Context Engineering for AI Agents: Lessons from Building Manus

Saturday, July 19 Tech

2025/7/18 --Yichao 'Peak' Ji

At the very beginning of the Manus project, my team and I faced a key decision: should we train an end-to-end agentic model using open-source foundations, or build an agent on top of the in-context learning abilities of frontier models?

Back in my first decade in NLP, we didn't have the luxury of that choice. In the distant days of BERT (yes, it's been seven years), models had to be fine-tuned—and evaluated—before they could transfer to a new task. That process often took weeks per iteration, even though the models were tiny compared to today's LLMs. For fast-moving applications, especially pre-PMF, such **slow feedback loops** are a deal-breaker. That was a bitter lesson from my last startup, where I trained models from scratch for open information extraction and semantic search. Then came GPT-3 and Flan-T5, and my in-house models became irrelevant overnight. Ironically, those same models marked the beginning of in-context learning—and a whole new path forward.

That hard-earned lesson made the choice clear: **Manus would bet on context engineering**. This allows us to ship improvements in hours instead of weeks, and kept our product orthogonal to the underlying models: **If model progress is the rising tide, we want Manus to be the boat**, not the pillar stuck to the seabed.

Still, context engineering turned out to be anything but straightforward. It's an experimental science—and we've rebuilt our agent framework four times, each time after discovering a better way to shape context. We affectionately refer to this manual process of architecture searching, prompt fiddling, and empirical guesswork as "**Stochastic *Graduate* Descent**". It's not elegant, but it works.

This post shares the local optima we arrived at through our own "SGD". If you're building your own AI agent, I hope these principles help you converge faster.

Design Around the KV-Cache

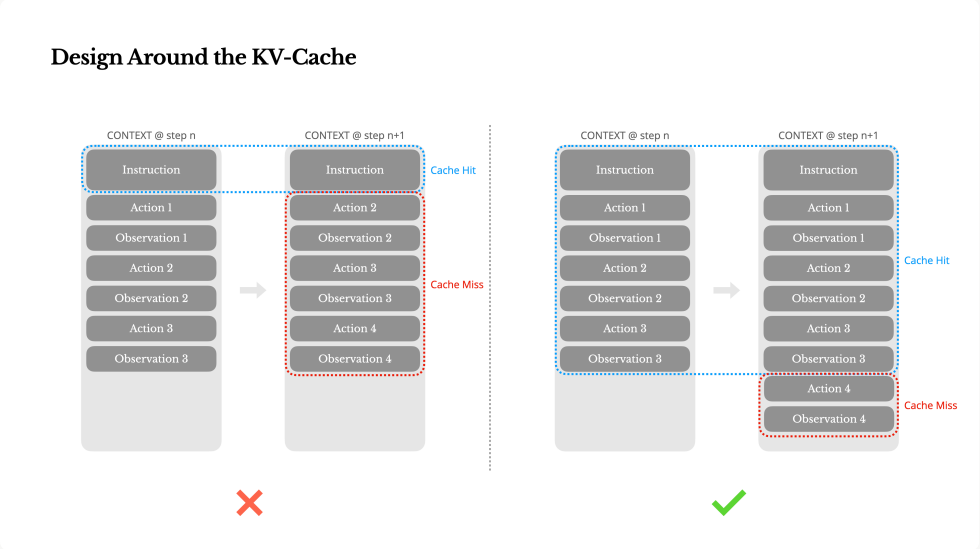
If I had to choose just one metric, I'd argue that the **KV-cache hit rate** is the single most important metric for a production-stage AI agent. It directly affects both latency and cost. To understand why, let's look at how a typical agent operates:

After receiving a user input, the agent proceeds through a chain of tool uses to complete the task. In each iteration, the model selects an **action** from a predefined action space based on the current context. That action is then

executed in the **environment** (e.g., Manus's virtual machine sandbox) to produce an **observation**. The action and observation are appended to the context, forming the input for the next iteration. This loop continues until the task is complete.

As you can imagine, the context grows with every step, while the output—usually a structured function call—remains relatively short. This makes the ratio between **prefilling** and **decoding** highly skewed in agents compared to chatbots. In Manus, for example, the average input-to-output token ratio is around **100:1**.

Fortunately, contexts with identical prefixes can take advantage of KV-cache, which drastically reduces **time-to-first-token (TTFT)** and inference cost—whether you're using a self-hosted model or calling an inference API. And we're not talking about small savings: with Claude Sonnet, for instance, cached input tokens cost **0.30 USD/MTok**, while uncached ones cost **3 USD/MTok**—a 10x difference.



From a context engineering perspective, improving KV-cache hit rate involves a few key practices:

1. **Keep your prompt prefix stable.** Due to the autoregressive nature of LLMs, even a single-token difference can invalidate the cache from that token onward. A common mistake is including a timestamp—especially one precise to the second—at the beginning of the system prompt. Sure, it lets the model tell you the current time, but it also kills your cache hit rate.
2. **Make your context append-only.** Avoid modifying previous actions or observations. Ensure your serialization is deterministic. Many programming languages and libraries don't guarantee stable key ordering when serializing JSON objects, which can silently break the cache.
3. **Mark cache breakpoints explicitly when needed.** Some model providers or inference frameworks don't support automatic incremental prefix caching, and instead require manual insertion of cache breakpoints in the context. When assigning these, account for potential cache expiration and at minimum, ensure the breakpoint includes the end of the system prompt.

Additionally, if you're self-hosting models using frameworks like vLLM, make sure prefix/prompt caching is enabled, and that you're using techniques like **session IDs** to route requests consistently across distributed workers.

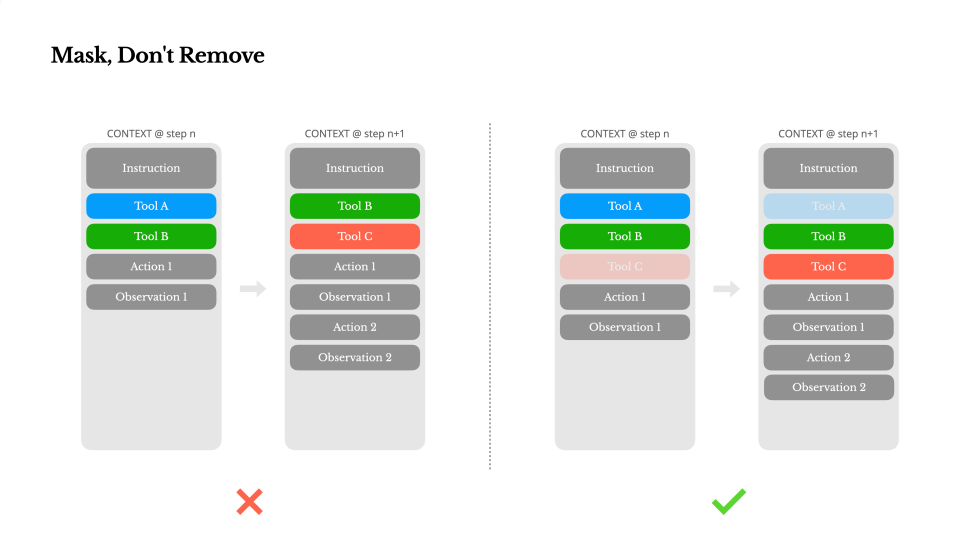
Mask, Don't Remove

As your agent takes on more capabilities, its action space naturally grows more complex—in plain terms, the **number of tools** explodes. The recent popularity of MCP only adds fuel to the fire. If you allow user-configurable tools, trust me: someone will inevitably plug hundreds of mysterious tools into your carefully curated action space. As a result, the model is more likely to select the wrong action or take an inefficient path. In short, your heavily armed agent gets dumber.

A natural reaction is to design a dynamic action space—perhaps loading tools on demand using something RAG-like. We tried that in Manus too. But our experiments suggest a clear rule: unless absolutely necessary, **avoid dynamically adding or removing tools mid-iteration**. There are two main reasons for this:

1. In most LLMs, tool definitions live near the front of the context after serialization, typically before or after the system prompt. So any change will invalidate the KV-cache for all subsequent actions and observations.
2. When previous actions and observations still refer to tools that are no longer defined in the current context, the model gets confused. Without constrained decoding, this often leads to **schema violations or hallucinated actions**.

To solve this while still improving action selection, Manus uses a context-aware state machine to manage tool availability. Rather than removing tools, it **masks the token logits** during decoding to prevent (or enforce) the selection of certain actions based on the current context.



In practice, most model providers and inference frameworks support some form of **response prefill**, which allows you to constrain the action space *without*

modifying the tool definitions. There are generally three modes of function calling (we'll use the Hermes format from NousResearch as an example):

- **Auto** – The model may choose to call a function or not. Implemented by prefilling only the reply prefix: `<|im_start|>assistant`
- **Required** – The model must call a function, but the choice is unconstrained. Implemented by prefilling up to tool call token: `<|im_start|>assistant<tool_call>`
- **Specified** – The model must call a function **from a specific subset**. Implemented by prefilling up to the beginning of the function name: `<|im_start|>assistant<tool_call>{"name": "browser_`

Using this, we constrain action selection by masking token logits directly. For example, when the user provides a new input, Manus must reply immediately instead of taking an action. We've also deliberately designed action names with consistent prefixes—e.g., all browser-related tools start with `browser_`, and command-line tools with `shell_`. This allows us to easily enforce that the agent only chooses from a certain group of tools at a given state **without using stateful logits processors**.

These designs help ensure that the Manus agent loop remains stable—even under a model-driven architecture.

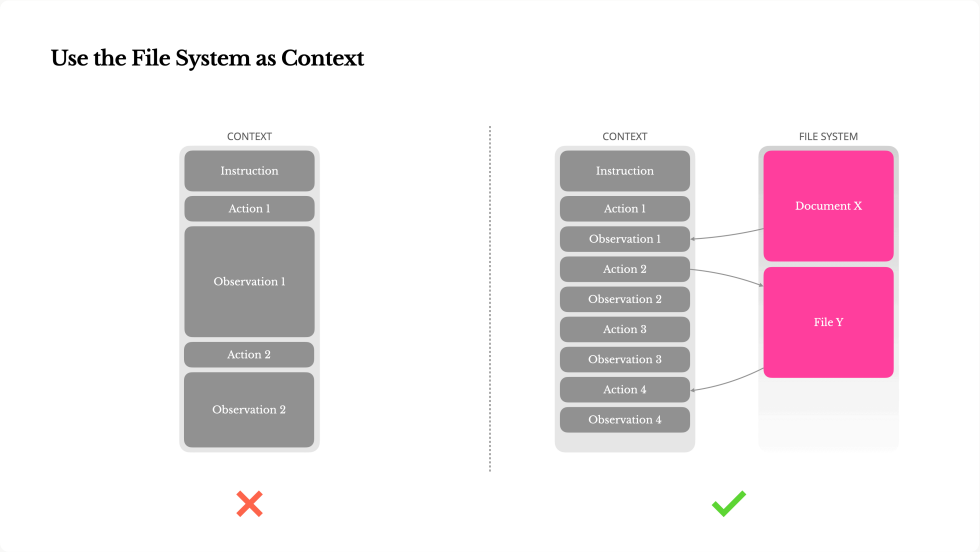
Use the File System as Context

Modern frontier LLMs now offer context windows of 128K tokens or more. But in real-world agentic scenarios, that's often not enough, and sometimes even a liability. There are three common pain points:

1. **Observations can be huge**, especially when agents interact with unstructured data like web pages or PDFs. It's easy to blow past the context limit.
2. **Model performance tends to degrade** beyond a certain context length, even if the window technically supports it.
3. **Long inputs are expensive**, even with prefix caching. You're still paying to transmit and prefill every token.

To deal with this, many agent systems implement context truncation or compression strategies. But overly aggressive compression inevitably leads to information loss. The problem is fundamental: an agent, by nature, must predict the next action based on all prior state—and you **can't** reliably predict which observation might become critical ten steps later. From a logical standpoint, any irreversible compression carries risk.

That's why we treat the **file system as the ultimate context** in Manus: unlimited in size, persistent by nature, and directly operable by the agent itself. The model learns to write to and read from files on demand—using the file system not just as storage, but as structured, externalized memory.



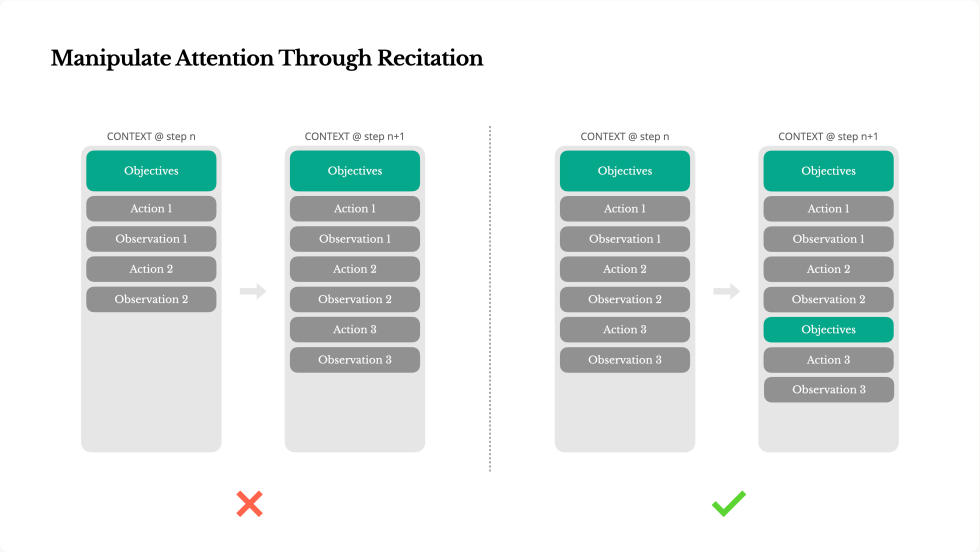
Our compression strategies are always designed to be **restorable**. For instance, the content of a web page can be dropped from the context as long as the URL is preserved, and a document's contents can be omitted if its path remains available in the sandbox. This allows Manus to shrink context length without permanently losing information.

While developing this feature, I found myself imagining what it would take for a **State Space Model (SSM)** to work effectively in an agentic setting. Unlike Transformers, SSMs lack full attention and struggle with long-range backward dependencies. But if they could master file-based memory—externalizing long-term state instead of holding it in context—then their speed and efficiency might unlock a new class of agents. Agentic SSMs could be the real successors to Neural Turing Machines.

Manipulate Attention Through Recitation

If you've worked with Manus, you've probably noticed something curious: when handling complex tasks, it tends to create a **todo.md** file—and update it step-by-step as the task progresses, checking off completed items.

That's not just cute behavior—it's a deliberate mechanism to **manipulate attention**.



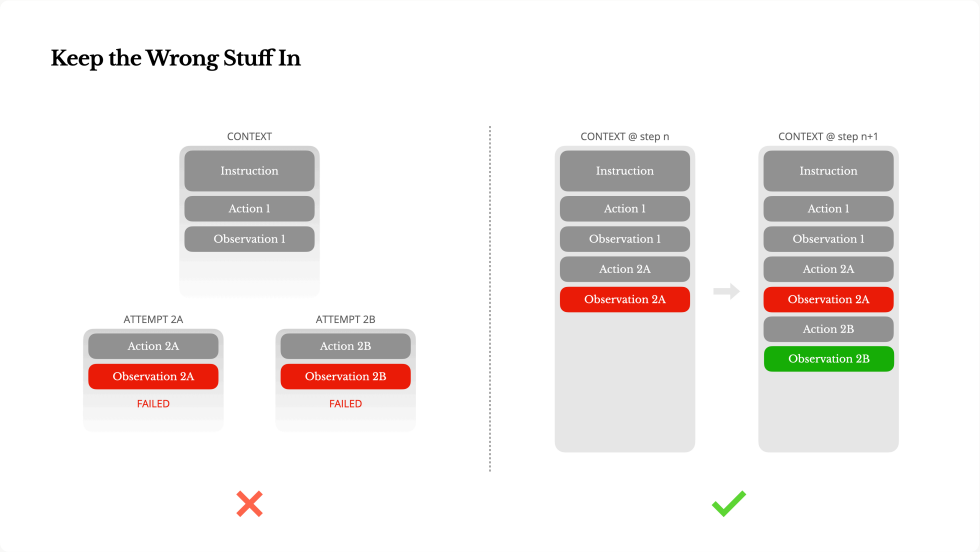
A typical task in Manus requires around **50 tool calls** on average. That's a long loop—and since Manus relies on LLMs for decision-making, it's vulnerable to drifting off-topic or forgetting earlier goals, especially in long contexts or complicated tasks.

By constantly rewriting the todo list, Manus is **reciting its objectives into the end of the context**. This pushes the global plan into the model's recent attention span, avoiding "**lost-in-the-middle**" issues and reducing goal misalignment. In effect, it's using natural language to bias its own focus toward the task objective—without needing special architectural changes.

Keep the Wrong Stuff In

Agents make mistakes. That's not a bug—it's reality. Language models hallucinate, environments return errors, external tools misbehave, and unexpected edge cases show up all the time. In multi-step tasks, failure is not the exception; it's part of the loop.

And yet, a common impulse is to hide these errors: clean up the trace, retry the action, or reset the model's state and leave it to the magical "temperature". That feels safer, more controlled. But it comes at a cost: **Erasing failure removes evidence**. And without evidence, the model can't adapt.



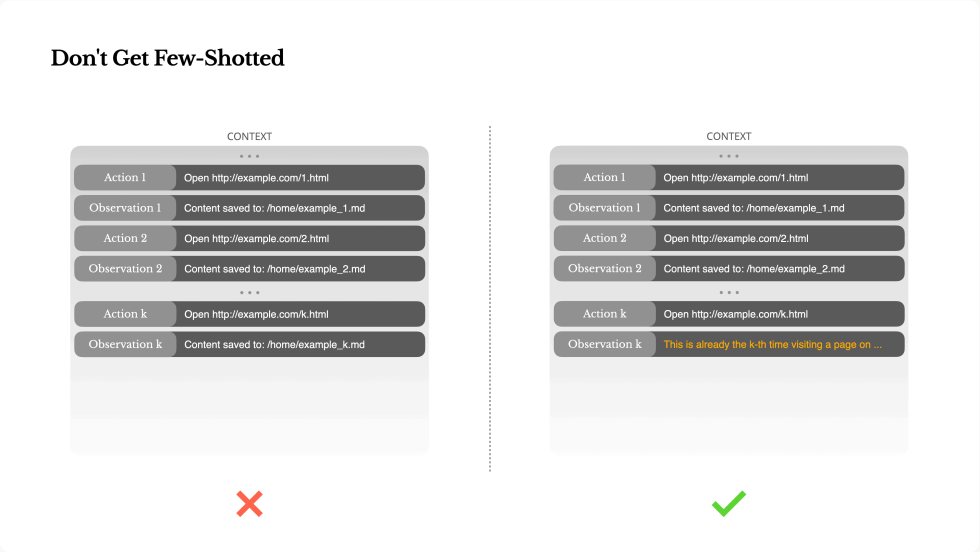
In our experience, one of the most effective ways to improve agent behavior is deceptively simple: **leave the wrong turns in the context**. When the model sees a failed action—and the resulting observation or stack trace—it implicitly updates its internal beliefs. This shifts its prior away from similar actions, reducing the chance of repeating the same mistake.
In fact, we believe **error recovery** is one of the clearest indicators of true agentic behavior. Yet it's still underrepresented in most academic work and public benchmarks, which often focus on task success under ideal conditions.

Don't Get Few-Shotted

Few-shot prompting is a common technique for improving LLM outputs. But in agent systems, it can backfire in subtle ways.

Language models are excellent mimics; they **imitate the pattern of behavior** in the context. If your context is full of similar past action-observation pairs, the model will tend to follow that pattern, even when it's no longer optimal.

This can be dangerous in tasks that involve repetitive decisions or actions. For example, when using Manus to help review a batch of 20 resumes, the agent often falls into a rhythm—repeating similar actions simply because that's what it sees in the context. This leads to drift, overgeneralization, or sometimes hallucination.



The fix is to **increase diversity**. Manus introduces small amounts of structured variation in actions and observations—different serialization templates, alternate phrasing, minor noise in order or formatting. This controlled randomness helps break the pattern and tweaks the model's attention.
In other words, **don't few-shot yourself into a rut**. The more uniform your context, the more brittle your agent becomes.

Conclusion

Context engineering is still an emerging science—but for agent systems, it's already essential. Models may be getting stronger, faster, and cheaper, but no amount of raw capability replaces the need for memory, environment, and feedback. How you shape the context ultimately defines how your agent behaves: how fast it runs, how well it recovers, and how far it scales.

At Manus, we've learned these lessons through repeated rewrites, dead ends, and **real-world testing across millions of users**. None of what we've shared here is universal truth—but these are the patterns that worked for us. If they help you avoid even one painful iteration, then this post did its job.

The agentic future will be built one context at a time. Engineer them well.

Less structure, more intelligence.