Building AI Agents to Manage Your Clouds

Introduction - Cost of Cloud Complexity

Today the software and SaaS landscape is powered by the cloud. Organizations are racing to embrace innovation, but the complexity of managing cloud resources often becomes a stumbling block. With hundreds of services, configurations, and APIs, even experienced teams face challenges in efficiently managing and optimizing their cloud environments.

For newcomers, this complexity translates into a steep learning curve. Teams often spend more time skilling up or firefighting operational issues than driving innovation. Sadly the result of this is lost productivity, increased costs, and missed opportunities.

All this growing complexity calls for a smarter, more efficient solution. Here is where AI-powered agents, specifically tailored for cloud management, offer hope.

AI Agents in Cloud Management

AI agents are emerging as game-changers in how we interact with cloud platforms. Unlike traditional automation tools or dashboards, AI agents offer:

- **Conversational Interfaces**: Simplifying the user experience by transforming complex operations into intuitive conversations.
- **Intelligent Insights:** Providing actionable recommendations and summarizing large datasets.
- **Dynamic Adaptability**: Learning from interactions and evolving to meet organizational needs.

These capabilities empower teams to focus on innovation and their core responsibilities rather than mastering (and re-learning) the intricacies of cloud infrastructure. By acting as virtual assistants, AI agents lower the barriers to entry for cloud management and free up engineers to tackle high-value tasks.

Importance of Open Source

Building AI agents using open-source tools isn't just a cost-saving measure - it's a strategic decision. Here's why:

- **Transparency**: Open-source tools are auditable, ensuring trust and reliability.
- Community Collaboration: They leverage a global ecosystem of contributors, constantly improving functionality and fixing bugs.
- Vendor Independence: Open-source solutions avoid lock-in, giving you the freedom to adapt and innovate.
- Cost-Effectiveness: With open-source, you only pay for infrastructure, not licensing fees.

By choosing open-source for our AI agent, the solution in this tutorial is both accessible and extensible, perfectly aligning with the ethos of cloud-native development.

Problem: Easy Cloud Insights

Cloud platforms like AWS offer unparalleled flexibility but at the cost of complexity. Consider these tasks:

- Fetching a list of all running EC2 instances.
- Analyzing the cost of underutilized resources.
- Ensuring security groups are configured correctly.

Each task requires navigating multiple services, APIs, and data formats. Manually managing this is time-consuming and error-prone.

Overview

In this tutorial, we will walk through building a very simple open-source AI agent to:

- Fetch and summarize cloud resources.
- Use natural language to interact with cloud services.
- Identify and use the most relevant tools for each query.

This solution/stack is based on Mistral 7B, served by Ollama for natural language understanding, FAISS running as an in-memory vector store for efficient tool retrieval, and Boto3 for AWS API interactions.

With that let us get started. If you are interested in the final code, it can be found in our GitHub repository.

Prerequisites

Our Agent will have some limitations being a simple demo (albeit with room for lots of extensions):

1. It will only be limited to obtaining insights from your AWS environment.
2. It will run locally on Ollama so it is not hosted anywhere yet.
3. It will not perform any system updates. Clearly, giving write/update access to a demo is not desirable.

Assumptions

- You have python3 and a virtualenv setup (for installing the necessary dependencies).
- You have configured your AWS cli environment (**aws configure**)
- You have installed Ollama.

With that, let us get started.

Tutorial: Building an Open Source AI Agent

Step 1 - Download the model

We will be using Mistral-7B model. Mistral is highly efficient and can handle queries with contextual understanding, making it suitable for summarization and Q&A tasks. Its size (7 billion parameters) is a good balance between computational efficiency and performance, ideal for on-premises or edge deployments in cloud environments. Mistral also offers open weights so it is ideal for enabling customizations for the cloud and Kubernetes domains.

```
ollama run mistral
```

This downloads mistral and starts the (Ollama) API server to start serving completions for our Agent. It also starts an interactive terminal where you can try out queries manually.

Step 2 - Setup python dependencies

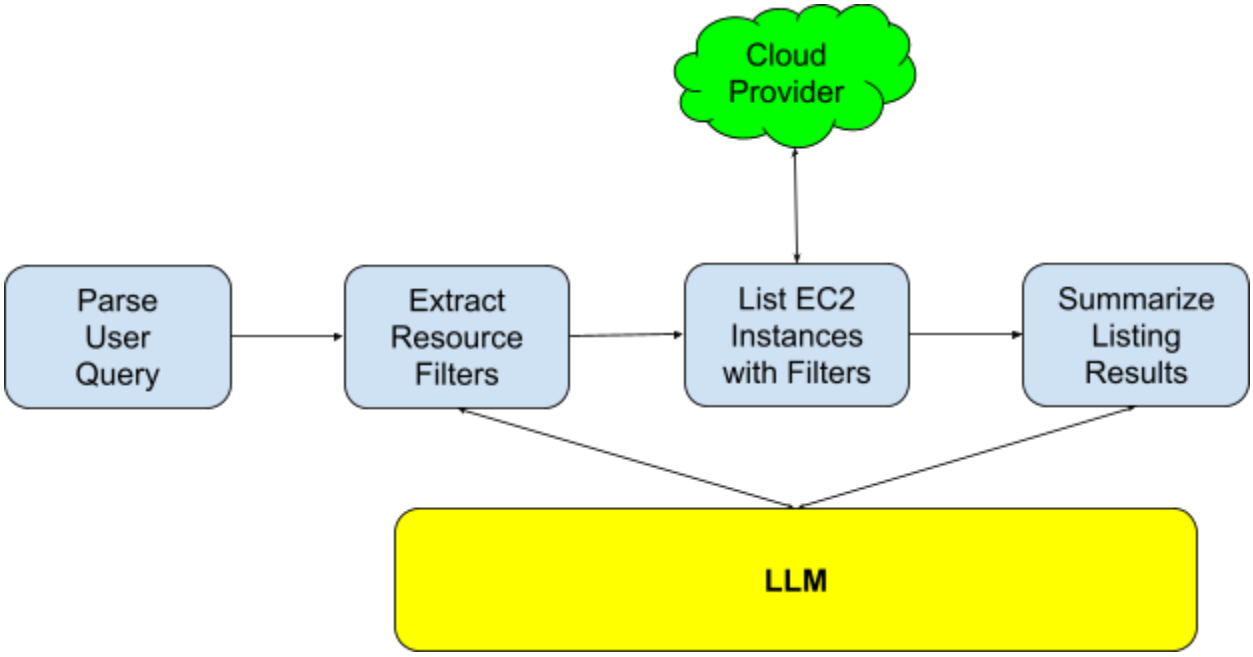
Once in your virtual environment, install the following dependencies:

```
pip install boto3 requests
```

Step 3 - A basic version

We will start with a very basic version. Get a list of EC2 instances that match the filters provided by the user's query.

The flow will look like:



Before doing this, we will create a generic helper class that acts as an interface with our LLM so that it can handle sending of a prompt to the LLM and parsing the result as JSON. This abstraction makes it easier to treat the LLM calls as a blackbox without worrying about the details of response parsing etc.

```python
import os, json
import requests

MODEL = "mistral"
url = "http://localhost:11434/api/generate"

class LLM:
    def call(self, prompt):
        payload = {"model": MODEL, "prompt": prompt, "stream": False}
        response = requests.post(url, json=payload)
        return response.json()["response"]
```

This is a pretty basic LLM wrapper:

1. Take a prompt from the user
2. Send the prompt to the LLM (without streaming so that we wait for the complete response before parsing it)
3. Parse the response as JSON (this will throw an exception if JSON fails but this is fine and will be handled upstream)

Now we can simply use an LLM `instance` as we see fit going forward.

Back to our first version, let us look at the user query parsing method:

```
def parse_query(llm, user_query):
    prompt = f"""You are an intelligent assistant for AWS cloud management. Your task is to
interpret the user's query and return a valid JSON object with:

- action: Name of the action to take to obtain ec2 resources.
- region: The region where the aws action should be taken.
- filter: The filters to apply to fetch the ec2 resources on.

The user query is: "{user_query}"
"""
    return llm.call(prompt)
```

Calling this with the query: List all my ec2 instances in us-west-2 should yield something like (depending on your environment):

```
' {\n    "action": "describe_instances",\n    "region": "us-west-2",\n    "filter": {}\n  }'
```

When the JSON is decoded, it should yield:

```
{
  "action": "DescribeInstances",
  "region": "us-west-2",
  "filter": None
}
```

Next, we will create a "helper" function that runs the appropriate command:

```
def fetch_ec2_instances(region, state=None):
    """Fetch EC2 instances based on region and state filter."""
    ec2 = boto3.client('ec2', region_name=region)
    response = ec2.describe_instances()
    instances = [
        {
            "InstanceId": instance["InstanceId"],
            "State": instance["State"]["Name"],
            "Type": instance["InstanceType"]
        }
        for reservation in response["Reservations"]
        for instance in reservation["Instances"]
        if state is None or instance["State"]["Name"] == state
    ]
    return instances
```

We would invoke this from the results of the previous call as:

```
query = json.loads(...previous_query_results...)
region = query.get("region", "us-east-1")
filters = query.get("filters", {})
fetch_ec2_instances(region, filters.get("state"))
```

Running this for our region (us-west-2) above should yield (something like):

```
[{"InstanceId": "i-096d05cd9f5465a27", "State": "running", "Type": "t3.4xlarge"},
 {"InstanceId": "i-80747e0dff16a4180", "State": "running", "Type": "t3.4xlarge"},
 {"InstanceId": "i-c0af3e69bc8180251", "State": "running", "Type": "t4.xlarge"},
 ...
]
```

Finally, we will take this response and pass it to a summarizer so we can see the results in a more understandable way:

```
def summarize_data(llm, data, resource_type):
    """Summarize resource data using Mistral."""
    prompt = f"Summarize the following {resource_type} data: {data}"
    return llm.call(prompt)
```

For example: **print(summarize_data(llm, x, "EC2 instances"))** should yield (something similar to):

Here is a summary of the provided EC2 instances data:

- Total number of running instances: 16
- All instances have the same state: 'running'
- The instance types are a mix of t3 (2xlarge, xlarge), t5 (xlarge), t4 (xlarge) and t2 (xlarge, 2xlarge). There is also one instance of type t3.4xlarge that appears twice in the list.
- No information about the Availability Zone or instance launch time is provided in this dataset.

Voila!

Let us put this into a loop so the user can enter their queries without having to invoke Python functions manually:

```python
def main():
    llm = LLM()
    print("Welcome to the Cloud Inventory Dashboard! Ask me about your AWS resources.")
    while True:
        # User Input
        user_query = input("\nYou: ")
        if user_query.lower() in ["exit", "quit"]:
            print("Goodbye!")
            break

        # Parse Query
        try:
            parsed_query = parse_query(llm, user_query)
            print(f"\nParsed Query: {parsed_query}")

            query = json.loads(parsed_query)
            action = query.get("action")
            resource = query.get("resource")
            region = query.get("region", "us-east-1")
            filters = query.get("filters", {})

            # Fetch Data Based on Resource Type
            state = filters.get("state")
            data = fetch_ec2_instances(region, state)
            summary = summarize_data(data, "EC2 instances")
            print(f"\nAgent: {summary}")
        except Exception as e:
            print(f"\nAgent: Sorry, something went wrong. {str(e)}")
```

Running this should get the user in a loop to enter the queries, e.g.,:

You: list all my ec2 instances

Parsed Query: {
"action": "DescribeInstances",
"region": "us-west-2",
"filter": null
}

Agent: 1. The provided data represents information about multiple Amazon Elastic Compute Cloud (EC2) instances that are currently running on the AWS platform.

2. A total of 15 instances are listed, with each instance identified by an InstanceId and having a State and Type associated with it.

3. In terms of their States, all instances are 'running'.

4. The Types of instances include t3.4xlarge (8 instances), t3.xlarge (1 instance), t5.xlarge (1 instance), t4.xlarge (1 instance), and various types of M5D instances such as t2.xlarge (3

instances), t2.2xlarge (4 instances).

5. Some additional instance IDs include: i-aaaa, i-bbbb, i-cccc, i-dddd, i-eeee, i-ffff, i-gggg, i-hhhh, i-iiii, i-jjjj, ...

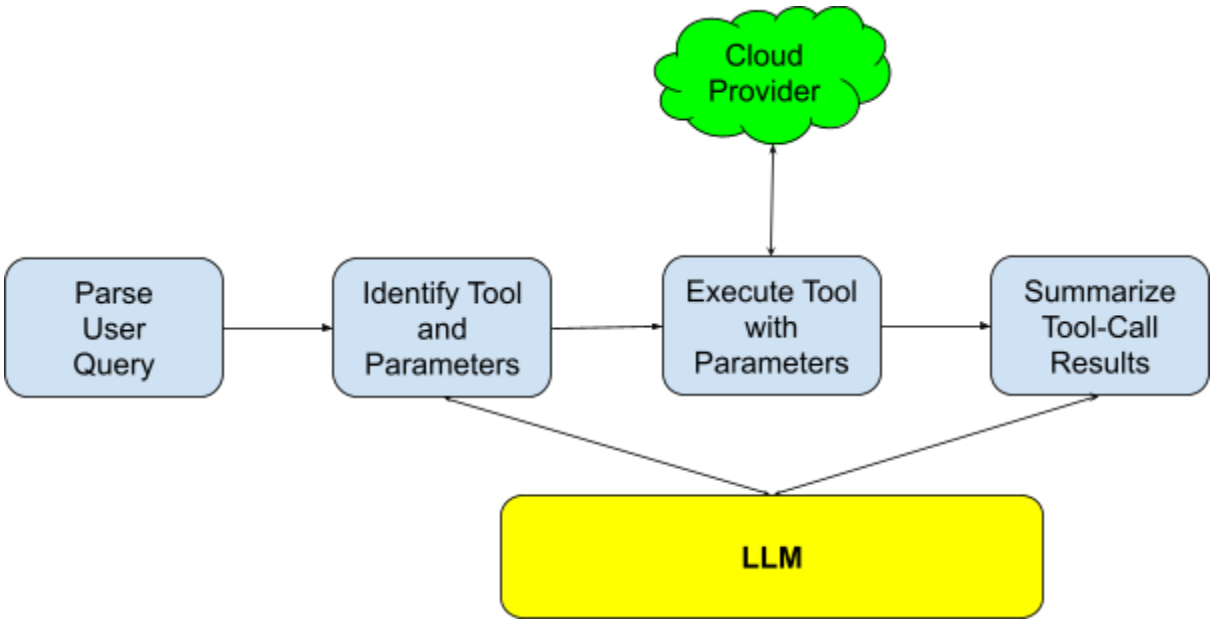
You: list all my instances in us-east-1

Parsed Query: {
"action": "DescribeInstances",
"region": "us-east-1",
"filter": {}
}

Agent: There is no EC2 (Amazon Elastic Compute Cloud) instances data provided to summarize. To help you, please provide the necessary data or details about the EC2 instances you want summarized. This could include instance type, region, operating system, memory, CPU, storage, and other relevant attributes. Once I have that information, I can help you by summarizing it for better understanding.

Step 4 - Adding more resource types

We built a very basic REPL loop running that takes a user query, parses it, submits it to boto for execution, and then summarizes the results. Clearly, this is a very limited Agent. Simply calling ec2's DescribeInstances with varying filters barely an Agent makes! Let us add more resource types. Here we introduce **tool calling**. Instead of calling a single tool, we will delegate the tool selection also to the LLM. Our flow is now thus:



Before we can do this let us build a tool inventory:

```
tools = [
    {
        "name": "fetch_ec2_instances",
        "description": "Fetches details of EC2 instances in a specific AWS region.",
        "parameters": {
            "region": "The AWS region to query (e.g., us-east-1).",
            "state": "Optional. Filter instances by state (e.g., running, stopped)."
        }
    },
    {
        "name": "fetch_s3_buckets",
        "description": "Fetches the list of S3 buckets in the account.",
        "parameters": {}
    },
    {
        "name": "fetch_rds_instances",
        "description": "Fetches details of RDS instances in a specific AWS region.",
        "parameters": {
            "region": "The AWS region to query (e.g., us-east-1)."
        }
    }
]
```

And their accompanying methods:

```
import boto3

def fetch_rds_instances(region):
    """Fetch details of RDS instances in a specific AWS region."""
    rds = boto3.client('rds', region_name=region)
    response = rds.describe_db_instances()
    instances = [
        {
            "DBInstanceIdentifier": db["DBInstanceIdentifier"],
            "Engine": db["Engine"],
            "Status": db["DBInstanceStatus"],
            "Region": region
        }
        for db in response.get("DBInstances", [])
    ]
    return instances

def fetch_s3_buckets():
    """Fetch the list of S3 buckets in the account."""
    s3 = boto3.client('s3')
    response = s3.list_buckets()
```

```
    buckets = [
        {
            "Name": bucket["Name"],
            "CreationDate": bucket["CreationDate"].strftime("%Y-%m-%d %H:%M:%S")
        }
        for bucket in response.get("Buckets", [])
    ]
    return buckets
```

Now we can modify our query parser to include tools:

```
def parse_query_with_tools(llm, user_query, tools):
    """Parse user query and select the appropriate tool using Mistral."""
    tools_description = "\n".join(
        f"- Tool: {tool['name']}\n  Description: {tool['description']}\n  Parameters:
{tool['parameters']}"
        for tool in tools
    )

    prompt = f"""
You are an intelligent assistant for AWS cloud management. Your task is to interpret the
user's query and select the most appropriate tool from the following list:

{tools_description}

User Query: "{user_query}"

Return a JSON object with:
- tool: The name of the tool to use.
- parameters: The parameters required for the selected tool.
    """

    return llm.call(prompt)
```

Now with

json.loads(parse_query_with_tools(llm, "List all my ec2 instances in us-west-2", tools))

we should see something like this:

```
{
    "tool": "fetch_ec2_instances",
    "parameters": {"region": "us-west-2"}
}
```

With this calling our tool is now a straightforward matter. We simply delegate the tool execution to a central “router” method:

```
def call_tool(tool_name, parameters):
    """Call the appropriate tool with the given parameters."""
    if tool_name == "fetch_ec2_instances":
        return fetch_ec2_instances(parameters.get("region"), parameters.get("state"))
    elif tool_name == "fetch_s3_buckets":
        return fetch_s3_buckets()
    elif tool_name == "fetch_rds_instances":
        return fetch_rds_instances(parameters.get("region"))
    else:
       raise ValueError(f"Unknown tool: {tool_name}")
```

Will also update our main loop so that tool identification and invocation are incorporated:

```
def main():
    ...
    while True:
        parsed_result = ....        # Same as before in calling the query parser

        # Get the tool, parameters and invoke the caller:
        tool_name = parsed_result.get("tool")
        parameters = parsed_result.get("parameters", {})
        print(f"\nAgent: Using tool '{tool_name}' with parameters {parameters}")


       # Call the appropriate tool
       tool_output = call_tool(tool_name, parameters)
       summary = summarize_data(llm, tool_output, tool_name)

       # Print summary as before
       print(f"\nAgent: {summary}")
```

And now the user can do more across a wider range of tools:

```
Welcome to the Cloud Inventory Dashboard! Ask me about your AWS resources.

You: list my s3 buckets

Parsed Query:  {
    "tool": "fetch_s3_buckets",
    "parameters": {}
  }

Agent: Using tool 'fetch_s3_buckets' with parameters {}
```

```
Agent:  The provided data represents a list of S3 buckets in Amazon Web Services (AWS). Here are the details for each bucket:

1. xxxx was created on June 14, 2024 at 20:29:45.
2. xxxx was created on June 14, 2024 at 02:56:24.
....
11. yyyy was created on October 3, 2024 at 22:56:56.
12. yyyy was created on October 31, 2024 at 18:20:21.

Each bucket has a unique name and a creation date associated with it.

You: list my ecs instances in us-west-2

Parsed Query:  {
    "tool": "fetch_ec2_instances",
    "parameters": {
      "region": "us-west-2"
    }
  }

Agent: Using tool 'fetch_ec2_instances' with parameters {'region': 'us-west-2'}

Agent:  This data represents a list of Amazon EC2 instances, each instance having an ID, state, and type. Here is a summary of the instances:

1. Instance with ID i-0690dc59d5f645a27, Type t3.4xlarge, State running
2. Instance with ID i-0847e70dff161a408, Type t3.4xlarge, State running
...
14. Instance with ID i-0b884a4b2abb54eb3, Type t2.2xlarge, State running
15. Instance with ID i-0dddb524e3346e931, Type t3.4xlarge, State running

All instances except one (t4.xlarge) are currently running. The types of the instances vary between t2.xlarge, t2.2xlarge, t3.xlarge, and t3.4xlarge.
```

Step 5 - Loading tools dynamically

We can add more tools to our list above. However, this will get cumbersome pretty soon:

1. Manually maintaining this list and mapping an interface is painful and error-prone.
2. The number of tools themselves can be in the thousands so a prompt with all the tools is not feasible and expensive.
3. Most tools will be inappropriate for the query so a way for early pruning is desirable.

Let us solve the first problem (first).

In order to load tools dynamically we have a few options:

1. Move our tool list to a config file and load the config file at startup.
2. Depends on an API spec (like Swagger) to load and manage tool definitions.
3. Dynamically load operation methods from the boto3 library.
4. A hybrid of the above approaches.

For this tutorial, we will keep it simple and implement option 3. Boto3 is a powerful client for managing AWS resources. We can load the list of methods dynamically from this library and treat them as "individual tools".

Generate Tool Definitions from Boto3

Our simple wrapper function for iterating through all boto services is shown below. It goes through the list dynamically, extracts the input

```
def generate_tool_definitions():
    """Generate tool definitions for all supported AWS services and their key operations."""
    print("Generating tool definitions...")
    session = boto3.Session(region_name=os.environ.get("AWS_REGION"))
    def get_method_name(client, operation):
        method_name = [k for k,v in client.meta.method_to_api_mapping.items() if v ==
operation]
        if method_name: method_name = method_name[0]
        return method_name

   tool_definitions = []

   # Get all available services - eg ec2, rds, etc
   for service in session.get_available_services():
       try:
           # Create a client for the particular service
           client = session.client(service)

           # Get all the operations available for this service
           operations = client.meta.service_model.operation_names

           for operation in operations:
               # Get the name of the python method for this operation
               # eg Converts "DescribeInstances -> describe_instances"
               method_name = get_method_name(client, operation)
               if not method_name: continue

               # Very important to make sure the operation is a read operation only
               if get_operation_type(operation) != "read": continue

               # Get the input parameter types for this operation
               opmodel = client.meta.service_model.operation_model(operation)
               shape = opmodel.input_shape
               if not shape: continue

               # Create a tool entry
               tool_definitions.append({
```

```
                    "name": f"{service}_{method_name}",
                    "description": opmodel.documentation,
                    "parameters": shape.members
                })
        except Exception as e:
            print(f"Could not load operations for {service}: {e}")

    return tool_definitions
```

Note that above we are ensuring that we only add read operations (a full-fledged agent can of course do more). We do that in this helper method that infers the method based on a few parameters:

```
def get_operation_type(client, operation_name):
    """Determine if an operation is read or write based on its metadata."""
    try:
        operation_model = client.meta.service_model.operation_model(operation_name)
        http_method = operation_model.http.get("method", "").upper()

        # Classify based on HTTP method
        if http_method == "GET":
            return "read"
        elif http_method in ["POST", "PUT", "DELETE"]:
            return "write"

        # Fallback classification based on naming conventions
        if any(keyword in operation_name.lower() for keyword in ["describe", "list", "get",
"check"]):
            return "read"
        elif any(keyword in operation_name.lower() for keyword in ["create", "update",
"delete", "put"]):
            return "write"
    except Exception as e:
        print(f"Error determining operation type for {operation_name}: {e}")
    return "unknown"
```

```
[{'description': '<p>Retroactively applies the archive rule to existing '
                 'findings that meet the archive rule criteria.</p>',
  'name': 'accessanalyzer_apply_archive_rule',
  'parameters': OrderedDict([('analyzerArn', <StringShape(AnalyzerArn)>),
                             ('ruleName', <StringShape(Name)>),
                             ('clientToken', <StringShape(String)>)])},
 {'description': '<p>Cancels the requested policy generation.</p>',
  'name': 'accessanalyzer_cancel_policy_generation',
  'parameters': OrderedDict({'jobId': <StringShape(JobId)>})},
 {'description': "<p>Checks whether the specified access isn't allowed by a "
```

```
            'policy.</p>',
  'name': 'accessanalyzer_check_access_not_granted',
  'parameters': OrderedDict([('policyDocument',
                              <StringShape(AccessCheckPolicyDocument)>),
                             ('access',
                              <ListShape(CheckAccessNotGrantedRequestAccessList)>),
                             ('policyType',
                              <StringShape(AccessCheckPolicyType)>)])},
 {'description': '<p>Checks whether new access is allowed for an updated '
            'policy when compared to the existing policy.</p> <p>You can '
            'find examples for reference policies and learn how to set up '
            'and run a custom policy check for new access in the <a '

'href="https://github.com/aws-samples/iam-access-analyzer-custom-policy-check-samples"
>IAM '
            'Access Analyzer custom policy checks samples</a> repository '
            'on GitHub. The reference policies in this repository are '
            'meant to be passed to the '
            '<code>existingPolicyDocument</code> request parameter.</p>',
  'name': 'accessanalyzer_check_no_new_access',
  'parameters': OrderedDict([('newPolicyDocument',
                              <StringShape(AccessCheckPolicyDocument)>),
                             ('existingPolicyDocument',
                              <StringShape(AccessCheckPolicyDocument)>),
                             ('policyType',
                              <StringShape(AccessCheckPolicyType)>)])},
 {'description': '<p>Checks whether a resource policy can grant public access '
            'to the specified resource type.</p>',
  'name': 'accessanalyzer_check_no_public_access',
  'parameters': OrderedDict([('policyDocument',
                              <StringShape(AccessCheckPolicyDocument)>),
                             ('resourceType',
                              <StringShape(AccessCheckResourceType)>)])}]
```

Here the "name" of each tool is a combination of the tool's service and the operation. E.g., **accessanalyzer_apply_archive_rule** tool above corresponds to the **apply_archive_rule** operation in the `**accessanalyzer**` service.

Update call_tool router method

The call_tool method is now also simpler as it is just a matter of a lookup and an invocation:

```python
def call_tool(tool_name, parameters):
    """Dynamically call a boto3 service operation based on the tool name and parameters."""
    try:
        # Extract the service and operation from the tool name
```

```
        service_name, operation_name = tool_name.split("_", 1)

        # Initialize the boto3 client for the service
        client = boto3.client(service_name, region_name=os.environ.get("AWS_REGION"))

       # Call the operation dynamically
       operation = getattr(client, operation_name)
       response = operation(**parameters)
       return response
   except Exception as e:
       raise ValueError(f"Error executing tool '{tool_name}': {str(e)}")
```

Load tools at the start

Our main loop would need to evolve to load these tools at the start:

```
def main():
    available_tools = generate_tool_definitions()

    while True:
        # get user query

        # Pass generated tools to the query parser
        parsed_query = parse_query_with_tools(llm, user_query, available_tools)

        # Continue as before
```

That's it. Now we have a more "rounded" agent that looks up all available tools to pick the right one to satisfy a particular user query.

Results for the "list my ec2 instances" query:

```
Generating tool definitions...
Welcome to the Cloud Inventory Dashboard! Ask me about your AWS resources.

You: list my ec2 instances

Parsed Query:  {
    "tool": "ec2_describe_instances",
    "parameters": {
      "Filters": [
        {
          "Name": "instance-state-name",
          "Values": ["running"]
        }
```

```
      ]
    }
  }
```

Agent: Using tool 'ec2_describe_instances' with parameters {'Filters': [{'Name': 'instance-state-name', 'Values': ['running']}]}

Agent: The provided output appears to be a response from an AWS SDK or API call regarding the details of an EC2 instance with ID `i-0dddb524e3346e931`. Here's a summary:

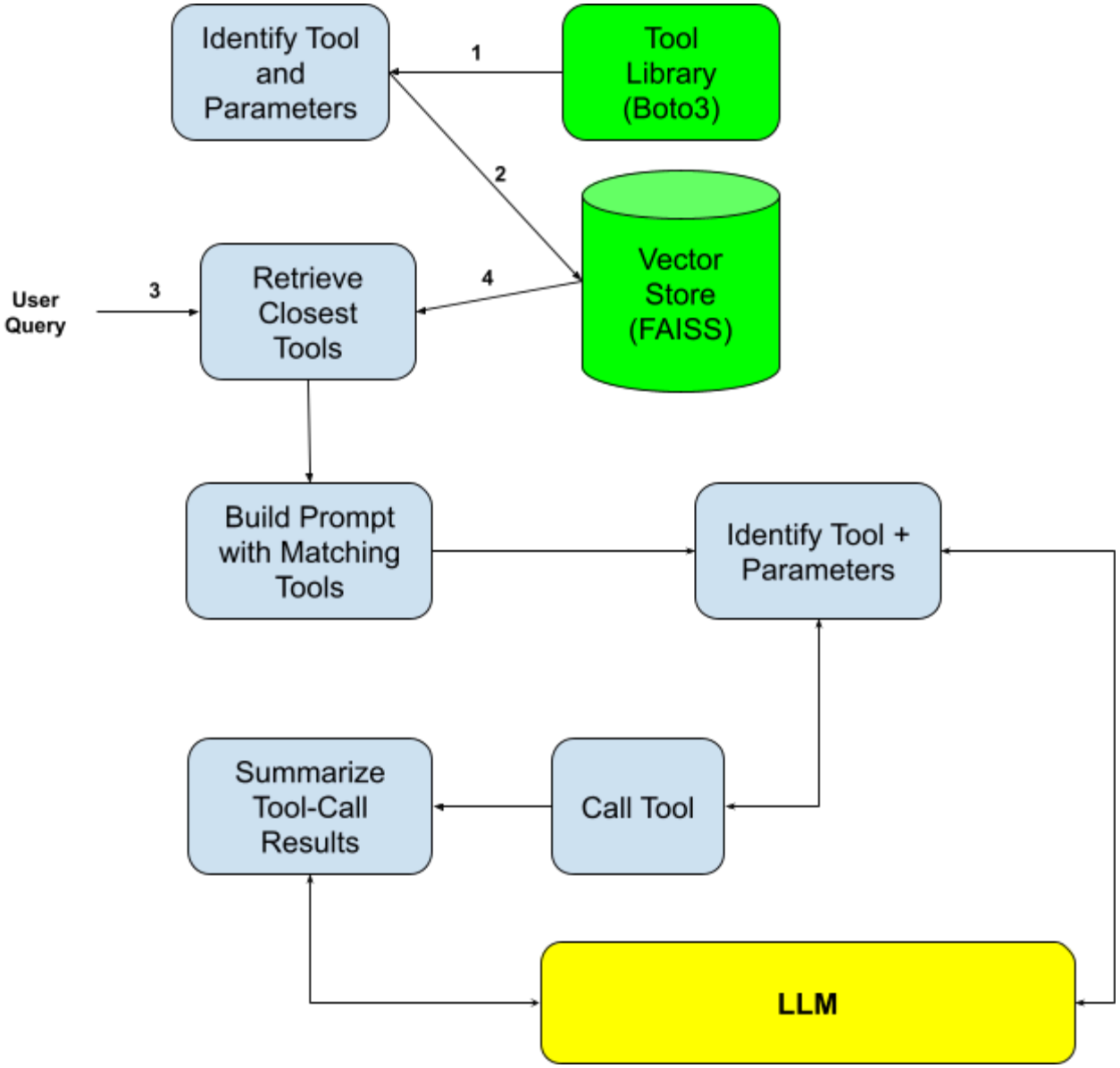
- Instance ID: xxxxxx
- Instance Type: t3.4xlarge
- CpuOptions: 4 cores, 2 threads per core
- CapacityReservationSpecification: open preference
- EnclaveOptions: disabled
- PlatformDetails: Linux/UNIX
- VirtualizationType: hvm
- Monitoring: disabled
- CurrentInstanceBootMode: legacy-bios
- Operator: unmanaged (as there is no EC2 Systems Manager managed instance)
- No tags with specific keys provided in the summary but they are available in the detailed output.

Sadly this is slow. Where the previous query was executed in less than a second, this query took a whopping 14 seconds. The reason was that we are passing a list of ALL the tools - all 16000 of them to the LLM (before we limited this to read-only operations). This is expensive and wasteful. We will fix this next.

Step 6 - Narrow the search with RAG

We can optimize the agent by storing tools in an in-memory vector store and retrieving the top N closest tools based on the user query, we can integrate a vector similarity search mechanism. This will ensure that only relevant tools are passed to the LLM for query parsing. This technique is known as Retrieval-Augmented-Generation (RAG). In a production system, we would be using a more robust and persistent vector store. For our tutorial, we will use a highly performant in-memory vector store - FAISS.

Our setup will be close to:



Install requirements

```
pip install sentence-transformers faiss-cpu
```

The sentence-transformers library will allow us to embed text vectors and faiss will allow fast similarity searches.

Create a Vector Store abstraction

We will create a vector store class that will abstract away the details of working with embeddings and similarity searches etc:

```
import faiss
from sentence_transformers import SentenceTransformer
import numpy as np

class VectorStore:
    pass
```

We will add key methods to the vector store one by one.

Load embedding Models

The VectorStore class first loads an embedding model - [all-MiniLM-L6-V2](). Embedding models allow us to convert text into floating point vectors. These vectors are crucial for performing similarity searches:

```
class VectorStore:
    ...

    def __init__(self):
        print("Loading embedding model....")
        # Load the SentenceTransformer model for embedding
        self.embedding_model = SentenceTransformer('all-MiniLM-L6-v2')  # Lightweight and
efficient

        # Initialize the FAISS index
       print("Initializing FAISS Index....")
       self.index =
faiss.IndexFlatL2(self.embedding_model.get_sentence_embedding_dimension())
       self.tool_embeddings = []  # To store embeddings
       self.tools = []
```

We also keep track of the tools and their embeddings that will be used later.

Enabling adding of tools

The member method method **add_tools** lets us add tools into our vector store:

```
class VectorStore:
    ...

    def add_tools(self, tools):
        """Build an in-memory vector store for tool descriptions."""
        # Embed tool descriptions
        print(f"Adding {len(tools)} tools...")
```

```
        descriptions = [f"{tool['name']}: {tool['description']}" for tool in tools]

        embeddings = []
        batch_size = 1000
        for i in range(0, len(descriptions), batch_size):
            print("Processing Batch: ", i)
            batch = descriptions[i:i + batch_size]
            batch_embeddings = self.embedding_model.encode(batch, convert_to_numpy=True)
            embeddings.extend(batch_embeddings)

        # Add embeddings to the FAISS index
        self.tools.extend(tools)
        self.tool_embeddings.append(embeddings)
        self.index.add(np.array(embeddings))
```

Something to think about for the future. Loading and managing embeddings in a vector store is a whole area on its own. We clearly do not want to load them on every start of the agent. Instead, an external store (eg Elastic, pg_vector, etc.) would be highly desirable - along with an ingestion pipeline where new embeddings (for new tools) can be added in parallel. This tutorial will NOT cover those topics.

Tool retriever

Given a query, we need a method to find tools most related to this query. Let us add the retrieve_relevant_tools member method to this. It allows us to retrieve tools that are most similar to "intent" with the query based on a vector search:

```
class VectorStore:
    ...

    def retrieve_relevant_tools(self, query, top_n=5):
        """Retrieve the top N relevant tools for a user query."""
        query_embedding = self.embedding_model.encode(query, convert_to_numpy=True)
        distances, indices = self.index.search(query_embedding.reshape(1, -1), top_n)

        # Get the closest tools
        relevant_tools = [self.tools[i] for i in indices[0]]
        return relevant_tools
```

Add a prompt-builder

Finally, we need a prompt builder that combines this all with the user query that can then be passed to the LLM like our previous examples:

```
class VectorStore:
    ...
```

```
    def prompt_for_query_with_tools(self, user_query, top_n=5):
        """Parse user query and select the appropriate tool using Mistral."""
        relevant_tools = self.retrieve_relevant_tools(user_query, top_n)
        tools_description = "\n".join(
            f"- Tool: {tool['name']}\n  Description: {tool['description']}\n  Parameters:
{tool['parameters']}"
            for tool in relevant_tools
        )

        prompt = f"""
You are an intelligent assistant for AWS cloud management. Your task is to interpret the
user's query and select the most appropriate tool from the following list:

{tools_description}

User Query: "{user_query}"

Return a JSON object with:
- tool: The name of the tool to use.
- parameters: The parameters required for the selected tool.
"""
        return prompt
```

Putting it all together

Now we are ready to put this all together. All that is needed now is to change the main runloop. Let us see the full version of this loop with all the changes and comments inline:

```
1 def main():
2     # First get a list of all the available tools
3     available_tools = tools.generate_tool_definitions()
4
5     llm = LLM()
6
7     # Create the vector store and add all tools' embeddings into it
8     vs = rag.VectorStore()
9     vs.add_tools(available_tools)
10
11    def parse_query_with_rag(user_query):
12        prompt = vs.prompt_for_query_with_tools(user_query)
13        print("RAG Prompt: ", prompt)
14        return llm.call(prompt)
15
16    print("Welcome to the Cloud Inventory Dashboard! Ask me about your AWS resources.")
17    while True:
18        # User Input
19        user_query = input("\nYou: ")
20        if user_query.lower() in ["exit", "quit"]:
21            print("Goodbye!")
22            break
```

```
23
24        try:
25            # Call the vector store for a prompt for the given query and send that to the LLM
26            parsed_query = parse_query_with_rag(user_query)
27            print(f"\nParsed Query: {parsed_query}")
28            try:
29                parsed_result = json.loads(parsed_query)
30            except Exception as e:
31                print("Query is not json: ", e)
32
33                continue
34
35            tool_name = parsed_result.get("tool")
36            parameters = parsed_result.get("parameters", {})
37
38
39            print(f"\nAgent: Using tool '{tool_name}' with parameters {parameters}")
40
41            # Call the appropriate tool
42            tool_output = tools.call_tool_dyn(tool_name, parameters)
43            summary = summarize_data(llm, tool_output, tool_name)
44            print(f"\nAgent: {summary}")
45        except Exception as e:
46            print(f"\nAgent: Sorry, something went wrong. {str(e)}")
47            raise e
```

There are 3 main (but simple) changes:

1. **LINE 3**: Like before we load the generated tools
2. **LINE 11**: We added a helper method that takes the user query, passes it to the VectorStore, and generates a customized prompt that incorporates the user query and the top 5 most closely related tools.
3. **LINE 26**: The helper method is called to parse the user query.

That's it. Rest is business as usual.

While there is a one-off slight delay initially in loading and vectorizing the tools, the rest of the user interactions are snappy.

Extensions and Future Work

Our AI agent is just the beginning. Here's where we're headed next:

- **Multi-Cloud Support**: Extend the agent to interact with GCP and Azure APIs.
- **Deeper Observability**: Integrate with monitoring tools like Prometheus and Datadog for richer insights.
- **Proactive Recommendations**: Use machine learning to suggest optimizations (e.g., resizing instances, cost savings).

- **Streamlined Deployment**: Package the agent as a serverless function or Kubernetes operator.
- **Evals**: To ensure that as models change or as the agent evolves, guarantees on performance are tracked.

Conclusion

Managing cloud complexity doesn't have to be daunting. By leveraging the power of open-source AI agents, we can democratize access to cloud insights, reduce cognitive load, and empower teams to focus on what matters most: innovation.

Our journey shows how combining AI, open source, and a bit of creativity can transform cloud operations. Whether you're a seasoned cloud architect or a developer just starting out, this is the perfect time to explore how AI agents can simplify your workflows.

References

1. https://mistral.ai/news/announcing-mistral-7b/
2. https://ollama.com/
3. https://github.com/facebookresearch/faiss
4. https://boto3.amazonaws.com/v1/documentation/api/latest/index.html
5. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
6. https://en.wikipedia.org/wiki/Retrieval-augmented_generation

Author Bio

Sriram Panyam is a seasoned Software Engineering Leader with deep expertise in AI, Cloud Computing, and SaaS platforms. As the Chief Architect of Omlet Inc., he drives innovation in open observability through a cutting-edge platform built on OpenTelemetry. Sriram has authored numerous insightful articles published in *DZone*, *IEEE Xplore*, and the *Forbes Technology Council*, and serves as an active reviewer for esteemed conferences and journals, including *IEEE Engineering Management Review* and *Computer Magazine*.

Sriram is a staunch advocate for open source. As part of the LF AI & Data, Generative AI Commons Education and Outreach Committee, Sriram is equally passionate about mentoring and coaching engineers and managers, empowering them to reach their full potential and advance their careers. His leadership philosophy emphasizes technical excellence, collaboration, and continuous learning, making him a respected voice in the tech community.