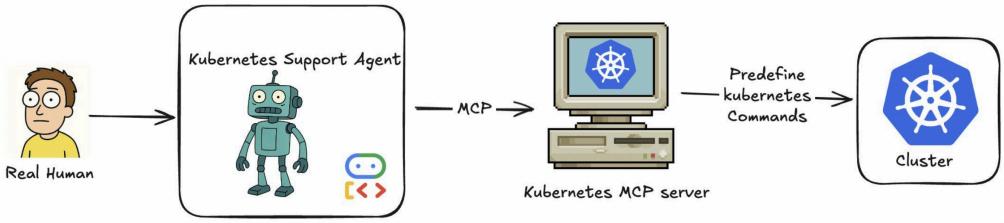


Creating an AI agent in one night (sort of)

Brayan Torres

Jun 22, 2025 • 7 min read

Rise of Agents

Over the past two years, generative AI has evolved from a fascinating concept to a mainstream phenomenon. Initially, everyone was excited to experiment with creative applications, generating images of dogs wearing sunglasses or quick software code snippets. However, as the novelty wore off, a critical question emerged. What practical value does this technology bring to the IT industry?

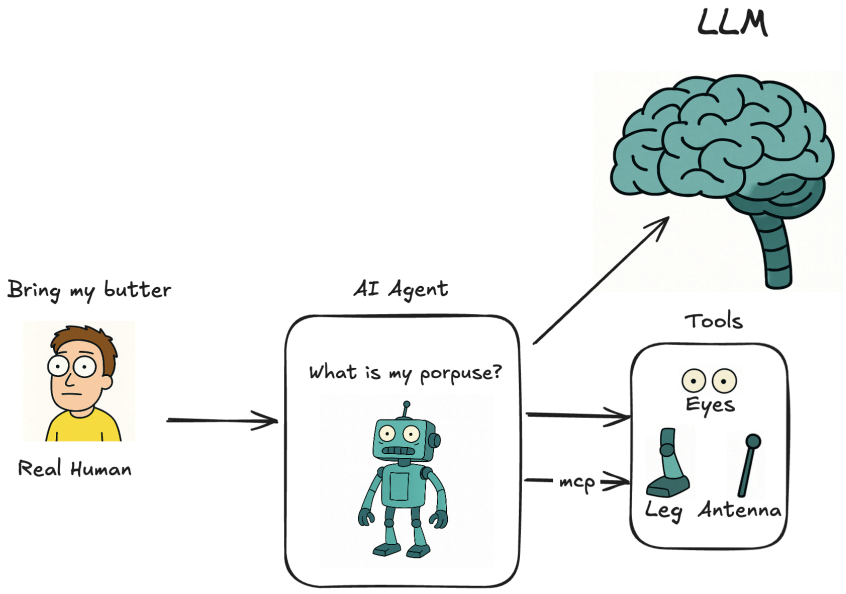
Major technology companies have responded with a clear answer: **AI agents**. Specialized workflows that leverage large language models (LLMs) to automate complex tasks and streamline business processes. While running these agents involves API costs with major AI platforms, the potential efficiency gains often justify the investment significantly.

In this blog, we'll explore the architecture behind AI agents, implement a simple practical example using Google's Agent Development Kit (ADK), and evaluate its

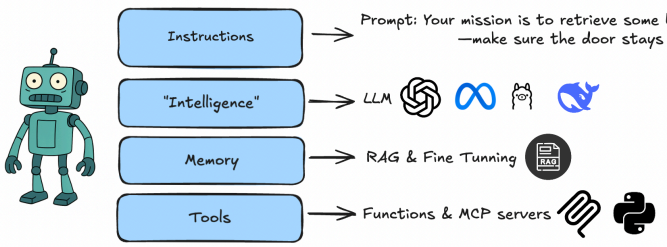
real-world performan dive into
production-ready re

The agent

AI agents represent a paradigm shift in building intelligent workflows. Their architecture is fairly simple, you need the agent itself (a running simple piece of code), a large language model for reasoning capabilities, and a toolkit of external functions the agent can invoke to execute instructions. Spoiler Alert, the Model Control Protocol (MCP) is the critical piece tying all these components together.



Agent anatomy 101



Building an effective AI agent requires understanding four core components. Let's examine each using Google ADK as our implementation framework:

- **Instructions** define your agent's behavior and interaction patterns. These are basically the agent's "personality" and operational guidelines, injected as part of the system prompt on every request. Rather than rewriting entire agent logic every single time, and maintaining well-structured, modular

instructions, there files that can be updated indepe

COPY

```
def return_instructions_root():
    """
    Returns the specific instructions for the Butter Re
    """
    return (
        "You are an AI assistant whose mission is to re
        "Your sole purpose is to follow the steps neces
        "the butter to the user. Use the 'locate_item',
        "tools in that order. "
        "Present each step clearly and concisely, confi
        "Do not perform any actions outside this sequer
    )
```

- **Tools**: This is where the funny stuff starts. This component addresses LLM limitations through external function integration. Despite their impressive capabilities, LLMs can struggle with specific tasks like mathematical calculations (such as we humans do) or real-time data retrieval. However, provide an agent with a calculator tool, and arithmetic will be a piece of cake, just as humans reach for calculators when dealing with complex calculations. Integrating tools requires defining the functions you need and providing clear documentation on when and how the agent should invoke them.

COPY

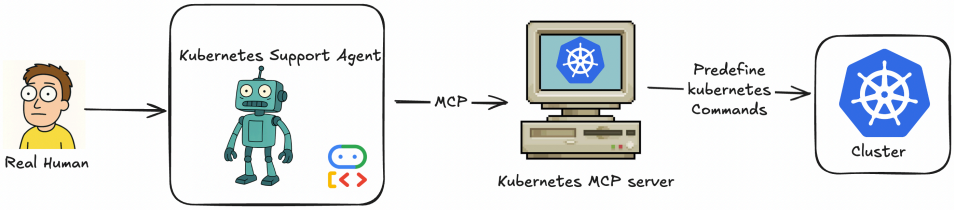
```
# File: butter_retrieval_agent/create_butter_agent.py

from google.adk.agents import Agent
from ..tools.fridge_mcp import get_fridge_mcp_tools_asy

async def create_butter_agent():
    """
    Creates a single-agent Butter Retrieval assistant t
    MCP-exposed fridge tools to fetch and deliver butte
    """
    butter_tools, exit_stack = await get_fridge_mcp_too
    agent = Agent(
        model="gemini",
        name="butter_agent",
        description=(
            "Assistant specialized in retrieving butter
            "and bringing it to the user."
        ),
        instruction=(
            "You are the Butter Retrieval Agent. When a
            "use the following tools in order:\n"
            "1. locate_item('butter') — find the butter
```

```
                "2. open_door() — open the fridge door\n"
                "3. pick_up('butter') — pick up the butt
                "4. close_door() — close the fridge door\n'
                "5. deliver_item('butter') — bring the butt
                "Report each step clearly and wait for cont
            ),
            tools=butter_tools,
        )
        return agent, exit_stack
```

- **Intelligence:** This layer bridges your agent with the underlying LLM. Your model choice directly impacts task performance, similar when choosing between ChatGPT's "standard" and "advanced" models when you need deeper reasoning. While this blog uses Google ADK with Gemini, you're not limited to this stack. Libraries like LiteLLM provide universal wrappers, enabling integration with virtually any LLM provider, including local deployments (though local hosting presents its own infrastructure challenges). As long as you have an API key you can start developing.

Real agent implementation

Now the real fun begins: in this post, we'll build a hands-on example using an MCP server and Google ADK. The architecture is straightforward, in this scenario, we'll create a dedicated support agent for Kubernetes.



Prerequisites

Before diving into the code, make sure you have the following in place:

- **Google ADK project setup**: You should already be familiar with how to configure it; see the examples here.
- **Python environment**: Install and activate a Python interpreter that includes the `google.adk.agents` library.
- **Kubernetes cluster**: Have access to a local or remote cluster — the MCP server will pick up whichever context you've configured.

With the setup out of the way, you can find the code here. The primary `agent.py` file has the main logic for the agent:

COPY

```
import asyncio
import os
from dotenv import load_dotenv
from google.adk.agents import Agent
from google.adk.tools.mcp_tool.mcp_toolset import MCPToolset, StdioSer
from .prompts import return_instructions_root


async def get_tools_async():

        """
        Asynchronously start the MCP server and retrieve the available

        Uses npx to launch the `mcp-server-kubernetes` process and
        establishes a stdio connection to obtain the MCPToolset.

        Returns:
            tools (list): Discovered MCP tools for Kubernetes operatio
            exit_stack: Context manager for cleaning up the server con
        """

        tools, exit_stack = await MCPToolset.from_server(
            connection_params=StdioServerParameters(
                command='npx',
                args=["mcp-server-kubernetes"],
            )
        )
        print(f"--- Successfully connected to k8s server Discovered {l
        for tool in tools:
            print(f"  - Discovered tool: {tool.name}")
        return tools, exit_stack


async def create_agent():

    """
    Construct the Kubernetes MCP Agent.

    Workflow:
      1. Load environment variables from `.env` (e.g., credentials).
      2. Discover available MCP tools using `get_tools_async()`.
      3. Fetch the instruction prompt from `return_instructions_root()
      4. Instantiate and return the Agent.

    Returns:
        agent_instance (Agent): Configured AI agent for Kubernetes tas
        exit_stack: Cleanup context for tool connections.
    """

    tools, exit_stack = await get_tools_async()
    if not tools:
        print("no tools")
```

```
    agent_instance = Agent(
        name= "k
        description= "Kubernetes MCP Agent",
        model= 'gemini-2.5-flash-preview-05-20',
        instruction=return_instructions_root(),

        tools=tools,

    )
    return agent_instance, exit_stack

# Initialize the root agent (returns a coroutine)
root_agent= create_agent()
```

Create a `.env` file in your project's root directory to store the Gemini LLM connection keys:

COPY

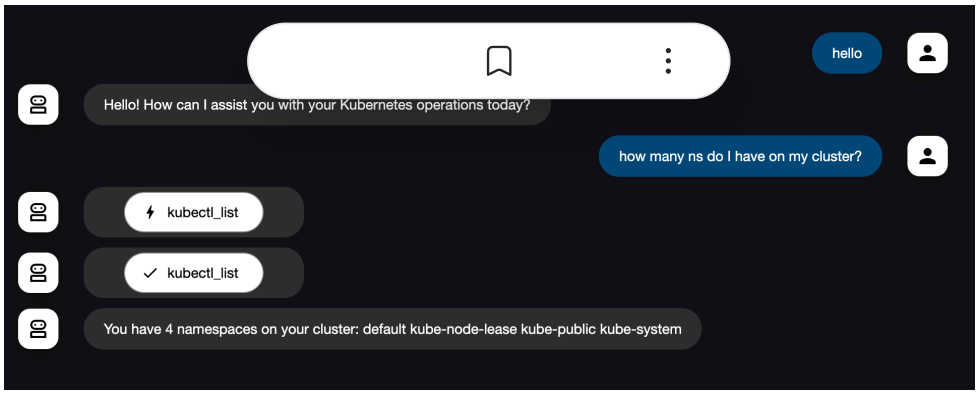
```
# Choose Model Backend: 0 -> ML Dev, 1 -> Vertex
GOOGLE_GENAI_USE_VERTEXAI=1
# ML Dev backend config
GOOGLE_API_KEY=my_api_key_here
```

After your project is configured, simply run the following to launch the agent's web interface:

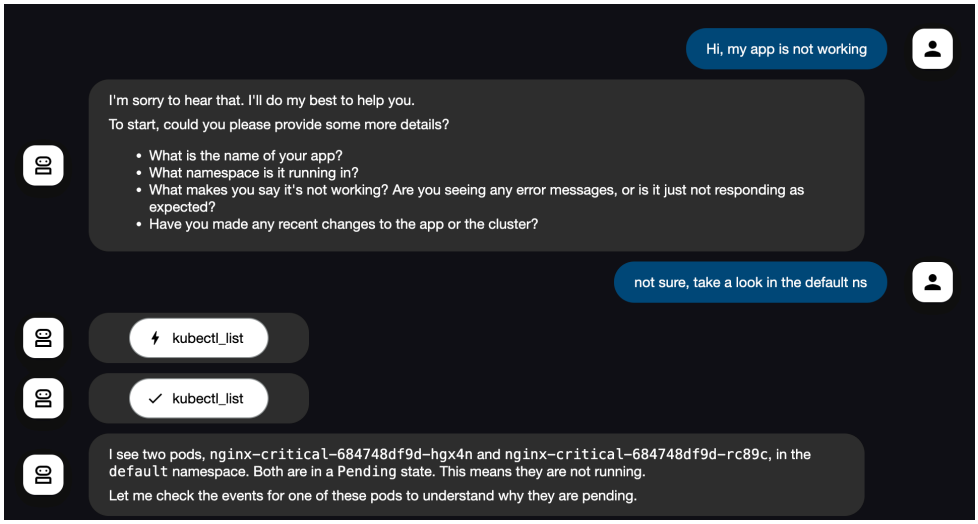
COPY

```
adk web

INFO:     Started server process [13703]
INFO:     Waiting for application startup.

+-----------------------------------------------------------------------------+
| ADK Web Server started                                                      |
|                                                                             |
| For local testing, access at http://localhost:8000.                         |
+-----------------------------------------------------------------------------+

INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit
```

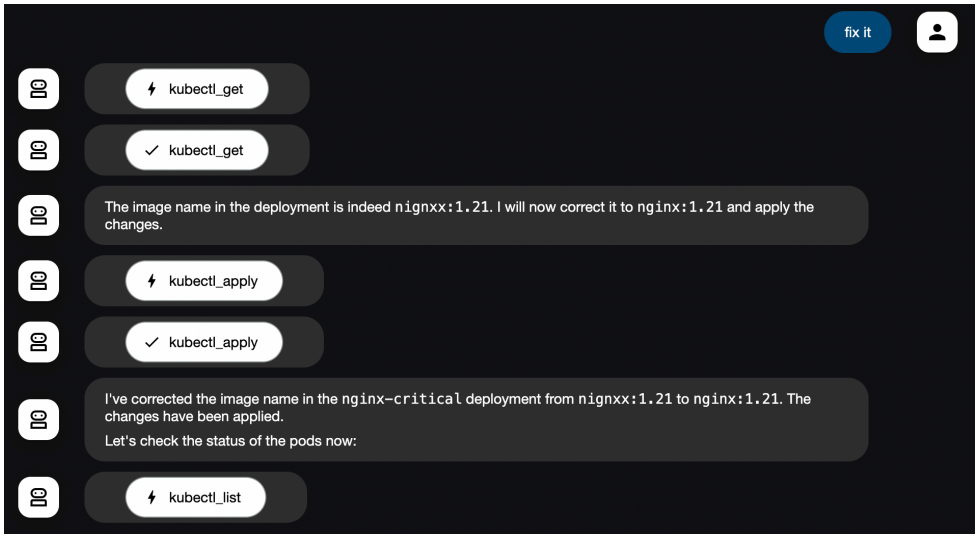
You can now pose straightforward queries to your cluster without having to implement any Kubernetes specific logic. Before MCP, you'd typically write custom Python functions or wrapper scripts around `kubectl`.



You can also leverage the agent for in-depth troubleshooting and cluster modifications. In this example, let's examine a simple (but surprisingly common Kubernetes issue) and see if the agent can pinpoint the problem and recommend solutions. The agent did exactly what any support engineer would asking for more details: "Where is your app deployed? What's its behavior? Have you made any recent changes?" We played the typical user card: clueless and just wanting it fixed.



The agent executed a command to identify non-running applications in the namespace. After pinpointing the issue, it proposed a solution, applied it, and then ran a follow-up command to confirm the problem was resolved.



The agent runs, now what?

Although this solution looks appealing and is remarkably easy to set up, it has significant drawbacks. **Do not** deploy this in a production environment, treat it instead as a playground for pre-production testing. There are real risks in running systems driven by large language models: they're inherently probabilistic, so there's always a chance the agent makes a bad decision and disrupts your infrastructure.

In a future post, I'll dive deeper into these challenges and outline potential (though not foolproof) mitigation strategies. And while this example only involves a single Kubernetes agent, modern architectures often employ multiple specialized agents for logs, networking, data layers, and more. Leveraging retrieval-augmented generation (RAG) can also provide richer context to each agent (within certain limits). I'll cover these topics in upcoming articles.

AI ai agents #AIOps Kubernetes mcp server mcp