2026-09-17 · Research

Toward Recursive Self-Improvement: How GLM Built Its Own Inference Infrastructure

Try it at Z.ai ↗ Call it at Z.ai ↗ Z.ai Coding Plan ↗

As we develop GLM, the model sometimes exhibits capabilities that surprise us, and even unsettle us.

In October 2025, we began researching how to strengthen its cybersecurity capabilities. Our reasoning at the time was straightforward: cybersecurity is a natural extension of coding. A model that can understand complex code should also be able to understand its vulnerabilities. We did not anticipate what would follow. In less than a year, our security partners used GLM to discover thousands of vulnerabilities in real-world codebases. The model began to reshape cybersecurity, while also introducing dangers that had not existed before. To enable responsible use of this capability, we had to design a trusted access program.

The most recent moment that shook us came from a more fundamental shift: GLM is increasingly helping build AI itself. We watched the model complete an infrastructure task that would previously have taken a team of experienced infrastructure engineers weeks. When we realized that this work would directly change how the next generation of models is trained, we became even more convinced: our successors are the AI systems we are creating ourselves.

Frankly, before GLM-4.7, our internal use of GLM for coding involved a certain amount of obligation. It was, after all, our own creation. At that point, product-market fit for coding had yet to arrive. Today, GLM-5.3 has become an indispensable daily coding partner for everyone on the team, and it is moving steadily toward replacing us. If this trend continues, given enough compute and enough time, its endpoint is a system that can design and train its own successor entirely autonomously. This is known as Recursive Self-Improvement, or RSI.

We are not there yet, but early forms of it are already emerging. This article documents one such early example.

Taking a model from its first successful run on new hardware to a high-performance inference service that can reliably handle production traffic is a major systems engineering undertaking. The launch of GLM-5.3-Flash followed the same path. We built a complete production-grade inference service from scratch on a cluster of more than 100,000 Chinese-made AI accelerators. All production inference for GLM-5.3-Flash runs on this system.

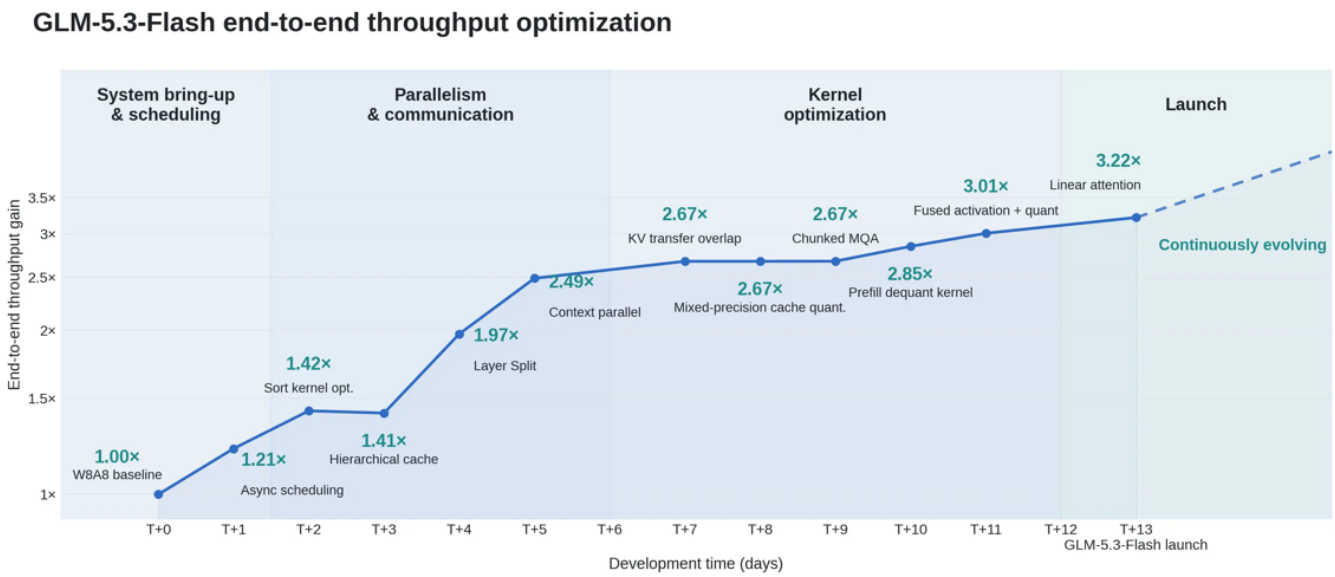
This was not easy. No one had previously deployed a cluster of Chinese-made accelerators at this scale. We faced relatively limited chip memory capacity and bandwidth, while also needing to support a new model architecture, a 1M-token context window, and multimodal requests. The ecosystem was immature, kernel support was incomplete, and much of what should have been documented had to be guessed. In the end, the work was completed. It was not done by a team of infrastructure engineers alone. Much of the work was carried out by an Infra Agent powered by GLM-5.3.

What happened next is already familiar. GLM-5.3-Flash was tested through real-world usage on OpenCode and OpenRouter under the anonymous model name Ox-Alpha. Within a week of launch, it became the most-used model on both platforms, processing more than 62 trillion tokens in six days.

We implemented a series of aggressive memory optimizations, including custom approaches that traded compute for bandwidth and communication for device memory. The resulting stack combined several key techniques: intra-node tensor parallelism for linear attention and the LM Head, ReplaySSM, W8A8 quantization, mixed-precision cache quantization using INT8/FP8/BF16, and Layer Split. On top of this, we introduced an Encode-Prefill-Decode (EPD) disaggregated architecture. Together, these optimizations

improved end-to-end serving performance by roughly 3×. Both hardware utilization efficiency and per-token cost reached levels comparable to mainstream NVIDIA GPUs.

With the Infra Agent's feedback loop running throughout the optimization process, GLM-5.3-Flash went from initial model adaptation to production readiness in less than two weeks, ultimately tripling end-to-end throughput relative to the initial baseline. Figure 1 shows the performance trajectory of GLM-5.3-Flash from its first successful run to production launch.



> ***Figure 1: The end-to-end throughput evolution of GLM-5.3-Flash.***

Throughout this process, we gradually realized that the Infra Agent's engineering effectiveness depended not only on the model's code generation and reasoning capabilities, but even more on whether the system could continuously provide useful feedback that could be traced to specific causes. A codebase provides only static context. Numerical discrepancies, performance regressions, and missed optimization targets in an inference system often arise from dynamic interactions across multiple layers, including kernel implementations, parallelism strategies, communication behavior, memory management, and serving orchestration.

Even if an agent can understand the entire codebase, feedback such as "numerical accuracy test failed," "TTFT increased by 30%," or "output throughput dropped by 20%" after a change still leaves it struggling to determine which layer is responsible, why its current hypothesis is wrong, and what it should test next. End-to-end metrics can tell an agent that results got worse, but they cannot explain why.

Therefore, alongside improving the agent's ability to write and modify code, we need to solve a more fundamental systems problem: **How do we turn sparse end-to-end results into fine-grained, attributable engineering feedback that directly guides the next action?** This is also the key to building an effective feedback loop for the Infra Agent.

From End-to-End Metrics to Attributable Feedback

Traditional inference system optimization has no shortage of tests, logs, profiling tools, or microbenchmarks. But these are usually scattered across different tools and engineering stages. Experienced engineers use the results of a load test to decide what to observe next, progressively inspecting kernel outputs, execution timelines, communication events, or thread states, and connecting information from different tools.

For an agent, unless these observation and validation methods are organized into directly accessible, repeatable workflows, the feedback it can actually use remains sparse. It may know that throughput has missed the target, yet still be unable to determine:

- Is a particular kernel taking too long, or is the compute device sitting idle?
- Is KV Transfer itself too slow, or is higher-level scheduling failing to advance transfers promptly?
- For which input shapes does an optimization work, and under what conditions does it regress?

A single end-to-end metric cannot answer these questions. We therefore incorporated correctness tests, runtime logs, execution traces, runtime events, microbenchmarks, and end-to-end metrics into the agent's iteration workflow, breaking the full system optimization process into steps that can be observed and validated locally. Kernel-level comparisons verify numerical correctness. Microbenchmarks measure local performance under specific input conditions. Execution traces and runtime events reveal the timing relationships among computation, waiting, and communication. The agent can choose the appropriate validation method for its current hypothesis, rather than waiting for a full service deployment and end-to-end load test after every change.

We call this approach "dense feedback." Here, "dense" does not mean feeding the agent as many logs and metrics as possible. Instead, it emphasizes three characteristics.

First, feedback must be sufficiently local. Wherever possible, it should be tied to specific engine launch parameters, code changes, kernels, input conditions, threads, execution intervals, or code paths, helping the agent narrow the scope of the problem. For example, instead of reporting that "model accuracy dropped after a fusion optimization," identifying the output differences for a specific request before and after the change gives the agent a better basis for constructing a minimal reproduction and analyzing the cause.

Second, feedback must be inexpensive and timely to obtain. Whenever the agent proposes a hypothesis, makes a change, or constructs a set of controlled experiments, it should have an appropriate way to validate it. Questions that can be answered by a kernel test or a local microbenchmark should not require a full service deployment and end-to-end load test every time. Shorter validation cycles help the agent correct course promptly and spend less effort on unproductive hypotheses.

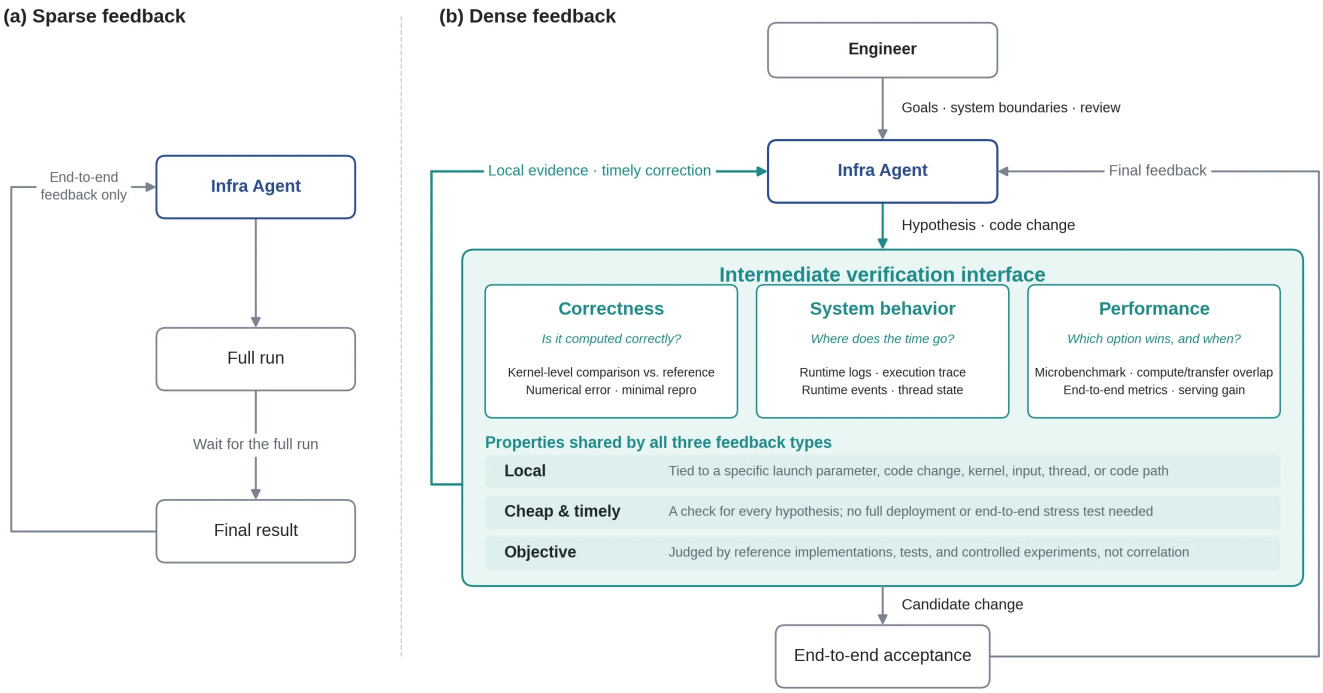
Third, feedback must support objective verification. Whether a change is correct and whether performance has improved should be determined by reference implementations, test results, and comparable experimental metrics. Runtime signals can help the agent identify possible causes, but correlations between observations alone cannot establish a root cause. Controlled experiments are still needed to verify whether a change to a specific path produces the expected effect.

Together, these three characteristics determine whether feedback is actionable. Correctness feedback answers, "Is the computation correct?" System behavior feedback identifies, "Where is the time going?" Performance feedback determines, "Which approach works better, and under what conditions?" Validation methods need not follow a fixed sequence. They should match the current hypothesis so that every experiment answers a specific question.

Local validation and end-to-end testing serve different roles in this process. The former eliminates incorrect or ineffective changes early and identifies candidates worth pursuing. The latter confirms whether local gains translate into real serving improvements and whether a proposed change introduces new regressions under actual workloads.

Based on these principles, the GLM-5.3-Flash launch established an optimization loop

Based on these principles, the GLM-5.3-Flash launch established an optimization loop involving engineers, the Infra Agent, and the experimental environment. Engineers defined objectives and system boundaries. The agent handled analysis, hypotheses, and code changes. The experimental environment provided layered, timely, and verifiable feedback. Together, they transformed a diagnostic process previously connected by engineers' experience into an engineering workflow the agent could execute continuously.



> ***Figure 2: The Infra Agent optimization loop built around dense feedback.***

This progression included both performance optimizations that directly increased throughput and bug fixes that did not immediately improve throughput but were essential to launching the system correctly and reliably. The following three cases illustrate how dense feedback helped the agent ensure that the system "computes correctly," explain "why it is not running fast enough," and explore "how to make it run faster."

Correctness Feedback: Helping the Agent Determine Whether the Model Computes Correctly

Inference performance optimization must be grounded in numerical correctness. For the agent, validation begins with establishing exactly which computations the inference engine

performs. High-level parallelism strategies change how kernel inputs are partitioned, which execution paths are taken, and how results are combined. Testing a kernel's output only under unpartitioned conditions is not enough to cover its behavior in an actual deployment. To address this, we established a mapping from the inference engine's parallelism strategies to kernel implementations, converting system-level deployment configurations into kernel-level tasks the agent could verify individually. This mapping helped the agent identify which kernels a parallel configuration involves, how inputs are partitioned, and which computation paths need to be compared against unpartitioned implementations.

On this basis, we had the agent compare numerical accuracy across different partitioned and unpartitioned execution paths. For the same inputs, after aligning computational semantics and output positions, it checked whether results from different execution methods satisfied numerical error tolerances. This connected parallel configurations, kernel paths, and observed errors. Once a test exposed a discrepancy, the agent could continue investigating the corresponding partitioning scheme and computation path, rather than restarting diagnosis at the level of the entire model.

It was during this kernel validation process that we discovered a numerical accuracy issue in the KDA kernel's Context Parallelism (CP) path. Differences between CP and non-CP results directed the investigation toward the state propagation and merging computations introduced by parallel execution.

CP partitioning requires merging states from different context shards. Its core computation can be simplified as follows:

```
M = tl.dot(M_chunk, M)     # Merge state transformations across
shards
S_next = tl.dot(M, S) + H  # Update the initial state for the next
shard
```

In the original implementation, `tl.dot` defaulted to TF32 computation for better performance, even when its inputs were FP32. This lower computational precision caused errors to accumulate during transformation merging and state updates, becoming more pronounced with long contexts.

The fix was to explicitly set `input_precision="tf32x3"` for both operations. This combines three TF32 Tensor Core operations to produce a higher-precision result, reducing

accumulated error while preserving as much of the Tensor Core performance advantage as possible.

In this case, the feedback environment began playing a role before the issue surfaced. The mapping from parallelism strategies to kernels defined what needed to be tested. Comparisons between partitioned and unpartitioned paths exposed numerical discrepancies. Analysis of computational precision explained their source. Regression tests provided a continuing basis for validating the changes. For the agent, this workflow turned system-level parallelism design into executable, traceable correctness tasks. After local validation, the candidate implementation still had to return to the target deployment for final acceptance testing of model-level accuracy and serving performance.

The numerical accuracy fixes have been merged upstream into Flash Linear Attention. See PR #1180 for details.

System Behavior Feedback: Pinpointing a KV Transfer Concurrency Bottleneck

For system-level performance problems, clearly defined test scenarios and performance constraints give the agent a starting point for identifying anomalies and choosing an analytical direction.

Our inference optimization engineers defined test scenarios for the agent, including Prefill alone, Prefill + KV Transfer, and Decode alone, to isolate the performance effects of different execution stages and their combinations. They also set acceptance criteria for each scenario. For example, under the same workload, the performance gap between Prefill + KV Transfer and the Prefill-only baseline should not exceed 5%.

However, the agent found that the performance gap exceeded 20% in some scenarios. This feedback narrowed the investigation to the additional overhead and concurrency interactions introduced by KV Transfer. The agent then examined the KV Tran
detail and identified an anomaly: in these scenarios, Python-side KV Transfer
never overlapped with the DeepEP dispatch/combine call intervals.

This observation led the agent to investigate concurrency between DeepEP a

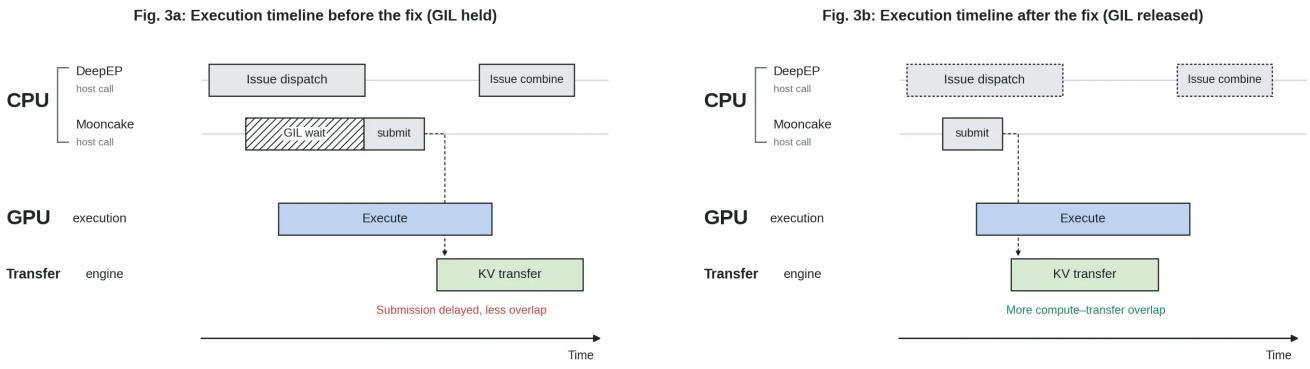


Transfer, following the call chain to the Python/C++ boundary. In DeepEP v1.2.1, the version we used, neither `intranode_dispatch` nor `intranode_combine` explicitly released the Python GIL. In addition, when dispatch needed the number of received tokens, it waited on the CPU for the GPU to return that information.

The key point is that entering C++ does not automatically release the GIL. While these calls held the lock, the Python thread responsible for Mooncake Transfer in the same process could not acquire the GIL promptly. Scheduling and submission of transfer tasks were therefore delayed, reducing the opportunity for KV Transfer to overlap with subsequent computation. Even if the underlying transfer mechanism supports asynchronous execution, blocked submission at a higher layer can prevent the expected parallelism from materializing fully.

The source code also provided a direct comparison: `internode_dispatch` in the same version already explicitly released the GIL, with a comment explaining that this was intended to avoid blocking KV Transfer in other threads while the CPU waited. This further supported the agent's assessment of the intra-node path.

The key fix was to release the GIL during the relevant C++ execution intervals, allowing the Mooncake Transfer Python thread to advance its tasks promptly. The fix had to be validated against both the timeline and the original performance constraint. The former checked whether scheduling and transfers gained the opportunity to overlap with computation; the latter determined whether this change improved actual serving performance. Under the same test conditions, the performance gap between Prefill + KV Transfer and Prefill alone fell below 1% after the fix.



> ***Figure 3: Discovering and fixing the KV Transfer concurrency bottleneck.***

In this case, the performance constraint first turned "not meeting expectations" into a

clearly defined test discrepancy. The timeline then narrowed the investigation to concurrency between two components. Finally, code analysis pinpointed the scope over which the GIL was held. Dense feedback connected end-to-end performance, cross-layer runtime behavior, and a specific implementation detail, providing evidence for every step of the agent's analysis.

Performance Feedback: Building on Existing Kernel Optimization Expertise and Turning It into System-Level Gains

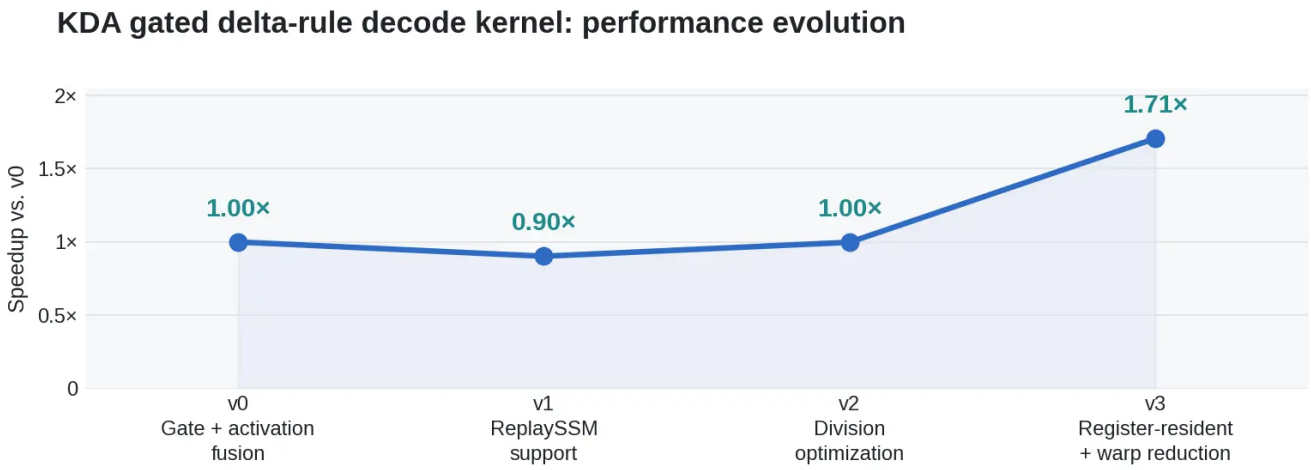
Kernel optimization must answer two questions: how to determine whether an optimization works, and where to find promising optimization directions.

First, kernel performance must be evaluated within the inference engine's actual execution environment. For example, giving a compute kernel more resources may shorten its own execution time while leaving fewer resources for the KV Transfer kernel, ultimately slowing the overall pipeline. The agent must therefore look beyond individual kernel latency and establish the right optimization objective in the context of the target workload, resource constraints, task overlap, and end-to-end gains.

Second, a great deal of optimization expertise is embedded in handwritten kernels from projects such as SGLang, Flash Linear Attention, and DeepGEMM. The agent needs to extract optimization techniques and their applicable conditions from this code, develop candidate approaches for the current kernel and target hardware, and validate their actual effects through experiments. Existing code supplies optimization directions; system feedback determines whether those optimizations hold up in practice.

We had the GLM-5.3-powered Infra Agent learn optimization techniques from existing kernels across different codebases, programming languages, and hardware platforms. Through incremental and ablation experiments, it distilled these techniques into "optimization skeletons" containing applicability conditions, transformation methods, resource constraints, and validation evidence. For a new kernel, the agent starts from these skeletons, then uses profiling and layered testing to reassess tiling, memory access, and resource allocation strategies. Validated changes and their applicability conditions feed

back into the skeleton library. Engineers primarily define objectives and constraints, and review critical changes involving numerical semantics, concurrency behavior, and production risk.



> ***Figure 4: The performance evolution of a representative KDA Decode kernel, from its baseline implementation to the production version.***

Figure 4 shows the performance evolution of a representative KDA Decode kernel. Introducing ReplaySSM to trade compute for memory caused the first increase in kernel execution time, from v0 to v1. The agent's division optimization then reduced v1's execution time by 9.6%. Next, after receiving feedback that computation was the primary bottleneck, the Infra Agent discovered that the original implementation tiled along the V dimension, causing the same FP32 normalization and gating computations to be repeated four times. It merged these tiles into a single thread block, kept the shared intermediate results register-resident, and replaced the redundant per-tile computation with a single warp-level reduction.By sacrificing some parallelism, it eliminated redundant computation at the source and achieved a 1.71× speedup over v2.

In this case, the GLM-5.3-powered Infra Agent extracted optimization expertise from existing implementations and applied it to the kernels running its own inference. Starting from the skeletons, it tuned each component, used layered validation to decide which changes to retain, and relied on end-to-end performance to establish their practical value. Validated insights then flowed back into the skeleton library. The model thus participated in optimizing its own inference system, while the experience accumulated with each deployment reduced the engineering effort required for the next round of optimization.

Let Feedback Drive Action, and Experiments Test Hypotheses

Together, these three cases show that the value of feedback lies not in its volume, but in whether it helps the agent answer the question at hand. Large amounts of unstructured logs can obscure critical signals. Profiling with incomplete coverage can lead to incorrect attribution. An optimization that works in a microbenchmark may not translate into end-to-end gains. Building a feedback environment therefore requires more than supplying tests, logs, and performance data. It also requires specifying what judgments each type of observation can support, what its limits are, and which conclusions must be confirmed through further experiments.

Engineers have three critical responsibilities in this process: defining optimization objectives and system constraints, building a feedback environment the agent can use directly, and reviewing critical changes involving system architecture, asynchronous concurrency, and production risk. Within this framework, the agent proposes hypotheses, implements changes, and runs experiments, then uses feedback to retain, revise, or reject its current approach. Correctness, stability, and end-to-end performance jointly define the final acceptance criteria.

Looking back at the GLM-5.3-Flash launch, numerical accuracy bugs in kernels, concurrency issues spanning the Python/C++ boundary, and performance optimization of critical kernels each represented engineering challenges at different layers. Through local tests, cross-layer observations, and layered benchmarks, initially ambiguous anomalies were progressively turned into testable engineering hypotheses. Complex systems problems were broken down into a series of observable, experimentally testable iterations with attributable outcomes. It was within this feedback loop that the agent's coding and reasoning capabilities translated into verifiable engineering progress.

The GLM-5.3-powered Infra Agent helped build the inference infrastructure. The system, jointly optimized by engineers and the agent, in turn enabled GLM-5.3-Flash to serve users reliably. **The model optimizes the system; the system runs the model.** This effort demonstrates that shortening systems engineering cycles requires more than stronger model capabilities. It requires an agent engineering loop that enables the model to

continuously receive feedback, test its judgments, and revise its actions.

Of course, we have not yet reached recursive self-improvement. Choosing objectives, setting boundaries, and assessing risk remain human responsibilities. We believe humans should continue to hold that line for a long time to come. But the numbers, two weeks, threefold throughput, and 100,000 accelerators, tell us that progress at this boundary will not slow down simply because we want it to.



Legal

Privacy Policy

Terms of Service

© 2026 Z.ai Inc.