LLM Agents & Context: A Warrior's Guide to Navigating the Dungeon

ZACHARY HUANG
JUL 04, 2025

17 Sl

> *Your agent has a legendary sword and a powerful spellbook. But what good are weapo your warrior is lost in a sprawling dungeon, unable to remember which rooms are clear and which hold treasure? In this guide, you'll learn the three master navigation technic of agent memory—the Scrying Spell, the Grand Strategy, and the Cautious Explorer's I It's time to teach ou*

1. Introducti ec

In our previous adve gr and forged our warrior in [LLM Agents are simply Graph — Tutorial For Dummie](https://pocketflow.substack.com)

Then, we equipped it with a deadly arsenal of actions in LLM Agents & Their Arse A Beginner's Guide. But now, our warrior faces its greatest challenge yet: the environment itself.

The agent's battle isn't on an open field; it's in a dark, complex dungeon—a large codebase, a multi-step research task, or a complex dataset. Here, the biggest dang isn't the monsters (the individual tasks), but getting lost, forgetting where you've b and losing sight of the treasure at the end.

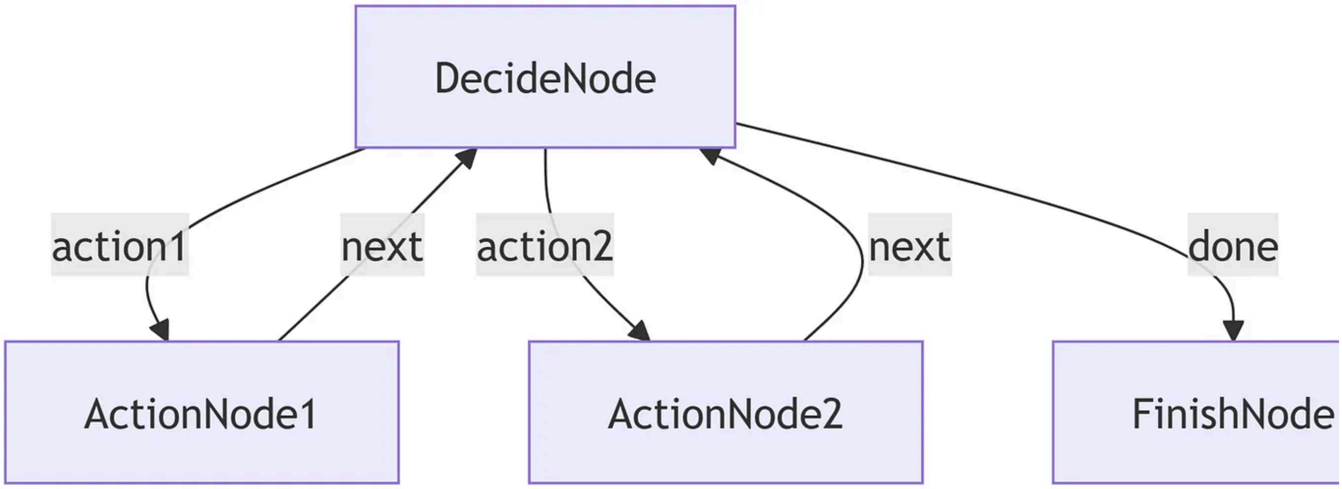
This brings us to the most critical, and often botched, aspect of agent design: cont management. Think of it as the warrior's **Cognitive Backpack**. In our PocketFlow framework, this is the simple `shared` dictionary that each Node reads from and w to. The naive approach is to stuff everything inside. Imagine loading the entire dungeon map, every monster's stat block, every rumor of treasure, and the history the last three adventurers who failed into the warrior's backpack right at the start. They'd collapse under the weight before they even took their first step.

Smart context management isn't about giving the agent *more* memory; it's about g it the **right** memory at the **right** time. In this guide, we'll stop treating our agent's memory like a junk drawer and start treating it like a high-tech utility belt. We wi learn three master-level navigation techniques to keep the backpack light, the war agile, and the path to victory clear.

2. The Warrior's Battle Plan: A Quick Recap

Before we teach our the battle plan. As we lea e, relentless loop: `Asse`

In the **PocketFlow** fr



The `DecideNode` is the warrior's brain—the battle tactician. It assesses the situa and chooses the next move. The `ActionNodes` are the specialist soldiers who car out a specific command. And the loop back is the report, bringing new informatio from the battlefield back to the tactician.

But how does the `DecideNode` *actually* think? How does it "assess" the situation? This isn't magic; it's a carefully crafted prompt. The node's entire worldview come from the `shared` dictionary, which is formatted and injected directly into its brai

The prompt inside our `DecideNode` looks something like this:

```
### CURRENT SITUATION
You are a research assistant. Here is what you know right now:
{context_from_shared_store}

### AVAILABLE ACTIONS
[1] search_web(query: str)
  Description: Search for new information online.
[2] write_file(filename: str, content: str)
  Description: Save information to a file.
[3] finish_task(r
  Description: Co

## YOUR NEXT MOVE
Based **only** on                                                ion
to take next and
```

The content of our `shared` dictionary—our cognitive backpack—is dropped direc into the `{context_from_shared_store}` placeholder. This is the **only thing t LLM sees**. Its entire universe of knowledge for making a decision is contained wit that block.

This reveals a critical truth: **the quality of the agent's decisions is 100% dependen the quality of the information in the `shared` store.** This simple dictionary is the important part of the agent's "brain." So, the central question becomes: what is th right way to manage it?

3. The Overwhelmed Warrior: Why Dumping A Context Fails

Imagine our warrior at the dungeon entrance. We, as the benevolent master, decid "help" by giving them *everything*. We cram the entire 500-page dungeon history, ev blueprint, every monster's family tree, and a transcript of every conversation ever about the dungeon into their backpack. "Good luck!" we say, as the warrior stumb forward, unable to even lift their sword under the crushing weight.

They enter the first room, which has a simple pressure plate on the floor. To solve they need to find the small, one-ounce stone they picked up just a moment ago. Bu find it, they have to rummage through the 200-pound bag of useless junk we gave them. They get distracted by a map of a different dungeon wing, start reading abo the goblin king's third cousin, and forget about the pressure plate entirely. They a paralyzed by information overload.

This is *exactly* what happens when you dump your entire `shared` history into an LLM's prompt on every turn. A cluttered backpack doesn't create a genius warrio creates an ineffective

- **Diluted Attentic** a massive context, n a of irrelevant data m important fact ("the user just asked to search for *this*") when it's buried in ten pages of previous search results. This is often called the "lost in the middle" problem, where information in the center of a large prompt is frequently igno

- **Sky-High Costs & Latency:** Every token in your prompt costs money and processing time. A cluttered backpack slows your warrior to a crawl and empt your coin purse. An agent that sends a 100,000-token context on every loop is only breathtakingly expensive but also painfully slow, making any real-time interaction impossible.
- **Increased Hallucinations:** When an LLM is given too much loosely related information, it starts to "cross the wires." It might grab a detail from an early, now-irrelevant step and incorrectly apply it to the current situation. It's the equivalent of our warrior trying to use a recipe for a health potion to disarm a magical trap—a confident but catastrophically wrong decision.

The takeaway is simple: **a bigger context does not equal a smarter agent.** Our goa not to build the biggest backpack, but the most efficient one. We need to stop beir hoarders and start being strategists, ensuring our warrior carries only what they n for the immediate fight.

4. Forging the Cognitive Backpack: Three Master Navigation Techniques

If stuffing everything into the backpack is the path to failure, what is the path to victory? The answer lies in transforming the backpack from a static, heavy burder a dynamic, intelligent system. A master warrior doesn't carry every tool for every possible situation. They carry a few versatile tools that allow them to adapt to *any* situation.

Here are the three master techniques for forging your agent's cognitive backpack.

4.1. Technique Demand)

The Metaphor: Our In of a map, they carry a magical "scrying orb." When they reach a fork in the pa don't guess. They hold up the orb and ask, "What lies down the left corridor?" The shows them a brief vision of the next room. They now have a small, relevant piece information. They add this "vision" to their mental map and then use the orb agai scout the right corridor. They build their map piece by piece, only gathering the information they need, precisely when they need it.

The Pattern: This is an **exploratory, on-demand** context strategy. Instead of feedi the agent a massive, pre-filled context, you give it the tools to build its own contex The agent's primary actions are not to solve the final problem, but to ask question about the environment. The `shared` store starts small and grows incrementally, populated only by the answers to the agent's own, self-directed inquiries. It's a **pu** **model** of information gathering, not a push model.

Concrete Example: The AI Data Analyst

Imagine an agent tasked with: "Find the total revenue from our top 5 customers la quarter."

A naive approach would be to dump the entire database schema into the initial prompt. The Scrying Spell approach is far more elegant:

- **Loop 1: Assess the Landscape**
  - `shared` **store:** `{"goal": "Find revenue of top 5 customers l quarter"}`
  - `DecideNode` **thinks:** "I have no idea what tables are in the database. I ne look."
  - **Action:** It calls `list_tables()`.
  - **Update** `shared`**:** The list of tables (`['customers', 'orders', 'products`
- **Loop 2: Zoom in**
  - `shared` **stor**
  - `DecideNod` ee know what columns are in the 'orders' table to find revenue and date:

  - **Action:** It calls `get_table_schema(table_name='orders')`.
  - **Update `shared`:** The schema for the `orders` table (`{'order_id': 'IN` `'customer_id': 'INT', 'order_date': 'DATE',` `'total_amount': 'DECIMAL'}`) is added.
- **Loop 3: Formulate the Attack**
  - **`shared` store:** Contains the goal, table names, and the `orders` schema.
  - **`DecideNode` thinks:** "Perfect. Now I have everything I need to write a pr SQL query to get the answer."
  - **Action:** It calls `execute_sql(query="SELECT ...")`.

Notice the difference. The context was never overwhelming. It was built intelliger step-by-step, by the agent itself. We didn't give it a map; we gave it a scrying orb a trusted it to find its own way.

4.2. Technique #2: The Grand Strategy (Map-Reduce)

The Metaphor: The warrior now faces not a dungeon, but an entire fortress. Tryin map it room by room would take forever. Instead, they first send a hawk into the s The hawk returns with a high-level sketch of the fortress: the barracks, the keep, a the treasury (`Map` phase). The warrior decides to tackle the keep first. They leave tl main map behind and take *only the detailed blueprint of the keep* with them (`Subtas Execution`). After conquering the keep and taking its treasure, they return to the starting point, drop off the loot, and then take *only the blueprint for the treasury* for t next mission. Once all wings are cleared, they have all the treasure in one place (`Reduce` phase).

The Pattern: This is the classic **divide-and-conquer** strategy, perfectly suited for t that are too large for

1. **Map:** A high-lev ow into smaller, ind

2. **Subtask Execution (in parallel or sequence):** For each subtask, the agent is ru with a **hermetically sealed context**. It is *only* given the information relevant to that one subtask, completely ignorant of the others. This keeps the context sm and focused.
3. **Reduce:** A final step where the results from all the independent subtasks are gathered and synthesized into a final, coherent output.

Concrete Example: The AI Codebase Knowledge Builder

This is the exact strategy used in our Codebase Knowledge Builder project, which turns an entire GitHub repository into a friendly tutorial. Stuffing a whole codeba into a prompt is impossible. Here's how the Grand Strategy makes it work:

- **Map Phase: The Hawk's View**
    - The `IdentifyAbstractions` and `OrderChapters` nodes act as the ha They scan the file structure and code at a high level (without reading every line) to create a plan.
    - `shared` **store output:** A list of core concepts and a recommended chapter order, like: `["1. BaseNode", "2. Flow", "3. SharedMemory"]`.
- **Subtask Execution Phase: Conquering the Keep**
    - The `WriteChapters` `BatchNode` in PocketFlow executes this phase perfectly. It iterates through the plan.
    - **For Chapter 1** (**"BaseNode"**): Its `prep` method intelligently scans the `shared['codebase']` and gathers *only the code files relevant to* `BaseNod` then calls the LLM with a tiny, focused prompt: "Write a chapter on BaseN using *only this specific code*." The LLM is completely unaware of the code f `Flow` or `SharedMemory`, preventing confusion.
    - **For Chapter** ete *different, isol*
- **Reduce Phase: C**

- The `CombineTutorial` node acts as the final organizer. It takes all the individually written chapter outputs from the `shared` store (which now contains the completed text for each chapter) and assembles them into a single, polished tutorial document with a table of contents and navigation

Without this strategy, the task would be impossible. With it, we can conquer a for of any size, one well-planned, focused assault at a time.

4.3. Technique #3: The Cautious Explorer (Backtrackin with Verification)

The Metaphor: Our warrior enters a room with a suspicious-looking lever. A reck warrior pulls it and hopes for the best. A cautious warrior pulls it (`Apply Change` but keeps one foot in the doorway, ready to jump back. They listen intently. Do the hear the satisfying *click* of a hidden door opening, or the terrifying *rumble* of a ceil collapse? (`Verify`). If it's the rumble, they immediately let go of the lever, which springs back into place (`Revert Change`), and they proceed to look for a differen solution. They are allowed to make mistakes, as long as they can observe the consequences and undo them.

The Pattern: This is a **trial-and-error with a safety net** strategy, essential for agen that modify their environment, like coding agents. The flow is cyclical:

1. **Apply Change:** The agent performs an action that alters the state (e.g., writes file).
2. **Verify:** A special node captures the *consequence* of that action. This isn't just t action's output; it's an observation of the new world state (e.g., a `git diff`, l output, or a failing test result).
3. **Decide & Revert** he LLM is then pro sfu not, should we re ial action restores th

Concrete Example: The AI Coding Agent

Imagine an agent tasked with fixing a bug in a Python file.

- **Loop 1: The Attempt**
  - `DecideNode` **thinks:** "Based on the bug report, I think the error is on line I will change `x > 5` to `x >= 5`."
  - **Action:** The `write_file` node modifies the Python file.
- **Loop 2: The Verification**
  - The `post` hook of the `write_file` node is a special `verify_code` func It doesn't just return "success." It runs the project's linter and unit tests.
  - **Verification Result:** The linter passes, but a unit test now fails with a new `AssertionError`. This full error message is the output.
  - **Update `shared`:** The `shared` store is updated with: `{"last_attempt" "Changed line 52 to x >= 5", "verification_log": "Lint OK. Tests: FAILED - AssertionError: Test case for x=5 failed."}`
- **Loop 3: The Reassessment**
  - The `DecideNode` sees the previous attempt and the failed test.
  - `DecideNode` **thinks:** "My last change was wrong. It broke a different test case. I must revert the change and try a different approach. The logic mus more complex than a simple comparison."
  - **Action:** It calls `revert_last_change()`, followed by a new `write_fi` with a completely different solution.

This loop of `Apply -> Verify -> Revert` allows the agent to safely explore t solution space without permanently breaking things. It can make hypotheses, test them, and backtrack to complex problems.

5. Conclusi Fc Memorizes

And so, the secrets of the dungeon are yours. We've moved beyond simply forging powerful warrior; we've now taught it how to navigate the most treacherous and complex environments. You now understand that an agent's true intelligence isn't measured by the size of its brain (the LLM) or the sharpness of its weapons (the ac space), but by the wisdom of its **context management**.

A cluttered, unmanaged `shared` store—our warrior's cognitive backpack—is a re for a slow, confused, and expensive agent. But a well-managed one is the key to a focused, efficient, and surprisingly clever digital warrior.

You've learned the three master navigation techniques, transforming you from a n agent blacksmith into a grand strategist:

- **The Scrying Spell (Context On-Demand):** The ultimate tool for exploration, allowing your agent to build its own map of the unknown, piece by piece, with ever getting overwhelmed.
- **The Grand Strategy (Map-Reduce):** Your weapon against overwhelming complexity, enabling your agent to conquer massive challenges like entire codebases by breaking them down into small, focused, and manageable battle
- **The Cautious Explorer (Backtracking with Verification):** The safety net that empowers your agent to make bold moves and try new things, secure in the knowledge that it can observe the consequences and gracefully retreat from a dead ends.

The next time you build an agent, don't just ask, "What can it do?" Instead, ask, "H will it think? How will it manage its focus?" By thoughtfully designing your agent cognitive backpack, you are no longer just coding a workflow; you are imparting wisdom. You are creating a smart warrior that doesn't just memorize the map, but navigates the dungeo

Ready to forge your *th*
context strategies, an *Ch*

out PocketFlow on GitHub and start building smarter today!

Thanks for reading Pocket Flow! Subscribe for free to receive new posts and support my work.

17 Likes

← Previous

Next

Discussion about this post

Comments Restacks

Write a comment...