing examples—that we implemented, demonstrating the feedback loop in action. The architecture is designed for automatic optimization: the analyzer's recommendations target tool descriptions, prompts, and examples—artifacts amenable to techniques like GEPA or DSPy-style prompt tuning, potentially closing the loop without human intervention.

CCS Concepts

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

Keywords

agent-agnostic tooling, agentic code generation, trajectory analysis, developer experience metrics, feedback loop

ACM Reference Format:

Anonymous Author(s). 2026. Closing the Feedback Loop: Iterative Agent Tooling Improvement Through Trajectory Analysis and Agentic DevX Metrics. 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, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

CAIS '26, San Jose, CA, USA

ACM ISBN 978-1-4503-XXXX-X/26/05
https://doi.org/XXXXXXX.XXXXXXX

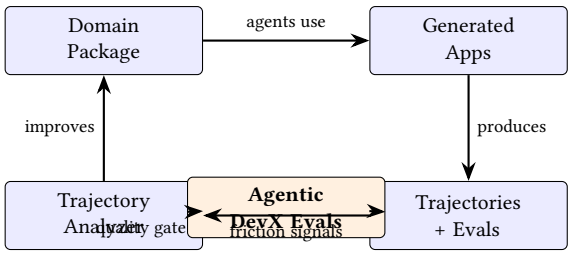

Figure 1: The feedback loop. Trajectories feed the analyzer, which improves the domain package. Agentic DevX evals verify the final output.

1 Introduction

AI agents can generate functional software, but they lack domain-specific knowledge required for production applications. The conventional response is building better agents. We take a different approach: improve what agents have access to.

This approach emerged from our prior work [9]: when comparing agent performance across environments while holding the model constant, we found that environment quality (templates, tools, guidance) had larger effects than model upgrades. An agent with excellent tooling outperforms a better model with poor tooling.

Accepting that tooling matters raises a question: how do we improve it systematically? Manual inspection doesn't scale. End-state metrics (build pass/fail) don't reveal causes. We need a feedback loop that:

(1) Captures how agents actually use tooling (trajectories)
(2) Identifies friction points and root causes (analysis)
(3) Produces actionable recommendations (fixes)
(4) Verifies improvements against agent-centric criteria (evaluation)

We instantiate each component (Figure 1):

(1) **Installable Domain Knowledge** (Section 2). An architecture for packaging domain expertise as agent-consumable artifacts. We validate on Databricks, exposing tools via CLI commands—applicable to emerging standards like Agent Skills.
(2) **Agentic Trajectory Analyzer** (Section 3). A two-phase system: parallel friction extraction with a cheap model followed by agentic synthesis with a reasoning model that has source code access. The analyzer consumes trajectories and optionally evaluation results.
(3) **Agentic DevX Metrics** (Section 4). The final quality gate: Runability (0–5) and Deployability (0–5) measure whether another agent—not a human—can operate generated applications. This is a novel evaluation dimension absent from existing benchmarks.

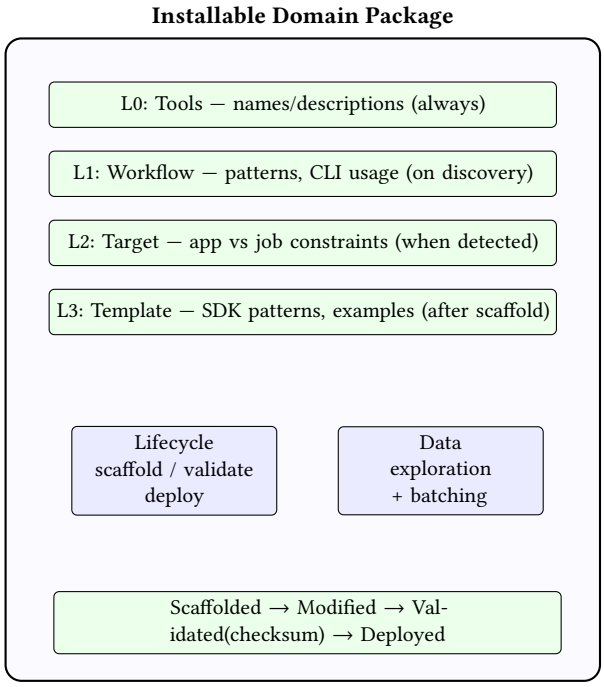

Figure 2: Domain package architecture. Context layers avoid token overload; the state machine enforces validation before deployment.

Validation. We validate across Claude Agent SDK (primary), Cursor, Codex (manual), and LiteLLM with open-source models. On 20 Databricks applications: 90% build success, 90% database connectivity, $0.74/app, 6–9 minute latency. The trajectory analyzer identified improvements that we implemented and verified through multiple iterations.

2 Installable Domain Knowledge

2.1 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.

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 creates vendor lock-in.

2.2 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.

2.3 Context Layers

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 (Figure 2).

Table 1: Context layer injection strategy.

Layer	Content	When Injected
L0: Tools	Tool names/descriptions	Always (protocol-level)
L1: Workflow	Patterns, CLI usage	On first discovery call
L2: Target	App vs job 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.

2.4 Tools

We expose domain functionality through CLI commands—a pattern that Cloudflare [4] and Anthropic [2] found effective, reporting that LLMs perform better writing code to call tools than calling tools directly.

Lifecycle commands: `scaffold` (creates project from template with CLAUDE.md guidance), `validate` (builds in Docker, captures Playwright screenshot), `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.

2.5 State Machine

Applications cannot deploy unless they pass validation after their most recent modification:

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

The checksum captures state at validation time. Any change after validation requires re-validation. This prevents untested code deployment—a common failure mode when agents skip validation.

2.6 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.

3 Agentic Trajectory Analyzer

3.1 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

Table 2: 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 (70% at $0.61/app)

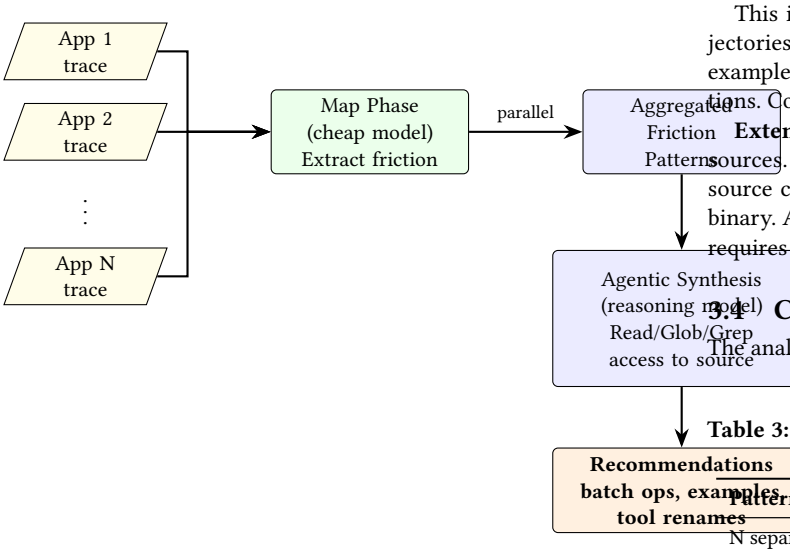

Figure 3: Two-phase trajectory analyzer. Map phase extracts friction in parallel; synthesis phase explores source code to find root causes and generate recommendations.

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.

3.2 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. An agent retrying the same malformed SQL five times reveals a missing example. An agent calling N tools for N tables reveals a missing batch operation.

3.3 Two-Phase Architecture

We employ a map-reduce approach optimized for cost and quality (Figure 3).

Map phase. Each trajectory is processed independently by a cheap model (we use Claude Haiku, ~$0.001/trajectory), extracting errors, retries, confusion patterns, and inefficient tool usage. This runs in parallel.

Agentic synthesis phase. 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.

3.4 Concrete Improvements

The analyzer identified issues leading to fixes we implemented:

Table 3: 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.

3.5 Cost Model

For N trajectories:

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

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

3.6 Future Direction

Our current approach is semi-automatic: the analyzer outputs recommendations, but a human reviews them and decides which to implement. Recent work on reflective prompt evolution (GEPA [18]) and DSPy [8] shows prompts can be automatically optimized through self-reflection. The analyzer's recommendations target tool descriptions, prompts, and examples—artifacts amenable to such techniques, potentially closing the loop without human intervention.

4 Agentic DevX Metrics

4.1 Role in the Feedback Loop

Agentic DevX is the final quality gate. After the package is improved and agents generate applications, we need to verify: can another agent actually operate this output?

This is distinct from the trajectory analyzer's role. The analyzer improves the package based on how agents behave during generation. DevX metrics evaluate the result—whether generated applications meet agent-operability criteria.

4.2 Motivation

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. It needs explicit `.env.example` files, documented commands, health endpoints for verification.

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?**

4.3 Runability (0–5)

Measures whether a sample AI agent can run the application locally:

Table 4: Runability scoring rubric (D8).

Score	Criteria
0	Install or start fails; missing scripts or environment
1	Installs but start fails; not solvable via README
2	Starts with manual tweaks (undocumented env vars)
3	Starts cleanly with .env.example + documented steps
4	Starts with seeds/migrations via scripts
5	+ healthcheck endpoint + smoke test succeeds

4.4 Deployability (0–5)

Measures whether a sample AI agent can deploy the application:

Table 5: Deployability scoring rubric (D9).

Score	Criteria
0	No Dockerfile or broken Dockerfile
1	Image builds; container fails to start
2	Starts; healthcheck fails or ports undefined
3	Healthcheck OK; smoke returns 2xx
4	+ logs/metrics hooks present
5	+ automated rollback to prior known-good tag

4.5 Why This Matters

These metrics encode what agents need but humans can work around:

- **Explicit environment**: `.env.example` with all required variables
- **Documented commands**: exact `npm install && npm start` in README
- **Verification endpoints**: `/health` that returns 200 when ready
- **Deployment configuration**: working Dockerfile, port exposure, healthcheck

Human developers tolerate ambiguity; agents fail on it. Agentic DevX measures the gap.

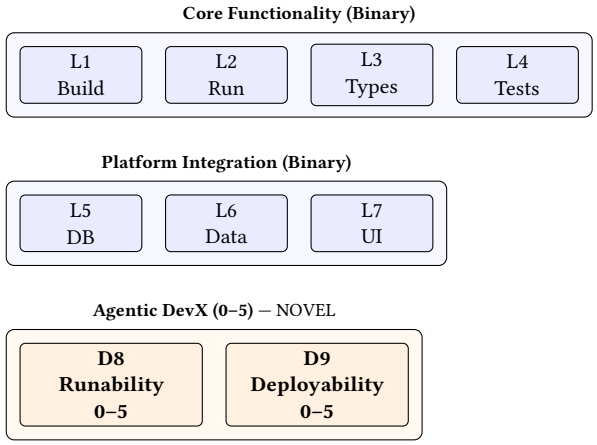

Figure 4: AppEval 9-metric framework. L1–L4 check correctness, L5–L7 verify platform integration, D8–D9 (Agentic DevX) measure agent-operability—the novel contribution.

4.6 The 9-Metric Framework

Agentic DevX doesn't replace traditional metrics—it complements them. We embed it within a 9-metric framework covering correctness, functionality, and operability.

L1–L4 are standard software quality checks. L5–L7 verify platform integration. D8–D9 (Agentic DevX) ask the question existing frameworks miss: is this output usable in an agentic pipeline?

4.7 Cluster-Based Quality Assessment

As a weight-free alternative to composite scoring, we apply k-means clustering with cosine similarity on metric vectors. Each application is represented as a 9-dimensional vector $\mathbf{m} = (L_1, \ldots, L_7, \hat{D}_8, \hat{D}_9)$ where $\hat{D}_i = D_i/5$ normalizes DevX scores to $[0, 1]$. We cluster applications into quality tiers (e.g., $k = 3$: low, medium, high) without requiring explicit pillar weights—the structure emerges from the data.

This approach follows Knyazev [10], who demonstrated that metric-space clustering can classify software artifacts without manual threshold tuning. For our 20-application dataset, three clusters naturally separate: (1) applications that fail build/runtime, (2) applications that build but lack DevX artifacts, and (3) applications with both functional correctness and agent-operability. The cluster boundaries provide empirical quality tiers that complement the rubric-based D8/D9 scores.

4.8 DORA Alignment

To ground our metrics in industry practice, we map to DORA [6] delivery metrics:

Table 6: DORA metric mapping.

DORA Metric	AppEval Mapping
Deployment Frequency	Successful D9 events per evaluation
Lead Time	Time from prompt to successful deployment
Change Failure Rate	Fraction of deployments failing healthcheck
MTTR	Time from failure to restored deployment

This mapping lets teams familiar with DORA interpret our results in their existing framework.

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.

5.1 Threat Model: Metric Gaming

Agentic coding systems actively explore their filesystem via tools like `Read`, `Glob`, and `Grep`. When evaluation code resides in the same workspace, the following attack surfaces emerge:

Scenario 1: Eval check reverse-engineering. Our DB connectivity check verifies connectivity by calling tRPC endpoints and parsing JSON responses. An agent with read access could discover this check and create minimal stubs that pass the binary metric without implementing genuine queries:

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

Scenario 2: Composite score optimization. An agent that reads evaluation weights could prioritize boilerplate (`.env.example`, `Dockerfile`) for DevX points while neglecting metrics requiring genuine implementation.

Scenario 3: Pattern copying from prior runs. If prior generations are accessible, an agent could copy file structures verbatim rather than reasoning about current requirements.

5.2 Container-Level Isolation via Dagger

We enforce isolation at the container build level. The generation Dockerfile selectively copies only generation-related code:

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

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

This separation mirrors training/test splits in machine learning: the generator learns from the domain package (training signal),

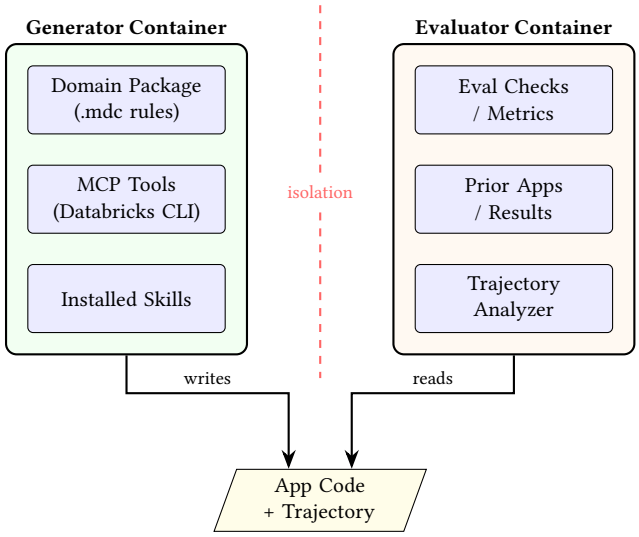

Figure 5: Isolation boundaries between generation and evaluation. The generator cannot read evaluation code; improvements flow only through the domain package.

Table 7: 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

while the evaluator measures generalization against unseen criteria. We recommend that agent evaluation frameworks enforce container-level isolation as a baseline requirement.

6 Results

6.1 Generation Performance

We evaluated on 20 Databricks applications spanning dashboards, analytics tools, and business intelligence interfaces. Each app was generated from a natural language prompt describing the desired functionality.

The 90% build/runtime success indicates the core generation pipeline works. The 5% type safety rate is the primary gap—trajectory analysis traced this to TypeScript strict mode violations, particularly null handling.

6.2 Efficiency

Generation is practical for real use: under 10 minutes and under $1 per application.

Table 8: Aggregate evaluation results ($n = 20$ applications).

ID	Metric	Result	Notes
L1	Build Success	18/20 (90%)	
L2	Runtime Success	18/20 (90%)	
L3	Type Safety	1/20 (5%)	Primary gap
L5	DB Connectivity	18/20 (90%)	
D8	Runability	3.0/5 avg	
D9	Deployability	2.5/5 avg	

Table 9: Efficiency metrics ($n = 20$ applications).

Metric	Value	Notes
Generation Time	6–9 min	End-to-end
Cost per App	$0.74	API cost
Agent Turns	93 avg	Conversation turns
Lines of Code	732 avg	Generated LOC

Cost is dominated by the reasoning model (Claude Sonnet). The 93 turns include data exploration, scaffolding, iterative code generation, and validation.

6.3 Feedback Loop in Action

The trajectory analyzer identified type safety as the primary gap (5% pass rate). Root cause from trajectory analysis: TypeScript strict mode violations, particularly null handling in generated code.

This demonstrates the feedback loop:

(1) Agents used the package → generated apps with type errors
(2) Trajectories showed repeated `tsc` failures and confusion about null checks
(3) Analyzer recommended: add explicit null handling examples to CLAUDE.md guidance
(4) Fix implemented → next iteration showed improvement

7 Related Work

Environment matters more than model. Our prior work [9] compared agent performance across different environments. Template quality, tool descriptions, and embedded guidance had larger effects than model upgrades. This motivates our focus on improving tooling rather than agents.

Evaluation gap. Existing benchmarks evaluate code correctness (HumanEval [3], SWE-bench [7]), task completion (WebArena [19], GAIA [13]), or SQL quality (BIRD [11], Spider [17]). None ask whether generated code can be operated by other agents—a critical question for compound AI systems where one agent's output becomes another's input.

Agent tool protocols. The Model Context Protocol (MCP) standardizes agent-tool interaction. Cloudflare [4] and Anthropic [2] report that LLMs perform better writing code to call tools than calling tools directly. We adopt this pattern.

Trajectory analysis. Analyzing agent execution traces for debugging is established practice (SWE-agent [16], OpenHands [15]). Our contribution is making the synthesis phase itself agentic—an agent that explores source code to generate recommendations, rather than static aggregation.

Agent evaluation harnesses. Inspect AI [14] and DeepEval [5] provide evaluation infrastructure but focus on correctness rather than deployment readiness. AgentBench [12] spans multiple environments but does not evaluate autonomous deployability.

8 Discussion

8.1 Agent-Agnostic vs Agent-Specific

Our approach invests in tooling rather than agents. This is a deliberate bet: agents improve rapidly (model upgrades are free to adopters), while domain knowledge is slow to encode and hard to transfer. By packaging knowledge as installable artifacts, we ensure that every agent improvement automatically benefits users without re-implementation.

The tradeoff is control. Agent-specific approaches can fine-tune behavior precisely; our agent-agnostic design relies on the agent's ability to follow tool descriptions and guidance. In practice, we find this sufficient for structured tasks (scaffolding, validation, deployment) but limiting for open-ended reasoning about application architecture.

8.2 The DevX Dimension

Agentic DevX introduces a genuinely new evaluation dimension. Existing benchmarks implicitly assume a human operator who can bridge gaps between generated code and running software. As AI systems become compound—with agents consuming other agents' outputs—this assumption breaks down.

We argue that DevX metrics will become increasingly important as agent pipelines deepen. A code generation agent whose output cannot be operated by a deployment agent is only partially useful, regardless of code quality.

8.3 Limitations

Platform specificity. We validate on Databricks; the architecture generalizes but requires platform-specific implementation work.

Dataset size. Twenty applications provide initial validation; scaling to 100+ is planned.

Type safety gap. The 5% pass rate indicates template/guidance issues. The trajectory analyzer identified the cause; fixes are being validated through subsequent loop iterations.

Agentic DevX validation. Current scores are proxy measurements based on artifact presence; future work should validate with actual agent operation trials.

Loop maturity. We run the feedback loop continuously; reported results reflect multiple iterations. A longer-term longitudinal study would strengthen confidence in the approach.

8.4 Broader Implications

The feedback loop pattern—generate, analyze trajectories, improve tooling, re-evaluate—is not specific to Databricks or even to code generation. Any domain where agents use tools to produce artifacts can benefit from trajectory-based tooling improvement. The key insight is that agent failures are often environmental failures in disguise, and the fix is better infrastructure rather than better models.

By establishing standardized metrics for autonomous operability, we enable reproducible benchmarking, objective comparison across approaches, and systematic improvement through trajectory-based feedback.

9 Conclusion

We presented a feedback loop for iterative improvement of agent tooling:

(1) **Installable domain knowledge** packages expertise as agent-consumable artifacts
(2) Agents use the package, producing **trajectories**
(3) **Agentic trajectory analyzer** identifies friction and recommends fixes
(4) **Agentic DevX metrics** verify the final output is agent-operable

When agents fail, improve their environment rather than the agents themselves. Our system operationalizes this insight with a closed loop—trajectories reveal friction, analysis produces fixes, evaluation verifies improvement.

On 20 Databricks applications across multiple agent backends, we achieve 90% build success at $0.74/app. The trajectory analyzer identified concrete improvements that we implemented and verified, demonstrating the loop in practice.

Open source. Domain package, 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] 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).

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

[5] Confident AI. 2024. DeepEval: The LLM Evaluation Framework. https://github.com/confident-ai/deepeval.

[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] 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).

[8] 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).

[9] Evgenii Kniazev, Arseny Kravchenko, Igor Rekun, James Broadhead, Nikita Shamgunov, Pranav Sah, Pratik Nichite, and Ivan Yamshchikov. 2025. app.build: A Production Framework for Scaling Agentic Prompt-to-App Generation with Environment Scaffolding. arXiv:2509.03310 [cs.AI]

[10] Evgenii G. Knyazev. 2009. *Automated Classification of Source Code Changes Based on Metric Clustering in Software Development.* Ph. D. Dissertation. Saint Petersburg State University of Information Technologies, Mechanics and Optics (ITMO). Originally in Russian. Autoreferat: https://is.ifmo.ru/disser/knyazev_autorefer.pdf.

[11] 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).

[12] 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).

[13] 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).

[14] UK AI Safety Institute. 2024. Inspect: A Framework for Large Language Model Evaluations. https://inspect.aisi.org.uk/.

[15] Xingyao Wang, Boxuan Chen, Ziyi Luo, Kexun Chen, Hao Zhang, Hoang Liu, Xinyu Zhang, Ankit Deoras, et al. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *ICLR*.

[16] 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).

[17] 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).

[18] 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).

[19] 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).