Tool-Making and Self-Evolving LLM Agents in Low-Latency Systems

Kalle Kujanpää Ning Liu Shahnawaz Alam Yeshwanth Reddy Sura
Tianyu Yang Kristina Klinkner Shervin Malmasi

Amazon, Fulfillment Technologies & Robotics
{kujanpaa,malmasi}@amazon.com

Abstract

Production LLM agents often waste latency and reliability by regenerating code for the same procedural steps on every request. We replace this inference-time coding loop with an agentic tool-making pipeline that compiles repeated SOP steps into validated, versioned tools before deployment. The tool-maker grounds synthesis in the live environment as it collects execution traces, observes backend schemas and values, generates candidate tools, and repairs them against labeled cases. At runtime, the production agent calls these tools directly and falls back to code generation only when needed. We deploy the approach in a Fulfillment Center alarm-triage system, where an agent diagnoses alarms against a 44-node SOP over heterogeneous metric backends. In production, tool calls reduce p50 latency by 42%. On 1,500 historical alarms, they reduce end-to-end error rate by up to 53% by suppressing run-to-run variance in repeated steps. Because tools return compact structured verdicts, they also enable a simpler direct-call architecture, reducing p50 latency by a further 62% in a controlled ablation. Versioned tools also improve auditability and expose specification gaps and upstream data drift. Our results show that agents that build and maintain their own tool libraries, a key element of self-evolving agents, can make industrial LLM systems faster, more reliable, and easier to operate.

1 Introduction

LLM coding agents are becoming a practical interface to complex software and operational systems. Instead of a fixed API or hand-written automation per task, these agents read natural-language instructions, write and execute code, and iterate until they produce an answer or action. This flexibility has enabled deployments in on-call incident triage (Chen et al., 2024b; Wang et al., 2024b), workflow-guided operations (Ye et al., 2025), and general code-based tool use (Wang et al., 2024a). In industry settings, many procedures are documented as standard operating procedures (SOPs), while the data needed to execute them resides in metric stores, logs, dashboards, and ticketing systems.

This flexibility, however, creates an efficiency and reliability bottleneck. In the prevailing CodeAct-style paradigm (Wang et al., 2024a), the agent generates and executes fresh code for each request at inference time. When the same workflow is repeated against a stable backend, the agent re-interprets the same instruction, rediscovers the same schema, and regenerates similar code, raising latency, cost, and run-to-run variance. Prior benchmarks show agents still struggle with workflow-guided tasks (Nandi et al., 2025; Wang et al., 2025b; Riddell et al., 2026), and even SOP-enhanced multi-agent systems leave a substantial accuracy gap on operational diagnostics (Pei et al., 2025). In our setting, most latency and correctness errors are caused by translating underspecified SOP text into a concrete query against a production metric backend.

This has motivated a shift from inference-time coding agents to self-evolving agents: systems that improve their own action space over time by constructing reusable tools, skills, or programs before they are needed in production. Recent work explores task-conditioned tool synthesis (Cai et al., 2024; Yuan et al., 2024; Wang et al., 2024c), closed-loop construction driven by feedback (Wang et al., 2023; Wölflein et al., 2025), reusable workflow induction from demonstrations (Wang et al., 2024d), and the translation of operational documents such as company policies and troubleshooting guides into tools (Zwerdling et al., 2025; Mao et al., 2026). The key idea is to amortize cost by identifying procedural steps executed repeatedly and compiling them into deterministic tools, which are then invoked cheaply and consistently at runtime.

We study this approach in a production agent at a Fulfillment Center (FC), where SOPs prescribe how alarms on robotic induct, conveyance,

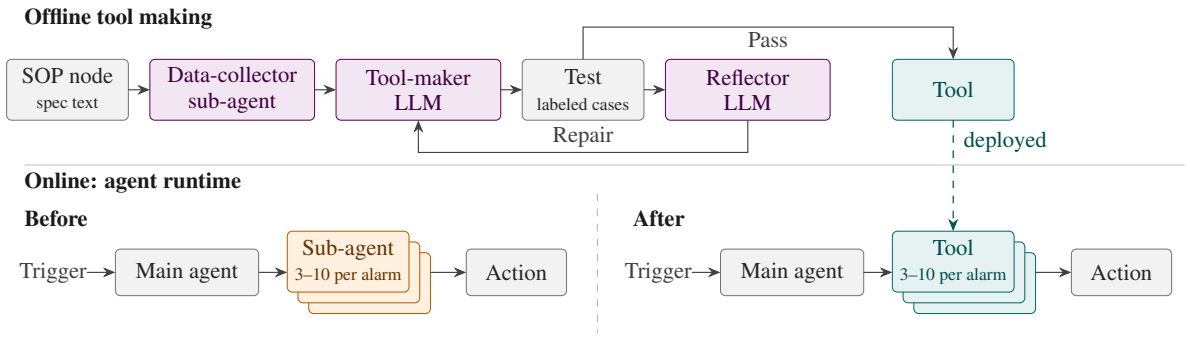


Figure 1: Offline (top), each SOP node is compiled into a tool: a data-collector sub-agent runs against MCP, producing a trace; a tool-maker LLM writes a candidate from it; and a reflector–tool-maker loop repairs the candidate against labeled cases. Online (bottom), the baseline sub-agent writes fresh MCP-querying code per node over 10–20 LLM turns, whereas our agent calls the compiled tools inline, one call per node.

and sortation subsystems are diagnosed and acted upon. These SOPs branch on thresholds from heterogeneous metric backends and select interventions from low-cost flow reallocations to high-cost staffing changes, ticket escalations, and equipment standbys. Latency and correctness are tightly coupled: failures cascade across subsystems within minutes, while an expensive action taken against the wrong cause is difficult to reverse. Manual execution by floor operators is slow under concurrent alarms and error-prone given the number of metrics, schemas, and conditional branches.

We report the design and production deployment of an agentic tool-making pipeline for this setting, triaging outbound dock alarms against a written SOP. The system has two modes (Figure 1). At build time, a tool-maker constructs a library of per-SOP-step tools against the production environment, observing schemas and execution traces and validating candidates against labeled cases rather than relying on SOP text alone. At inference time, the agent invokes these tools per alarm, falling back to code generation only when a tool is unavailable or fails. This preserves the flexibility of coding agents while moving repeated, latency-sensitive work into a reusable and auditable tool library.

We make two main contributions: an agentic tool-making pipeline that compiles repeated SOP steps into validated tools, and its integration into a deployed alarm-triage agent, where it cuts p50 latency by 42%, enables an architectural redesign that cuts it a further 62% in a controlled ablation, and reduces the end-to-end error rate by up to 53%. We also report three findings. The data-collection trace and test-repair loop are both necessary. Residual errors then concentrate on underspecified SOP steps, where targeted fixes raise pass@1 from 94.5% to 99.9%. Versioned tools with stable inputs and outputs reveal pre-existing upstream inconsistencies that run-to-run variation otherwise hides. Our results suggest that agents that compile their own experience into reusable tools are a practical path for bringing coding-agent flexibility to latency-sensitive industrial workflows.

2 Production Setting and Task

An outbound dock is the final operation in a fulfillment center, where packages are sorted and loaded before they leave for downstream logistics. The dock is tightly coupled: a degraded subsystem backs up the operations feeding it and cascades across the dock within minutes, amplifying a localized fault into an outsized delay that threatens delivery commitments. The agent triages alarms such as robotic stations idling due to upstream conveyance blockages, or producing elevated error rates from camera, sensor, label, or packaging defects.

The agent follows an SOP encoded as a decision tree of 44 decision nodes and 19 action nodes. Decision nodes are of three kinds. Observations test whether a metric crosses an alert threshold and serve as alarm entry points, root causes identify the underlying failure, and constraints gate the most expensive actions. Actions range from cheap (process alerts, flow reallocation) to costly (equipment shutdown, staffing changes). Each node specifies which data to query and how to process it.

The agent evaluates observations to localize the failure, reaches candidate root causes, and executes the action. Before shutting down a robot with a faulty suction cup, for example, constraint nodes verify that the shutdown is either safe (remaining capacity covers the item flow) or unavoidable (the error rate makes operation worse).

3 Method

We build on the hierarchical multi-agent architecture of Eluna (Liu et al., 2026b) where a main agent traverses the decision tree and delegates each evaluated node to a CodeAct-style sub-agent (Wang et al., 2024a) via a free-text question. The sub-agent generates and executes code that queries metrics through MCP (Model Context Protocol), iterating until it returns a free-text answer, which the main agent parses into a verdict (true/false/no data) to inform subsequent traversal. Delegation keeps the multi-turn data-processing context within the sub-agent, and sub-agent calls within a decision-tree layer run in parallel. Per alarm, the agent evaluates 3–10 nodes, and the sub-agent loop can exceed 100 LLM turns in the most complex cases.

The dominant per-alarm cost in this architecture is the sub-agent's loop, as most of its turns translate the same SOP text into the same MCP queries against an unchanging interface. We replace this loop with per-node tools, each a build-time compilation of the loop for one node into a single deterministic call that returns the verdict directly at inference. Tools are model-agnostic and regenerable independently of agent fine-tuning.

3.1 Tool-Making Pipeline

SOPs are authored by humans, and the SOP text omits execution details that a human reader would resolve from domain knowledge or by inspecting the data, such as field names, datatypes, null handling, and boundary-case conventions. These omissions cause persistent failures when a tool-maker is given only the SOP text. The pipeline closes the gap with two grounding mechanisms (Figure 1, top): a *data-collection trace* of real execution against the live schema, and a *test–repair loop* that corrects each candidate against labeled cases. Each tool is a function with a uniform signature mapping a (warehouse, timestamp, context) input to the node's verdict (true, false, or no data) and supporting detail. The context carries site-specific parameters and verdicts from parent nodes. Each node has a labeled training set, consisting of (input, verdict) pairs from an offline labeler whose logic was specified by subject-matter experts (Appendix B discusses the labeling).

The pipeline has three model components, shown left-to-right in Figure 1. The *data-collector sub-agent* is the baseline CodeAct sub-agent serving as the inference-time fallback, with access to the production MCP. For each node, it is run on three sampled training cases balanced across verdict labels, writing and executing query code and iterating to a final verdict that is graded against the case label. The resulting trace, passed to the tool-maker, contains the query code, the MCP responses and observed schema (field names, datatypes, value ranges), and the graded verdict, supplying the execution details the SOP omits.

The *tool-maker LLM* produces a candidate tool conditioned on the SOP node text, the node's position in the decision tree (its parent and child nodes), and the data-collector's traces. The candidate is then tested against the full labeled training set, and disagreements are passed to the test–repair loop. The *reflector LLM* writes a short diagnosis of the failures, after which the tool-maker does a rewrite conditioned on the diagnosis and the disagreements. The reflector receives the failing (input, expected, predicted) triples and the previous candidate, and the tool-maker receives the same and the reflector's diagnosis. The loop runs for a budget of three rounds and terminates early once a candidate passes the full training set. The candidate with the highest training-set pass rate is deployed. Appendix E shows an example from a generated tool.

At inference, tools are used in one of two configurations: the main agent delegates a node to a sub-agent that calls the node's tool, or the main agent calls the tool directly (Figure 1, bottom). Each tool returns the verdict, the observed value, the threshold, and a textual explanation. "No data" is reserved for missing metrics, and exceptions trigger fallback to the CodeAct sub-agent of the baseline. Only the affected node reverts to baseline latency and accuracy. The remaining nodes continue using their tools.

3.2 Agent Fine-Tuning

Production latency constraints cap the deployed alarm-triage agent at a small model size, and small off-the-shelf models fail to follow the SOP reliably. We therefore distill a larger teacher into the small student model: the teacher generates trajectories on the alarm-triage task, rejection sampling retains only those whose extracted observations, root causes, and actions all match ground truth (Yuan et al., 2023), and the student is fine-tuned on them with LoRA (Hu et al., 2022). To keep the student robust when tools are missing or fail, the teacher generates trajectories under varied tool availability: in 75% of trajectories, the generated

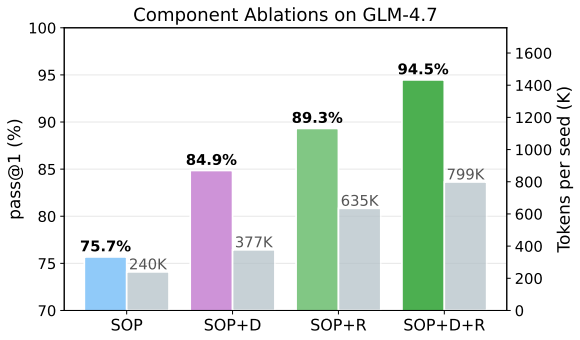


Figure 2: Per-node pass@1 (left) and generation cost in tokens (right) for each ablation configuration on GLM-4.7. +D adds the data-collection trace (Section 3.1); +R adds the test-repair loop. Only the deployed configuration (SOP+D+R) exceeds 90% pass@1.

Tool Maker	SOP	+D	+R	+D+R
GLM-4.7	75.7	84.9	89.3	**94.5**
GLM-5	76.8	83.4	88.0	93.6
Qwen3 235B	74.2	71.6	79.9	81.1
GLM-4.7 Flash	56.4	60.4	58.7	65.8

Table 1: Per-node pass@1 (%) by tool-maker and configuration. +D adds the data-collection trace; +R adds the test-repair loop. The deployed SOP+D+R column is in bold, with GLM-4.7 deployed. Confidence intervals and significance tests are in Table 9.

tools are available but 25% of tools are randomly disabled per trajectory, and in the remaining 25% all node tools are removed so the teacher falls back to the sub-agent for every node.

4 Tool-Making Experiments

4.1 Experimental Setup

Data. The pipeline is evaluated on 44 decision nodes. Each node has a labeled training set used by the pipeline and a separate held-out evaluation set. Per-node case counts range from 100 to 200, totaling 8,011 training cases and 8,022 evaluation cases. Data labels are evenly distributed.

Models. We compare the following four models as the tool-making agent (Section 3.1): GLM-4.7 (Zeng et al., 2025), GLM-5 (Zeng et al., 2026), Qwen3 235B (Yang et al., 2025), and GLM-4.7 Flash (Zeng et al., 2025). We use Claude Opus 4.6 for additional ablations (Anthropic, 2026) to prevent model capacity from confounding the comparison. When reflection is used, the reflector LLM is the same as the tool-maker LLM. The deployed alarm-triage agent uses a separately fine-tuned model (Section 3.2).

Evaluation. We evaluate the build-time pipeline by per-node pass@1: the probability that one full execution of the pipeline produces a tool that passes the held-out evaluation set for that node, where a tool passes only if it returns the correct verdict on every case in the set. Pass@1 is averaged across the 44 decision nodes. Confidence intervals and significance tests are reported in Appendix G.

4.2 Main Results

Figure 2 shows the per-node pass@1 and generation cost of each configuration on GLM-4.7, and Table 1 reports pass@1 for all four tool-makers. The full configuration adds the data-collection trace (+D) and the test-repair loop (+R) to the SOP text, reaching 94.5% pass@1 on GLM-4.7 and 93.6% on GLM-5. Both components are necessary: removing the repair loop reduces pass@1 by 9.6pp and removing the trace by 5.2pp, and only the full SOP+D+R configuration exceeds 90%. The two strong tool-makers, GLM-4.7 and GLM-5, behave alike, with the trace and the repair loop each adding a significant gain, whereas the weaker Qwen3 235B and GLM-4.7 Flash gain less and do not improve monotonically. We deploy GLM-4.7, which attains the highest pass@1 of the four. The appendices report further results (Appendix C). Repair compute cannot substitute for the trace, whose components each contribute a gain, and reflection-based repair outperforms repairing from raw failures only when the trace is present. The pipeline also approaches the full-data ceiling from a handful of labeled cases per node, with a label-free variant within a few points (Appendix D).

Across the 30 GLM-4.7 pipeline runs, each generated library contains 39–43 tools that pass their full held-out sets (median and mode: 42 of 44), and 38 of the 44 nodes succeed in all 30 runs. Failures are concentrated in three nodes, which succeed in 19, 3, and 0 of the 30 runs, respectively. When they fail, each misses only one to three fixed edge cases out of roughly 200. Thus, although these tools fail the strict pass@1 criterion, they remain correct on the overwhelming majority of their cases, which allows us to deploy them. The deployed library was produced by a single pipeline run under this configuration and contains 42 of 44 tools that pass all held-out cases.

4.3 Specification Quality as the Bottleneck

At this 94.5% ceiling, four of the 44 nodes account for almost all remaining errors. Three have edge

Configuration	pass@1
Original SOP, original data	94.5
Original SOP, augmented data	96.6
Clarified SOP, augmented data	**99.9**

Table 2: Effect of targeted specification and data improvements on the four residual nodes (Opus). Augmentation adds labeled cases for the edge-case nodes.

cases (negative denominators, null MCP values) that neither the SOP nor the training labels cover, so no failing case reaches the repair loop. The fourth has an ambiguous averaging window that the repair loop overfits away from the SOP (Appendix E). Two undeployed Opus runs confirm the cause: augmenting the training data raises pass@1 to 96.6%, and clarifying the ambiguous SOP raises it to 99.9% (Table 2). The headline 94.5% pass@1 in Section 4.2 is unaffected.

5 Agent Experiments and Deployment

The deployed alarm-triage agent uses small language models to meet production latency targets and calls the GLM-4.7-generated tool library from Section 4. We report runs with two such models (Qwen3 32B (Yang et al., 2025) and GLM-4.5-Air (Zeng et al., 2025)), while the tool-making pipeline (Section 3.1) runs offline with stronger models, paying their reasoning cost once per tool and amortizing it across every production call. We evaluate end-to-end correctness on 1,500 historical alarm scenarios with ground-truth observations, root causes, and actions, and latency from production. The evaluation alarms are separated from the fine-tuning trajectories (Section 3.2) by a temporal split and are disjoint from the cases used for tool generation. Non-deployed configurations are evaluated by offline ablation. Before a candidate agent is promoted to production, it runs in shadow: it executes on live alarms in parallel with the production agent without affecting outcomes, and discrepancies between the two are reviewed by humans. Since deployment, our tools have been invoked across over 5,000 production alarms.

5.1 Latency

Table 3 reports production p50 and p99 latency across three configurations. Replacing the sub-agent's CodeAct loop with tool calls cuts per-alarm production p50 by 42% on Qwen3 32B under the original two-tier sub-agent architecture (rows 1–2). The same comparison reduces per-alarm output tokens by 58% (14,998 to 6,225) and sub-agent turns by 45% (33.7 to 18.7). A per-component breakdown (Appendix F) shows the reduction is dominated by an 80% fall in sub-agent code-generation tokens, with main-agent tokens also falling.

Model	Configuration	p50	p99
Qwen3 32B	Sub-agent writes code	100	100
Qwen3 32B	Sub-agent calls tools	58	59
GLM-4.5-Air	Main-agent direct calls	26	29

Table 3: Production p50/p99 latency, relative to the no-tools sub-agent baseline (set to 100, lower is better). Rows 1 and 2 differ only in whether the agent uses tools. Row 3 changes both the architecture (direct calls) and the model (GLM-4.5-Air). A controlled ablation isolates the architecture-only effect, a 62% p50 reduction (Section 5.1).

	No tools	Sub-agent	Main-agent
Qwen3 32B	97.2	98.2^{**}	—
GLM-4.5-Air	98.3	99.2^{***}	99.3^{***}

Table 4: End-to-end agent accuracy (%, $N{=}1,500$). "No tools": the sub-agent writes MCP code, fine-tuned without tools. "Sub-agent" and "Main-agent" both use generated tools, the latter called directly by the main agent. Significance vs. no-tools baseline: $^{**}p < 0.01$, $^{***}p < 0.001$ (Tables 12–14).

Because tool outputs are small (a few tokens), reading them inline costs little, so the main agent can call tools without a sub-agent, removing the original rationale for the sub-agent layer (Section 3). This direct-call configuration's p50 is a further 55% below row 2, but it also runs GLM-4.5-Air, confounding the two effects. To isolate the architecture, we run an offline ablation that holds the model and inference framework fixed (GLM-4.5-Air, three seeds, 1,500 alarms): direct calls alone reduce p50 by 62% and p99 by 30% relative to the sub-agent architecture.

5.2 End-to-End Agent Evaluation

Table 4 reports end-to-end agent accuracy on the 1,500 historical alarms across the same three configurations. Tools reduce the error rate from 2.8% to 1.8% on Qwen3 32B and from 1.7% to 0.8% on GLM-4.5-Air, relative reductions of 36% and 53%. Manual inspection of trajectories on which the sub-agent and tool configurations disagree reveals two recurring sub-agent failure modes. First, it deviates from the SOP, using a similar but different metric or aggregating where the SOP specifies

a per-entity check. Second, it reaches the correct conclusion but reports it incorrectly to the main agent. Deterministic tools eliminate both.

The agent is fine-tuned on a mixture of trajectories with and without tools (Section 3.2) so that it can fall back to manual reasoning when a tool is unavailable. When the Qwen3 32B agent fine-tuned with tools is evaluated with tools disabled, its action accuracy (97.4%) and overall accuracy (96.9%) are statistically indistinguishable from the 97.4% and 97.2% obtained without tools, so mixed-trajectory fine-tuning preserves the fallback.

6 Discussion and Production Insights

Tools transfer across agents and models with a one-time generation cost. The library generated by GLM-4.7 serves both the Qwen3 32B and GLM-4.5-Air agents without regenerating any tool. Generating the full 44-node library is a one-time cost of 800K tokens (Figure 2).

Parameterized tools cover a family of deployments. A generated tool generalizes across deployments when the quantities that vary between them are tool inputs. In our case these are numeric thresholds, which differ across physical sites with alarm preferences and baseline robot error rates. Each tool resolves them from a runtime parameter store, so one library can serve multiple sites.

Logging tool outputs makes upstream data and environment inconsistencies visible. Because a tool returns the same value on the same input, its logs reveal environment issues that run-to-run variation in generated code hides. Every invocation logs the tool version, inputs, and outputs, and a monitoring agent reviews these logs in batch, surfacing drift that no single trajectory would reveal. A flagged tool is corrected by rerunning the generate-test-repair loop and promoting a new version once it passes the held-out set and manual review. Because the agent is decoupled from any specific tool version, deploying or reverting a fix is a configuration change requiring no retraining, and no rollback has been needed since deployment.

This mechanism has revealed three pre-existing inconsistencies in the upstream environment. First, one field differed in type between the offline and production MCPs. Generated code silently adapted, whereas the tool raised a reproducible exception that localized the issue. Second, a node's MCP reported cumulative events while the SOP referred to currently active ones: the tool passed offline evaluation, and the monitoring agent flagged the resulting high positive rate in production. Third, an endpoint silently changed its return format from percentages (75) to decimals (0.75), which the monitoring agent caught from the shifted value distribution.

7 Related Work

Tool use and tool-making. Early LLM tool use focused on calling fixed APIs (Schick et al., 2023; Patil et al., 2024). A parallel line gave agents an action space of executable code (Gao et al., 2023; Yao et al., 2022; Wang et al., 2024a). A more recent line of work has the agent construct its own tools. These include methods that synthesize tools from task descriptions or datasets (Cai et al., 2024; Qian et al., 2023; Yuan et al., 2024; Wang et al., 2024c) and agents that iteratively generate and refine tools through repeated execution and testing (Wang et al., 2023; Zheng et al., 2025; Wölflein et al., 2025). Operational documents can also be transformed into tools (Zwerdling et al., 2025; Mao et al., 2026; Ye et al., 2025). Recent research confirms that prompting-based agents fail to follow SOPs reliably when evaluated against the original SOP code as a verifier (Li et al., 2025; Nandi et al., 2025), with recurring failure modes including route blindness and calculation errors (Wang et al., 2025b). Recent industrial deployments include frameworks for alert triage and root cause analysis in production systems (Liu et al., 2026a; Bharadwaj and Tu, 2026). The closest peer work for SOP-to-code is Pei et al. (2025), but it is only evaluated on a benchmark. We contribute to this line of work by compiling SOPs into tools validated against labeled production cases.

Iterative refinement and data grounding. Our work is closely related to the execution-feedback refinement family (Chen et al., 2024a; Zhang et al., 2023; Zhong et al., 2024; Ridnik et al., 2024). Olausson et al. (2024) conclude that the success of self-repair depends on the feedback. Our finding that the pipeline plateaus without environmental grounding is consistent with this. The text-to-SQL literature also finds that schema alone hits a ceiling and grounding in database values is required (Li et al., 2023; Wang et al., 2025a; Cao et al., 2026). Our findings on the importance of specification quality for tool generation echo what has been reported earlier in the code and SQL generation literature (Wretblad et al., 2024; Jin et al., 2026).

8 Conclusion

We show that repeated inference-time code generation can be optimized for production LLM agents. By compiling SOP nodes into environment-grounded, validated, and versioned tools, our system moves recurring procedural work out of the latency-critical path while preserving fallback flexibility for failures. In a deployed fulfillment-center alarm-triage agent, this reduces latency and improves end-to-end accuracy.

The broader contribution is an operating pattern, a step toward self-evolving industrial agents: the agent maintains the implementation of its action space, building, testing, and repairing tools offline rather than improvising code on every request. The action space itself, defined by the SOP, and tool promotion remain human decisions. Once grounded tools are in place, the limiting factor becomes the quality of the underlying SOP specification instead of the agent's ability to rediscover schemas or rewrite boilerplate. This makes agent behavior faster, more consistent, and more auditable, suggesting a practical route to reliable low-latency LLM systems in operational workflows.

Limitations

Our agentic tool-making loop has been deployed on one application, outbound dock alarm triage. Generalization to other application domains, such as free-form runbooks, is not established. The setting nonetheless shares properties with typical agentic workflows: a written procedure that the agent must follow and that is executed against a stable backend, and latency or auditability constraints that make inference-time code generation less appealing as an alternative.

The improvement loop retains human oversight for production safety. It already automates parts of the cycle, such as drift detection and candidate tool generation, but we cannot yet guarantee that every failure would be caught without human review, so fully automating it remains open. Clarifying an underspecified SOP is inherently a human task, because the SOP encodes the author's intent for how the procedure should behave. Creating a new tool still benefits from labeled cases. Our label-free configuration trails the labeled ceiling by less than 3pp, and a fully label-free pipeline would remove the need for hand-labeled cases when adding new nodes.

References

Anthropic. 2026. Claude opus 4.6 system card. Accessed: 2026-06-04.

Aprameya Bharadwaj and Kyle Tu. 2026. Agentic observability: Automated alert triage for adobe e-commerce. *arXiv preprint arXiv:2602.02585*.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2024. Large language models as tool makers. In *International Conference on Learning Representations*.

Bowen Cao, Weibin Liao, Yushi Sun, Dong Fang, Haitao Li, and Wai Lam. 2026. Apex-sql: Talking to the data via agentic exploration for text-to-sql. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 234–244.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024a. Teaching large language models to self-debug. In *International Conference on Learning Representations*.

Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. 2024b. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 674–688.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International conference on machine learning*, pages 10764–10799. PMLR.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Tengjun Jin, Yoojin Choi, Yuxuan Zhu, and Daniel Kang. 2026. Pervasive annotation errors break text-to-sql benchmarks and leaderboards. *arXiv preprint arXiv:2601.08778*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can llm already serve

as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36:42330–42357.

Zekun Li, Shinda Huang, Jiangtian Wang, Nathan Zhang, Antonis Antoniades, Wenyue Hua, Kaijie Zhu, Sirui Zeng, Chi Wang, William Yang Wang, and Xifeng Yan. 2025. Sopbench: Evaluating language agents at following standard operating procedures and constraints. *arXiv preprint arXiv:2503.08669*.

Bochao Liu, Zhipeng Qian, Yang Zhao, Xinyuan Jiang, Zihan Liang, Yufei Ma, Junpeng Zhuang, Ben Chen, Shuo Yang, Hongen Wan, Yao Wu, Chenyi Lei, and Xiao Liang. 2026a. Bian que: An agentic framework with flexible skill arrangement for online system operations. *arXiv preprint arXiv:2604.26805*.

Ning Liu, P Aditya Sreekar, Kalle Kujanpää, Zhaoxuan Zhu, Kaiwen Liu, Chuanneng Sun, Jorge Marchena Menendez, Matthew Bales, Tianyu Yang, Shahnawaz Alam, Rose Yu, Baoyuan Liu, Kristina Klinkner, and Shervin Malmasi. 2026b. Eluna: An agentic LLM system for automating warehouse operations with reasoning and task execution. In *Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing: Industry Track*, Budapest (Hungary). Association for Computational Linguistics.

Jiayi Mao, Liqun Li, Yanjie Gao, Zegang Peng, Shilin He, Chaoyun Zhang, Si Qin, Samia Khalid, Qingwei Lin, Saravan Rajmohan, Sitaram Lanka, and Dongmei Zhang. 2026. Stepfly: Agentic troubleshooting guide automation for incident diagnosis. *Proceedings of the ACM on Software Engineering*, 3(FSE):3070–3092.

Subhrangshu Nandi, Arghya Datta, Rohith Nama, Udita Patel, Nikhil Vichare, Indranil Bhattacharya, Prince Grover, Shivam Asija, Giuseppe Carenini, Wei Zhang, Arushi Gupta, Sreyoshi Bhaduri, Jing Xu, Huzefa Raja, Shayan Ray, Aaron Chan, Esther Xu Fei, Gaoyuan Du, Zuhaib Akhtar, and 5 others. 2025. Sop-bench: Complex industrial sops for evaluating llm agents. *arXiv preprint arXiv:2506.08119*.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is self-repair a silver bullet for code generation? In *International Conference on Learning Representations*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565.

Changhua Pei, Zexin Wang, Fengrui Liu, Zeyan Li, Yang Liu, Xiao He, Rong Kang, Tieying Zhang, Jianjun Chen, Jianhui Li, Gaogang Xie, and Dan Pei. 2025. Flow-of-action: Sop enhanced llm-based multi-agent system for root cause analysis. In *Companion Proceedings of the ACM on Web Conference 2025*, pages 422–431.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 6922–6939, Singapore. Association for Computational Linguistics.

Evelien Riddell, James Riddell, Gengyi Sun, Michał Antkiewicz, and Krzysztof Czarnecki. 2026. Stalled, biased, and confused: Uncovering reasoning failures in llms for cloud-based root cause analysis. *arXiv preprint arXiv:2601.22208*.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025a. Mac-sql: A multi-agent collaborative framework for text-to-sql. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 540–557.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Jiaming Wang, Zhe Tang, Zehao Jin, Hefei Chen, Yilin Jin, Peng Ding, Xiaoyu Li, and Xuezhi Cao. 2025b. Sop-maze: Evaluating large language models on complicated business standard operating procedures. *arXiv preprint arXiv:2510.08942*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024a. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*.

Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Jihong Wang, Fengbin Yin, Lunting Fan, Lingfei Wu, and Qingsong Wen. 2024b. Rcagent: Cloud root cause analysis by autonomous agents with tool-augmented large language models. In *Proceedings of the 33rd ACM international conference on information and knowledge management*, pages 4966–4974.

Zhiruo Wang, Daniel Fried, and Graham Neubig. 2024c. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks. *arXiv preprint arXiv:2401.12869*.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. 2024d. Agent workflow memory. *arXiv preprint arXiv:2409.07429*.

Georg Wölflein, Dyke Ferber, Daniel Truhn, Ognjen Arandjelovic, and Jakob Nikolas Kather. 2025. Llm agents making agent tools. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 26092–26130.

Niklas Wretblad, Fredrik Riseby, Rahul Biswas, Amin Ahmadi, and Oskar Holmström. 2024. Understanding the effects of noise in text-to-sql: An examination of the bird-bench benchmark. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 356–369.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Anbang Ye, Qianran Ma, Jia Chen, Muqi Li, Tong Li, Fujiao Liu, Siqi Mai, Meichen Lu, Haitao Bao, and Yang You. 2025. Sop-agent: Empower general purpose ai agent with domain-specific sops. *arXiv preprint arXiv:2501.09316*.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi Fung, Hao Peng, and Heng Ji. 2024. Craft: Customizing llms by creating and retrieving from specialized toolsets. In *International Conference on Learning Representations*.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. 2023. Scaling relationship on learning mathematical reasoning with large language models. *arXiv preprint arXiv:2308.01825*.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, Chenzheng Zhu, Congfeng Yin, Cunxiang Wang, Gengzheng Pan, Hao Zeng, Haoke Zhang, Haoran Wang, Huilong Chen, Jiajie Zhang, and 166 others. 2026. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, Kedong Wang, Lucen Zhong, Mingdao Liu, Rui Lu, Shulin Cao, Xiaohan Zhang, Xuancheng Huang, Yao Wei, Yean Cheng, and 151 others. 2025. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787.

Yuze Zhao, Jintao Huang, Jinghan Hu, Xingjun Wang, Yunlin Mao, Daoze Zhang, Hong Zhang, Zeyinzi Jiang, Zhikai Wu, Baole Ai, Ang Wang, Wenmeng Zhou, and Yingda Chen. 2024. Swift:a scalable lightweight infrastructure for fine-tuning. *Preprint*, arXiv:2408.05517.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. 2025. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 851–870, Bangkok, Thailand. Association for Computational Linguistics.

Naama Zwerdling, David Boaz, Ella Rabinovich, Guy Uziel, David Amid, and Ateret Anaby Tavor. 2025. Towards enforcing company policy adherence in agentic workflows. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 595–606, Suzhou (China). Association for Computational Linguistics.

Appendix

A Experimental Setup

Seeds. Per-node pass@1 results in Section 4 use 30 independent generation runs per configuration, except the SOP column in Table 1 and the augmented-data row in Table 2, which use 10. End-to-end agent accuracy in Section 5 uses 3 seeds per configuration on the same 1,500 alarm scenarios.

Inference framework. The deployed agent (Qwen3 32B and GLM-4.5-Air) runs on vLLM (Kwon et al., 2023) for both offline benchmarks and production. The open-weight tool-makers (GLM-4.7, Qwen3 235B, GLM-4.7 Flash) likewise run on vLLM. GLM-5 and the Claude ablations are served via Amazon Bedrock.

Hardware. Inference, both offline benchmarks and production, runs on a single 8×H100 instance. Fine-tuning runs on 64 H100s for Qwen3 32B and 128 H100s for GLM-4.5-Air.

Latency measurement. For competitive reasons we cannot disclose absolute production latencies. Hence, Table 3 reports figures normalized to the no-tools sub-agent baseline.

Fine-tuning. We follow the training procedure of Liu et al. (2026b). We distill GLM-4.7 (Zeng et al., 2025) into the two deployed students, Qwen3 32B and GLM-4.5-Air. A teacher trajectory is retained only when its extracted observations, root causes, and actions all match the ground-truth labels. This filter accepts approximately 92% of trajectories (93% on the with-tools slice with dropout, 4,171/4,500, and 91% on the no-tools slice, 1,358/1,500). The same filter applied to the no-tools baseline accepts 91% of trajectories (6,786/7,500), so tool availability also raises the teacher's own accuracy. Students are fine-tuned with LoRA (Hu et al., 2022) using MS-SWIFT (Zhao et al., 2024), with rank and α of 256 and learning rate 2×10^{-4} for Qwen3 32B and 32 and 1×10^{-4} for GLM-4.5-Air. Both train at batch size 64 with a 5% linear warmup and cosine decay to zero, on 64 and 128 H100 GPUs respectively. The with-tools fine-tunes use one epoch, while the no-tools baseline fine-tunes use two epochs (Liu et al., 2026b). After the training, we merge the LoRA adapters so inference carries no added overhead.

B Ground-Truth Labeling

The pipeline consumes labeled (input, verdict) cases per node (Section 3.1). Human annotations could provide these labels directly, and only a few labels are needed: pass@1 reaches 91.8% from five labeled cases per node, and the label-free variant reaches 91.6% with none (Appendices C and D). However, we lacked the annotator capacity to label all the training and evaluation cases used in our experiments, so we produced labels programmatically. Subject-matter experts provide step-by-step guidance for each node's logic. We implement this logic, verify the implementation with them, and run the resulting offline labeler over the historical cases.

The labeler is not itself deployable as a production tool. It is written against dashboard data logged into a data lake of historical snapshots, whereas production tools query live MCPs with different schemas. Historical snapshots are complete, whereas live MCP data may arrive incomplete or with gaps (nulls, missing rows, partial windows), so the tools must be robust to varying patterns of missing data. Production tools also retrieve per-site thresholds from the runtime parameter store and return the observed value, threshold, and explanation alongside the verdict. The pipeline's task is to synthesize implementations that meet this production contract and reproduce the reference verdicts. We expect the pipeline's advantage over hand-written tools to lie less in the initial implementation than in maintenance: the upstream inconsistencies in Section 6 were corrected by rerunning the pipeline, whereas hand-maintained tools would consume engineering time with every environment change, and this cost scales with the size of the tool library.

C Further Method Ablations

C.1 Impact of Repair Loop

The deployed pipeline's repair loop uses single-path reflection, where the model produces a brief diagnosis of the failures and writes a corrected tool after each test round. This is repeated for three rounds. We compare this strategy against two alternatives: no reflection (the model repairs from failures alone without a separate diagnosis step), and multi-hypothesis branching. In multi-hypothesis branching, three independent diagnoses are generated per round. Each is tried as a separate repair branch and we keep the lowest-failure branch.

Figure 3 shows that reflection and branching reach similar performance at similar total cost (556K tokens for reflection at 10 rounds, 597K for branching at 4 rounds). Branching reaches the ceiling in fewer rounds because it explores three diagnoses in parallel each round. No-reflection plateaus around 92.5% because additional rounds repeat the same diagnostic errors.

C.2 Impact of the Data-Collection Trace

The two intermediate conditions in Table 5 each prepend only a part of the trace to the tool-maker's prompt: the schema metadata observed by the sub-agent (column names, dtypes, row counts), or the sub-agent's verdict graded against the labeled answer.

Schema metadata adds $+1.8$pp and grading the sub-agent's verdict adds $+1.7$pp, whereas the full trace adds another $+1.2$pp by showing the tool-maker working code. This is a form of execution-grounded feedback that prior work has identified as a significant contributor to repair quality (Olausson et al., 2024).

Two concrete failure modes illustrate the gain.

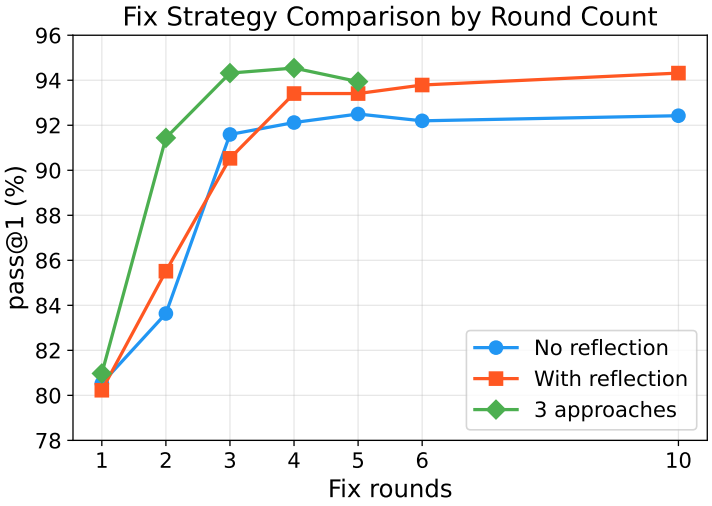


Figure 3: pass@1 across repair rounds for each repair strategy, without the data-collection trace (SOP+R, Opus). No-reflection plateaus around 92.5%, while reflection peaks at 94.3% by round 10 but underperforms no-reflection at low round counts. Multi-hypothesis branching reaches 94.3% by round 3 at roughly 3× the per-round cost.

Trace content	pass@1	Prompt content
No trace	91.6	SOP only
Metadata only	93.4	Cols, shapes, rows
Grading only	93.3	Determination, label
Full trace	94.5	Code, MCP, grading

Table 5: Trace-content ablation (Opus, three repair rounds). Each component contributes a consistent gain. Confidence intervals are in Table 10.

Including the grading corrects semantic biases that are underspecified by the SOP. One tool should return true on zero violations, and models consistently emit a `count > 0` guard. When the sub-agent makes the same error on a sampled case, the trace shows the sub-agent's verdict marked incorrect, and the tool-maker drops the guard in subsequent generations. Including the code can help prevent boundary bugs. Zero-shot tool-maker code often rounds the outcome before comparing to the threshold, whereas the sub-agent's code does not exhibit this failure mode.

C.3 Interaction Between the Repair Loop and the Trace

Table 6 reports each repair-loop strategy at three repair rounds, the deployed budget, with and without the data-collection trace. The main finding is that the trace yields the largest gain under reflection (+4.0pp). At three rounds without the trace, reflection underperforms no-reflection (90.5% vs. 91.6%), though reflection eventually catches up as shown in Appendix C.1. The mechanism is that the trace's working code and graded verdict ground the diagnosis step in concrete failure modes, so reflection no longer expends repair rounds on incorrect hypotheses. No-reflection benefits less (91.6% to 92.8%, +1.2pp) because without an explicit diagnosis step, it does not fully exploit the information in the trace. Branching is essentially unchanged (94.3% to 94.4%). With three independent diagnoses, branching explores enough alternatives to converge even without the data-collection trace. With the trace, reflection (94.5%, 395K tokens) and branching (94.4%, 496K tokens) reach the same accuracy, but reflection costs 20% fewer tokens, so the deployed pipeline uses reflection.

C.4 Data Efficiency

Each new SOP node requires labeled training cases to generate its tool. Hence, a practical pipeline must reach high pass@1 from few labels. With grounding, pass@1 reaches 91.8% at $N{=}5$ training cases per node, within 3pp of the full-set ceiling (Figure 4). Removing labels entirely is also feasible. We investigate a label-free tool-maker that uses judge-filtered pseudo-labels and multi-candidate voting instead of the labeled repair-loop test cases.

Strategy (3 rounds)	No trace	Data + grading	Δ
No reflection	91.6	92.8	+1.2
With reflection	90.5	94.5	+4.0
Three hypotheses	94.3	94.4	+0.1

Table 6: Interaction between data grounding and repair strategy (Opus, 3 rounds). pass@1 (%). Confidence intervals and t-tests are in Table 11.

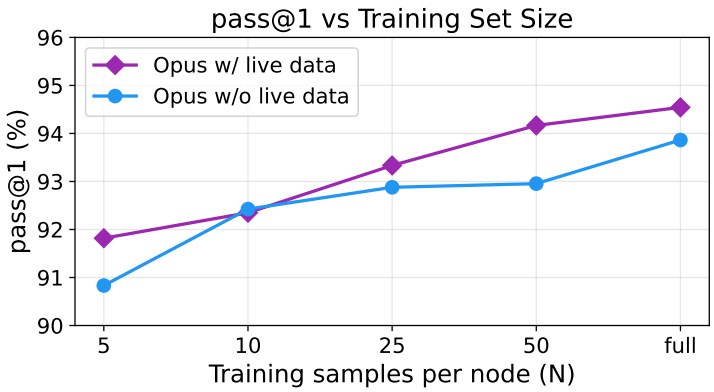


Figure 4: pass@1 vs. training-set size, with and without data grounding (Opus).

This configuration reaches 91.6% pass@1, also within 3pp of the labeled ceiling (Appendix D).

C.5 Token Scaling

We perform two follow-ups to the main-text GLM-4.7 ablation (Figure 2). First, we investigate whether GLM-4.7 repair-only catches up to the deployed configuration at higher token budgets. Second, we report the same ablation on a stronger tool-maker, Opus 4.6.

SOP+R does not scale with tokens on GLM-4.7. Table 7 compares the best SOP+R configuration we ran against a higher-budget SOP+R configuration that explores three repair branches per round. Despite spending 55% more tokens, branching reduces pass@1 from 89.3% to 87.0%. The cheapest SOP+D+R configuration (1 sample, 3-round repair) reaches 91.2% at 596K tokens, which is fewer tokens than the best SOP+R point. The deployed SOP+D+R configuration in the main text reaches 94.5% at 799K. No SOP+R configuration we ran exceeds 89.3%.

Opus 4.6 component ablation. Figure 5 repeats the Figure 2 ablation on Opus 4.6. The repair loop alone reaches 94.3% pass@1 at 556K tokens with 10 rounds. The deployed configuration reaches comparable accuracy (94.5%) at 395K tokens, a 29% token reduction. The trace helps Opus reach the ceiling at lower cost, but the ceiling is unchanged.

D Label-Free Generation

We investigate how to replace the labeled repair loop with a pipeline that combines voting and LLM judges. For each tool, K independent sub-agents each investigate one sampled unlabeled case. An LLM judge filters out traces that are not consistent with the SOP. From the surviving traces, we generate N independent candidates, and execute each candidate against the K sampled cases. We then assign a pseudo-label for each case by majority vote across the surviving traces and the N candidate outputs, and iterate a repair loop for R rounds. Each round, every candidate is executed on the K investigated cases. Candidates that agree with all current pseudo-labels and pass a code-review judge (an LLM that inspects the candidate tool's code for consistency with the SOP) are carried forward unchanged. The remaining candidates see their disagreement cases as failure feedback, write a short diagnosis, and regenerate. After all N candidates have either regenerated or been carried forward, the pseudo-labels for each case are recomputed by majority vote over the updated candidate outputs and the judge-passing sub-agent determinations. The final candidate is selected by counting how many of the K investigated cases each candidate matches the pseudo-label on. Ties are broken first by the code-review judge, then by code length.

Configuration	Tokens (K)	pass@1
SOP+R, 4 rounds	635	89.3
SOP+R, 3 rounds × 3 branches	982	87.0
SOP+D+R, 1 sample, 3 rounds	596	91.2
SOP+D+R, 3 samples, 3 rounds	799	94.5

Table 7: GLM-4.7 SOP+R configurations at higher token budgets, paired with two SOP+D+R configurations for reference. +D adds the data-collection trace; +R adds the test-repair loop. Per-seed tokens are means over 30 seeds (10 for the 3-branches row).

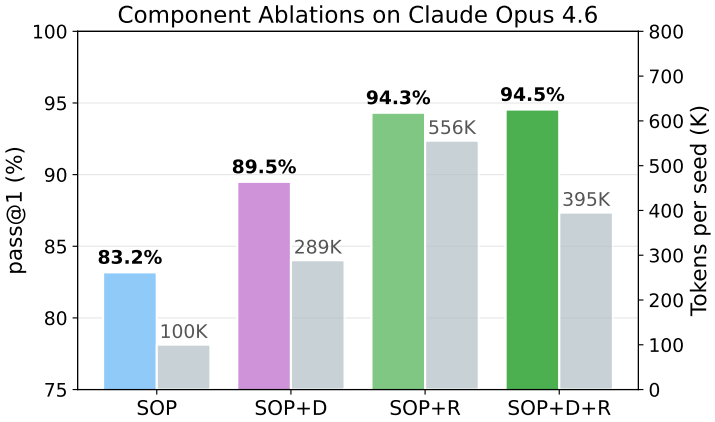


Figure 5: Per-node pass@1 (left bars) and generation cost in tokens per seed (right bars) for each ablation configuration on Claude Opus 4.6. The deployed configuration matches the repair-only ceiling within seed variance at 29% fewer tokens.

The headline configuration is GLM-4.7 as the tool-maker, Claude Sonnet 4.5 as the trace judge, and Claude Opus 4.7 as the reflector agent, with $K{=}3$, $N{=}10$, and $R{=}5$. The reflector here is Opus 4.7 due to model availability at the time of this experiment. We do not attribute the result to the newer model. This reaches 91.6% pass@1 (cf. the 94.5% labeled ceiling). The most significant component is voting across N candidates: $N{=}1$ loses 8.6pp. Other ablations (removing the repair loop, reducing to $K{=}1$, skipping the code-review judge) each cost 2.1–3.4pp. Once we clarify the specification as in Section 4.3, Opus 4.7 without labels reaches 99.1% pass@1.

E The Repair Loop Unlearns Correct Code

The fourth residual node in Section 4.3 is a precondition for a downstaffing action: downstaffing only proceeds if the collector belt is sufficiently empty. The SOP requires, for each quadrant, a 5-minute average leading up to that quadrant's most recent timestamp, with no rows older than 10 minutes included. The round-0 candidate below is mostly correct, but uses $>$ instead of the SOP's $\geq$ at the start boundary, which produces 6 test failures in round 1. The round-2 candidate replaces the 5-minute averaging logic with a simple average over the full 10-minute window, which produces fewer test failures, but no longer matches the SOP.

```
# Round 0: per-quadrant 5-minute window
# (uses > instead of the SOP's implied >=)
fresh = df[
    (df['five_minute_interval'] >= start_time) &
    (df['five_minute_interval'] <= current_time)
]
for q, col in quadrants.items():
    q_data = fresh[fresh[col].notna()]
    analysis_time = q_data['five_minute_interval'].max()
    five_min_start = max(analysis_time - timedelta(
      minutes=5), start_time)
    subset = q_data[
        (q_data['five_minute_interval'] >
      five_min_start) &
        (q_data['five_minute_interval'] <= analysis_time
      )
    ]
    avgs[q] = subset[col].mean()

# Round 2: 5-minute window collapsed to the
# full 10-minute window per quadrant
fresh = df[
    (df['five_minute_interval'] >= start_time) &
    (df['five_minute_interval'] <= current_time)
]
for q, col in quadrants.items():
    avgs[q] = fresh[col].dropna().mean()

# Both rounds: thresholding and outcome (unchanged)
north_avg = (avgs['NW'] + avgs['NE']) / 2
south_avg = (avgs['SW'] + avgs['SE']) / 2
holds = (north_avg <= north_threshold and
         south_avg <= south_threshold)
```

F Per-Component Output-Token Breakdown

Section 5 reports a 58% reduction in per-alarm output tokens with tools. Table 8 gives a breakdown by agent role (main vs. sub-agent) and by content type (reasoning, prose text, code blocks). The main contributor to the reduction is sub-agent code generation (-80%), which is expected when per-invocation code generation is replaced with tool calls. The sub-agent also needs to reason less, as calling the tool is simpler than writing the code. The main agent's tokens also fall (-16%), as reading a structured verdict is cheaper than parsing the sub-agent's free-text answer. The 20% of sub-agent code tokens that remain consist mostly of the tool calls themselves, which appear as code blocks under the CodeAct-style execution model, and the agent's programmatic to-do list.

Component	With tools	No tools	Δ
Main: thinking	1,118	1,349	-17%
Main: text	391	436	-10%
Main: code	202	259	-22%
Main total	**1,711**	**2,044**	$\mathbf{-16\%}$
Sub: thinking	2,013	4,144	-51%
Sub: text	1,035	1,358	-24%
Sub: code	1,466	7,452	-80%
Sub total	**4,514**	**12,954**	$\mathbf{-65\%}$
Total	**6,225**	**14,998**	$\mathbf{-58\%}$

Table 8: Per-trace output-token breakdown by agent role and content type, per-alarm production means (Qwen3 32B).

G Confidence Intervals and Significance Tests

Point estimates reported elsewhere in the paper are repeated here with 95% confidence intervals (mean $\pm 1.96\times$ SE over per-seed pass@1 scores) and corresponding significance tests.

Per-seed pass@1 with confidence intervals. Tables 9, 10, and 11 give the build-time pipeline results from Section 4 and Appendix C with 95% CIs and paired t-tests over 30 seeds (10 for the SOP column).

Significance of end-to-end correctness (Table 4). For each model, we run three seeds with tools and three seeds without tools, and we evaluate each seed on the same 1,500 tasks. To compute confidence intervals and p-values, we aggregate at the task level because we cannot treat the 4,500 (task, seed) pairs as independent. We average the three seed-level results per task, and we resample tasks with replacement. The reported 95% CIs are the 2.5th and 97.5th percentiles of the resampled accuracy difference (with tools minus without tools) over $n_{\text{boot}} = 5{,}000$ bootstrap iterations. The reported p-value is the fraction of bootstrap iterations in which the resampled accuracy difference was zero or had the opposite sign to the observed delta, multiplied by two for the two-sided test.

Improvements in overall accuracy are significant for both agent models in the sub-agent architecture (Tables 12 and 13) and persist when the main agent calls tools directly (Table 14). Qwen3 32B's gains come primarily from improved root-cause and action accuracy, whereas GLM-4.5-Air analyzes observations more accurately.

Significance of repair loop and trace contributions. For the two best models, GLM-4.7 and GLM-5, all three pairwise comparisons are significant ($p < 10^{-4}$). The deployed configuration (SOP+D+R) is significantly better than either single-component ablation, and both components contribute. On Qwen3 235B, SOP+D+R is not significantly better than SOP+R alone. Manual inspection of regressions shows that Qwen3 235B incorrectly infers schema details from the data-collection trace (wrong tool selection, wrong column types, missing columns), producing tools that crash or return "No data" on inputs the SOP-only configuration handles correctly. On GLM-4.7 Flash, the smaller tool-maker fails to productively use the repair loop without the trace to ground the repairs. The performance of SOP+D+R exceeds that of SOP+R and SOP+D, and neither component alone matches the gain of the combined configuration.

Significance of specification interventions. Both interventions, clarifying the SOP and augmenting the training data, are individually significant (Table 16).

	SOP	SOP+D	SOP+R	SOP+D+R
GLM-4.7	75.7 [74.5, 76.8]	84.9 [83.5, 86.2]	89.3 [87.8, 90.8]	94.5 [93.8, 95.1]
GLM-5	76.8 [75.4, 78.3]	83.4 [82.0, 84.8]	88.0 [87.0, 89.0]	93.6 [92.8, 94.3]
Qwen3 235B	74.2 [72.3, 76.2]	71.6 [70.2, 72.9]	79.9 [78.9, 80.9]	81.1 [79.9, 82.2]
GLM-4.7 Flash	56.4 [54.4, 58.5]	60.4 [59.2, 61.6]	58.7 [57.7, 59.8]	65.8 [64.3, 67.3]

Table 9: Component ablations of the deployed pipeline with 95% CIs (pass@1, %), corresponding to the point estimates in Table 1. +D adds the data-collection trace (Section 3.1); +R adds the test-repair loop. SOP+D+R is the deployed configuration. Differences between SOP+D+R and the SOP+D and SOP+R ablations are statistically significant on GLM-4.7, GLM-5, and GLM-4.7 Flash (significance tests in Table 15).

Trace content	pass@1
No trace	91.6 [90.7, 92.5]
Metadata only	93.4 [92.9, 93.9]
Grading only	93.3 [92.4, 94.3]
Full trace	94.5 [93.8, 95.3]

Table 10: Trace-content ablation with CIs (Table 5). Opus, three repair rounds.

Strategy	No trace	Data + grading	Δ	t	p
No reflection	91.6 [90.7, 92.5]	92.8 [92.2, 93.4]	$+1.21$	2.50	0.018
With reflection	90.5 [89.5, 91.5]	94.5 [93.8, 95.3]	$+4.01$	6.25	$<10^{-6}$
Three hypotheses	94.3 [93.3, 95.3]	94.4 [93.6, 95.2]	$+0.08$	0.11	0.91

Table 11: Data grounding and repair strategy with CIs and paired t-tests (Table 6). Opus, three repair rounds, 30 seeds.

Metric	With	Without	Δ	95% CI	p
Obs acc.	99.62	99.71	-0.09	$[-0.38, +0.18]$	0.57
RC acc.	100.00	99.33	$+0.67$	$[+0.33, +1.04]$	$<10^{-3}$
Act acc.	98.38	97.44	$+0.93$	$[+0.31, +1.58]$	0.0024
Overall acc.	98.18	97.18	$+1.00$	$[+0.31, +1.69]$	0.0052

Table 12: Qwen3 32B, with vs. without tools (%). 95% CIs and p-values are bootstrap-by-task ($n_{\text{boot}} = 5{,}000$).

Metric	With	Without	Δ	95% CI	p
Obs acc.	99.87	99.18	$+0.69$	$[+0.38, +1.00]$	$<10^{-3}$
RC acc.	99.71	99.49	$+0.22$	$[-0.04, +0.51]$	0.12
Act acc.	99.36	99.29	$+0.07$	$[-0.22, +0.38]$	0.70
Overall acc.	99.24	98.29	$+0.96$	$[+0.51, +1.42]$	$<10^{-3}$

Table 13: GLM-4.5-Air, with vs. without tools (%). 95% CIs and p-values are bootstrap-by-task ($n_{\text{boot}} = 5{,}000$).

Metric	Main-agent direct	Without	Δ	95% CI	p
Obs acc.	99.91	99.18	$+0.73$	$[+0.44, +1.04]$	$<10^{-3}$
RC acc.	99.98	99.49	$+0.49$	$[+0.24, +0.76]$	$<10^{-3}$
Act acc.	99.40	99.29	$+0.11$	$[-0.22, +0.44]$	0.57
Overall acc.	99.31	98.29	$+1.02$	$[+0.56, +1.49]$	$<10^{-3}$

Table 14: GLM-4.5-Air with main-agent direct tool calls vs. without tools (%). 95% CIs and p-values are bootstrap-by-task ($n_{\text{boot}} = 5{,}000$).

Comparison	Δ	t	p
GLM-4.7: SOP+R vs. SOP+D	$+4.46$	4.60	7.7×10^{-5}
GLM-4.7: SOP+D+R vs. SOP+R	$+5.15$	6.32	6.6×10^{-7}
GLM-4.7: SOP+D+R vs. SOP+D	$+9.62$	13.58	4.2×10^{-14}
GLM-5: SOP+R vs. SOP+D	$+4.57$	4.78	4.6×10^{-5}
GLM-5: SOP+D+R vs. SOP+R	$+5.55$	7.52	2.7×10^{-8}
GLM-5: SOP+D+R vs. SOP+D	$+10.12$	13.21	8.5×10^{-14}
Qwen3 235B: SOP+R vs. SOP+D	$+8.34$	10.38	2.8×10^{-11}
Qwen3 235B: SOP+D+R vs. SOP+R	$+1.14$	1.72	0.096
Qwen3 235B: SOP+D+R vs. SOP+D	$+9.48$	11.42	3.0×10^{-12}
GLM-4.7 Flash: SOP+R vs. SOP+D	-1.66	-2.12	0.042
GLM-4.7 Flash: SOP+D+R vs. SOP+R	$+7.05$	6.78	1.9×10^{-7}
GLM-4.7 Flash: SOP+D+R vs. SOP+D	$+5.39$	5.38	8.8×10^{-6}

Table 15: Significance of Table 9 ablation differences. +D adds the data-collection trace; +R adds the test-repair loop. Paired t-tests over 30 seeds. Δ in pp.

Comparison	Δ	t	p	n_x	n_y
Aug. data vs. baseline	$+2.05$	3.25	4.1×10^{-3}	10	30
Clarified SOP vs. aug. data	$+3.33$	6.49	9.3×10^{-5}	30	10
Clarified SOP vs. baseline	$+5.38$	14.22	3.1×10^{-15}	30	30

Table 16: Significance of the specification ceiling interventions (Table 2). Two-sample Welch's t-tests over per-seed pass@1. Δ in pp. n_x, n_y are the per-condition seed counts.