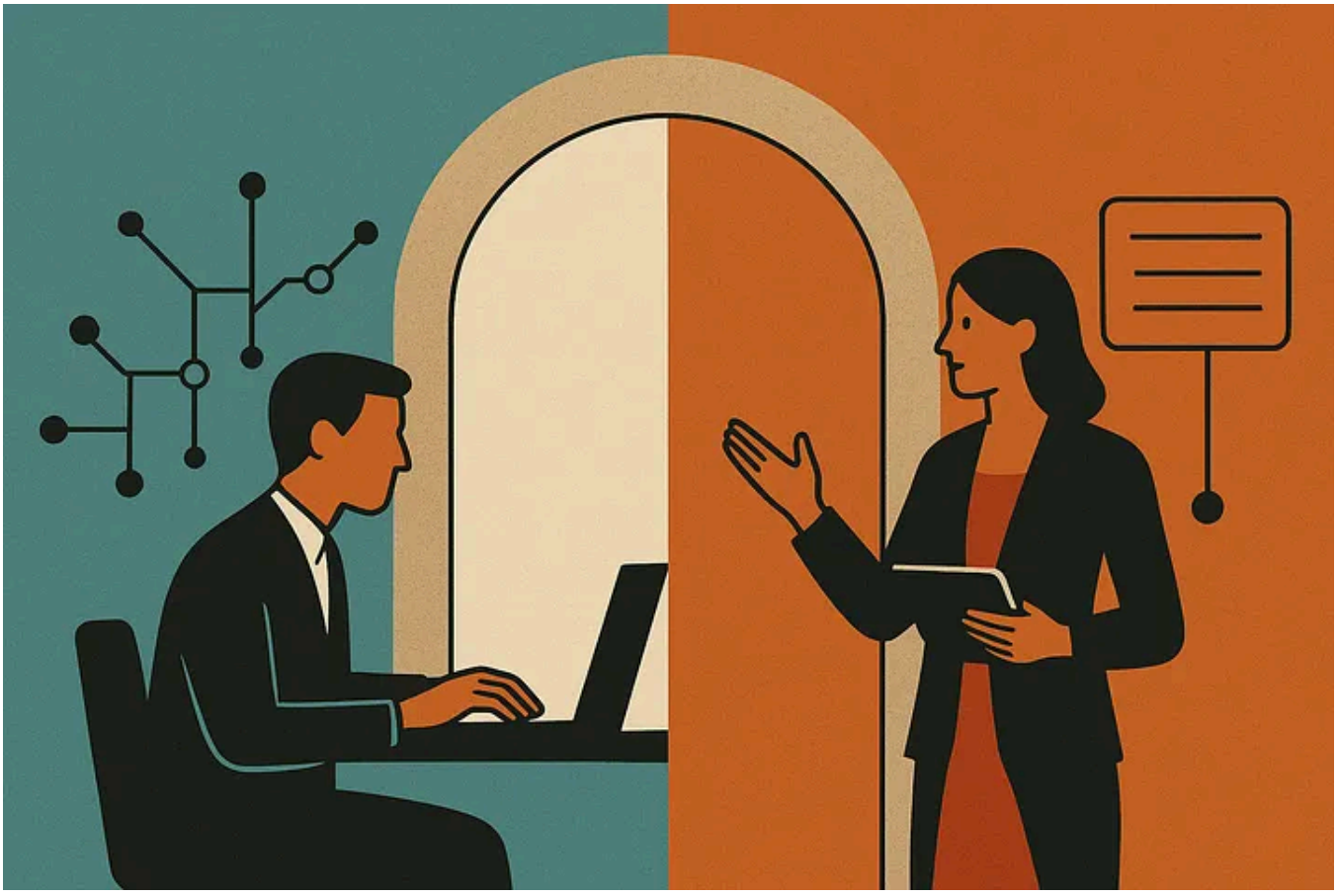
Does Your Agent Need an Agent?

8 min read · 19 hours ago

Bob Dickinson Follow

The Architectural Divide Between Direct LLM API Executors and Cognitive Orchestrators



When directly using Large Language Model (LLM) APIs to build intelligent applications, developers often observe a performance gap: proprietary chat applications (e.g., ChatGPT, Claude Code) consistently produce higher quality results, often demonstrating superior apparent reasoning, even when using the

same underlying models that their own apps or agents are using. What is the magic secret sauce that these applications are deploying to get such amazing results?

Consider something like Claude Code "Skills". They are pretty amazing, but when you dig into how they work, the miracle is not so much that Claude Code knows HOW to use them (they're just prompt information really), it's that it knows WHEN to use them.

The Observed Performance Gap

The core function of an agent application is to maintain conversation history and select relevant tools and other context to pass to the LLM. A common pattern in initial agent development involves creating a thin wrapper around a vendor LLM chat API, where the entire conversation history, full list of available tools, and sometimes all system rules, are submitted on every single API call to the model.

This architecture, which we'll call the **Passive Context Aggregator**, inevitably leads to two significant problems: excessive cost, and sometimes outright request failure, due to ballooning context windows, and more importantly, a marked degradation in the quality of reasoning. The paradox is that the *same* powerful foundation model can appear significantly less capable when deployed via the PCA pattern than when accessed through a proprietary application.

The solution employed by sophisticated platforms is a layered approach where an intelligence layer, the **Control Plane**, manages the conversation state, resources, and context. This system, which we define as the **Cognitive Orchestrator**, delegates key decisions and workflows to specialized processes or sub-agents, ensuring the workhorse LLM receives the most optimized prompt and context possible.

Defining the Passive Context Aggregator (PCA)

The **Passive Context Aggregator** (PCA) is characterized by a reliance on **Static Context Inclusion**. **PCA** architectures typically feature one or more of the following:

Unmanaged State: The agent accrues the entire message history and passes it to the LLM on every turn. No mechanisms for summarization, compression, or relevance filtering are employed.

Fixed-Template Execution: All configured tools, rules, and system instructions are included in the prompt for every call, regardless of the relevance of that context to the current query.

LLM Role Overload: The single execution LLM is burdened with multiple, simultaneous objectives: **Planner** (deciding which tools to use), **Filter** (discerning relevant history), and **Executor** (solving the problem). This competition among internal goals reduces response quality.

The PCA relies solely on the LLM's inherent ability to filter relevant information from excessive context. As context size, tool count, and rule complexity increase, the LLM's focus disperses, leading directly to **degraded and inconsistent reasoning quality**.

The Cognitive Orchestrator and the Control Plane

The **Cognitive Orchestrator (CO)** architecture solves the PCA's limitations by introducing a decoupled Control Plane, often implemented as a **Supervisor Agent** or a system of chained, specialized agents. The CO ensures that the primary execution LLM is isolated to be a pure **Executor**.

The CO's primary function is to preprocess the user query and the accumulated state, generating an **optimized, relevant, and concise prompt with a singular goal** for the final model call.

Key Architectural Components of a Cognitive Orchestrator

A Cognitive Orchestrator can implement many techniques, with some of the most common being:

Memory Management Agent (Summarizer): Its functionality is to analyze the full chat history and employ a smaller LLM or heuristic to compress, summarize, or extract the most relevant core themes. The primary gain is **Efficiency & Context Scaling**, which prevents LLM performance degradation from "context drowning."

Routing and Relevance Filter: Its functionality is to determine, using heuristics or a small LLM, which specific tools, rules, or external data sources are relevant to the *current* query. The primary gain is **Precision & Focus**, which enforces a singular, focused goal for the Execution LLM, drastically improving response quality and consistency by filtering out noise.

External Grounding (Lookup Agent): Its functionality is to explicitly check if the query requires up-to-date or factual information before any LLM call, executing a search and providing the results as optimized context. The primary gain is **Accuracy**

(Grounding), which mitigates the risk of the LLM relying on stale training data, ensuring factually grounded responses.

Hierarchical Planning (Decomposition Agent): Its functionality is to break complex requests into a sequential chain of sub-steps, executing the plan step-by-step and using the output of one step as input for the next. The primary gain is **Reliability & Complexity**, which enables the system to handle multi-step tasks that are intractable for the PCA.

The Core Advantage: Preventing Cognitive Overload

The most significant comparative advantage of the Cognitive Orchestrator over the Passive Context Aggregator is the optimization of the **LLM's cognitive load.** There can also be an added benefit of cost savings from token reduction, but the central design imperative is **response quality**.

By performing the roles of **Planner, Filter, and Resource Manager** (and others) outside of the main LLM, the Cognitive Orchestrator avoids the cognitive burden inherent in the PCA.

To recap:

Passive Context Aggregator: Relies solely on the LLM's raw ability to filter relevant information from excessive context. The forced multi-tasking leads to **degraded and inconsistent reasoning quality**.

Cognitive Orchestrator: Prevents **LLM Cognitive Overload** (too many goals, tools, and irrelevant data). The LLM is isolated to be a pure **Executor**. By delivering an *optimized, relevant, and concise* prompt with a **singular goal**, the system achieves **dramatically higher quality results** and consistency.

In essence, the "sophisticated" solutions are running a small, fast **Control Agent** that writes the perfect, optimized prompt for the larger, more powerful **Execution Agent.** This division of labor is the true architectural leap in modern LLM application development.

Agent Frameworks

If you're not rolling your own agent code, but are using an agent framework, you will likely have access to some degree of **Cognitive Orchestration** (usually called by other names). For example, frameworks like **LangChain** and, particularly, **LangGraph**, are

built specifically to enable this architectural separation by allowing developers to construct a dedicated **Control Plane**. In **LangGraph**, the execution flow is defined as a **state machine** where you insert small, fast functional nodes (procedural functions or smaller, inexpensive LLMs) to perform the planning, filtering, and resource management tasks. This is achieved through **conditional routing**, which dynamically decides whether the graph should flow to a specialized tool execution agent (like a code interpreter), a factual lookup agent, or directly to the final execution LLM based on the user's current query.

These modular components handle the key aspects of Cognitive Orchestration. For instance, **Memory Management** is handled by dedicated components like LangChain's *ConversationSummaryBufferMemory*, which automatically compresses chat history and prevents context bloat outside the main prompt. For **Relevance Filtering**, you simply create a small node that classifies the user's intent (e.g., "Math" or "Search"), which then gates the flow to the appropriate tool agent. By offloading these preparatory, cognitive tasks to specialized chained functional nodes, the system ensures the primary **Execution LLM** receives a highly refined, concise prompt with a **singular goal**, which is the secret to achieving dramatically better reasoning quality and consistency.

TsAgent Context Orchestration via Supervision

We have our own agent development platform called TsAgent. It is an open-source TypeScript-first platform for building, testing, running, and orchestrating AI agents. It provides a complete ecosystem from desktop and CLI apps for no-code agent creation to production-ready agent servers, all supported by TypeScript APIs.

Rules and References to Create Agents that Learn and Remember

One of the key design elements of **TsAgent** is its **Rules** and **References** support, where the user can manage rules (sets of instructions for the model) and references (any sort of ground truth or other data specific to a given topic or type of problem). We allow the user to have a high level of control of which rules and references are included in the current session context (through our UX or via the API).

We also exposed the management of rules and references to the LLM itself through a set of internal tools, to do things like list rules or references, create or update rules or reference, or include/exclude given rules/references in the current session context. And we added system prompts to instruct the LLM to use these tools, and to

manage these rules and references (creating new ones when needed, adding relevant ones to its context, etc).

What we found was that the LLMs would interact with rules and references when specifically directed by a user prompt, but would almost never do it on its own (even with lots of system prompting). Our plan of making agents that would "learn and remember" did not really work out of the gate.

Cognitive Orchestration to the Rescue

Inspired by the benefits of **Context Orchestration** covered above, we implemented a control plane in TsAgent that we call the **Supervisor System**. It allows for simple modules called **Supervisors** to be plugged into the agent conversation with full access to the context (including message history, rules, references, tools, statistics. etc). Multiple Supervisors can be chained together, with each Supervisor performing a specific, focussed role. One Supervisor might summarize message history, and another might be responsible for selecting the correct set of tools required for a given prompt.

One of the Supervisors we provide out-of-the-box is the **Supervisor Agent**, which is what it sounds like, a Supervisor that sends LLM requests and responses for an executor agent (the main agent) to a supervising agent, where that supervising agent has full access to the context of the executor agent (so it can do things like summarize context, select appropriate tools, etc, as directed by its own system prompts). You can build these Supervisor Agents using the standard TsAgent tooling (including through the UX) and they can do anything they are prompted to do.

As just one example, instead of sending every configured tool with every LLM request like we used to do, we have a Supervisor Agent responsible specifically for tool selection that looks at the context and all available tools, and includes only the relevant tools into the executor agent's context. And that itself was a no-code agent built with TsAgent tooling in a few minutes, which produced LLM requests with smaller context (lower cost) that ran faster and produced better results.

The Supervisor control plane, especially combined with the Supervisor Agent, should open up a world of possibilities for building agents that really can learn and remember, and give much better answers while they're doing it.

Conclusion

The superior performance of proprietary LLM applications, and advanced agent frameworks, stems from their Control Plane architectures, not just their foundation models. Moving from a **Passive Context Aggregator** to a **Cognitive Orchestrator** is a shift from simple **Data Plane-centric** execution to sophisticated **Control Plane-centric** management. For developers looking to close the agent performance gap, the strategic integration of a Context Orchestration solution to manage context, tools, and goals is essential.

Ai Agent | Llm | AI

Follow

Written by Bob Dickinson

20 followers · 6 following

Scalable CTO

No responses yet

Bgerby

What are your thoughts?

More from Bob Dickinson

Bob Dickinson

Stop Running Your MCP Tools via npx/uvx Right Now

TL;DR — If you are running MCP tools on your development machine using npx or uvx, stop right now. This widespread behaviour, encouraged...

May 14 3 3

Bob Dickinson

Environment Variables with Containerized Next.js

When I started developing with Next.js I used environment variables fairly liberally, especially for things like API authentication params...

Jul 25, 2024 2

Bob Dickinson

Actually, Go Ahead and Run Your MCP Tools via npx/uvx

TeamSpark MCP ToolVault to the Rescue

Aug 4 36

Bob Dickinson

Easy Structured Logging with Next.js in Google Cloud

In the Beginning, There Was No Logging

Jan 14 11

See all from Bob Dickinson

Recommended from Medium

Micheal Lanham

Microsoft Just Solved the AI Agent Problem (And Open-Sourced the Solution)

How the new Agent Framework unifies Semantic Kernel and AutoGen to bring enterprise-grade AI agents from prototype to production

Oct 14 46 1

In Coinmonks by Sweety Tripathi

Understanding Long-Term Memory in LangGraph: A Hands-On Guide

Maintain Context over time, adapt to our preferences , learn from our past instructions

Oct 13 7

In Google Cloud - Community by Karl Weinmeister

Python and Rust interoperability: A walkthrough for building a high performance MCP server

You'll learn step-by-step instructions for including Rust code with your Python code. We'll build a tool for AI agents compliant with MCP.

Oct 19 85

In Level Up Coding by Arjun

Building Voice Agents in 2025—Part I

Your Voice Agent Shouldn’t Sound Like a Fax Machine.

Aug 12 14

Chamuditha Kekulawala

Model Context Protocol (MCP) and it’s limitations

In part 1 we got a high level understanding of what MCP is. Now let's go into more technical details: MCP provides a standardized way for...

May 28 5

Heeki Park

Building an MCP server as an API developer

Anthropic released MCP at the end of November 2024. It took a few months for it to catch on, but my, oh my, it feels like the community is...

May 14 571 5

See more recommendations