READ MODE · ANSWER FIRST

Build a Data Analysis Agent

An interactive, visual explanation of Build a Data Analysis Agent, from setup through the complete system.

CHEAT SHEET · 6 ESSENTIAL IDEAS

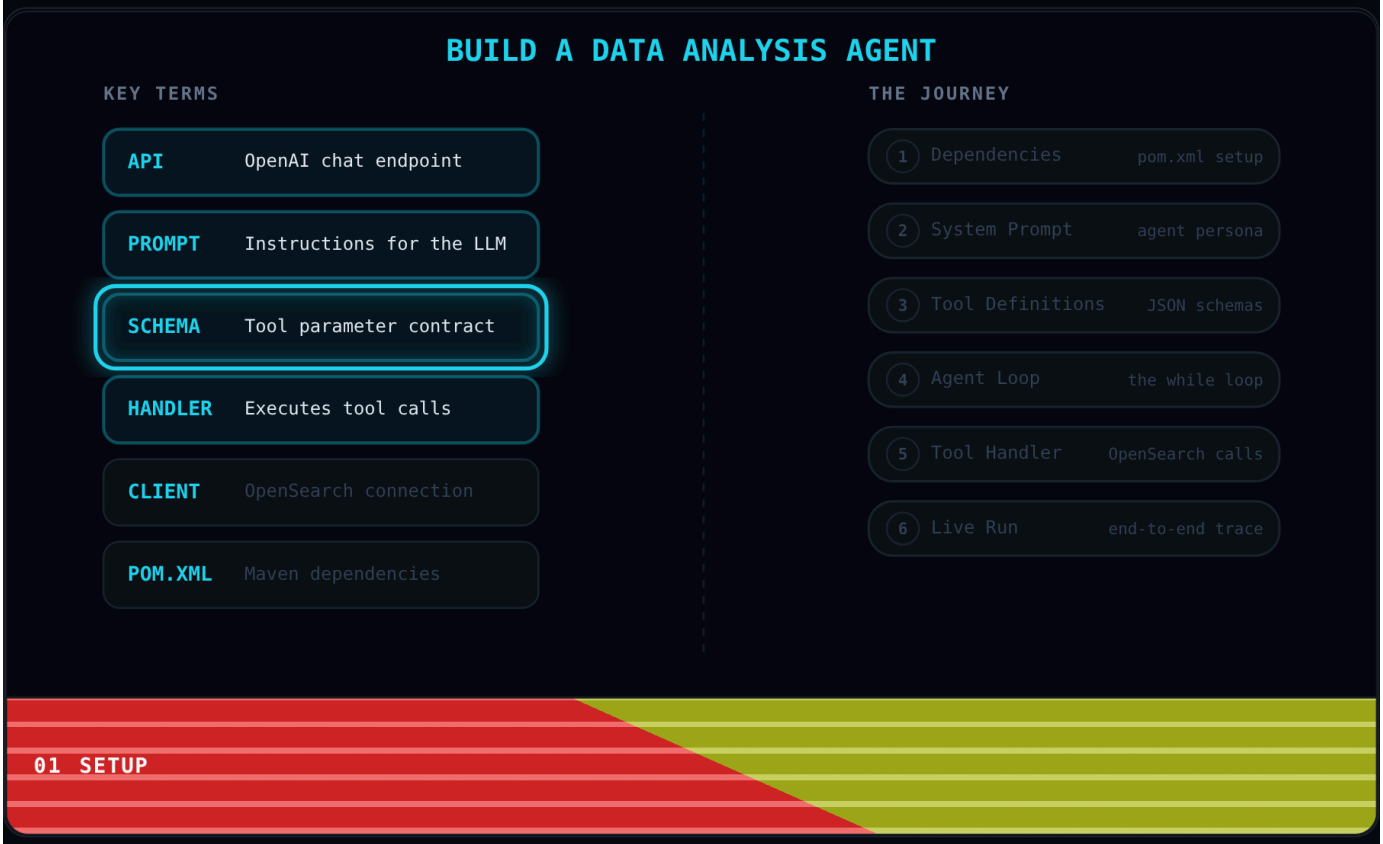
The whole story in 6 lines

Build a Data Analysis Agent becomes easier to reason about when every stage is connected as one system.

1. Three dependencies is all it takes. Jackson handles the JSON for API calls, the OpenSearch client handles the data side, and MCP bridges...
2. A good system prompt turns a general-purpose LLM into a focused specialist. Ours tells the model it is a data analyst with access to...
3. Each tool definition is a contract. The name and description tell the LLM when to use it.
4. The entire agent loop is about 20 lines of Java. It is the same think-act-observe cycle from Part 1, translated directly into code.
5. The handler is where the agent touches the real world. It translates the LLM's abstract tool call into a concrete OpenSearch query.
6. The terminal trace maps directly to the agent loop: each iteration is visible as a think-call-result block.

STAGE 1 OF 8

Setup



Welcome to Part 2. In Part 1, you learned the concepts: the agent loop, the harness, MCP, and tool call flow. Now we turn those concepts into working Java code. By the end of this tutorial, you will have a complete data analysis agent.

For GPT-5.4, your agent should call the OpenAI Responses API over HTTP. Java's built-in HttpClient sends JSON requests and receives JSON responses. No SDK needed — just raw HTTP with the right headers and body.

A system prompt is a string you write that tells the LLM who it is and how to behave. For our agent, it says: "You are a data analyst with access to OpenSearch." The prompt shapes every response the model produces.

A tool schema is a JSON definition that describes what a tool can do: its name, description, and what parameters it accepts. The LLM reads these schemas to decide which tool to call and with what arguments.

The tool handler is the Java method that actually does the work when the LLM calls a tool. It receives the tool name and arguments, calls OpenSearch, and returns the result. This is where your agent connects to the real world.

Everything starts with pom.xml, the Maven file that declares your project dependencies. Three libraries are all you need: Jackson for JSON, the OpenSearch client, and the MCP Java SDK. Let us start by setting up the project.

STAGE 2 OF 8

Dependencies

02 DEPENDENCIES

Now that we know what we are building, let us set up the project. Every Java application starts with declaring its dependencies. We need three libraries, and each one maps to a concept from Part 1.

First dependency: Jackson for JSON. Our agent makes raw HTTP calls to the OpenAI API, so we need to build JSON request bodies and parse JSON responses. Java's built-in HttpClient handles the HTTP part — no extra library needed.

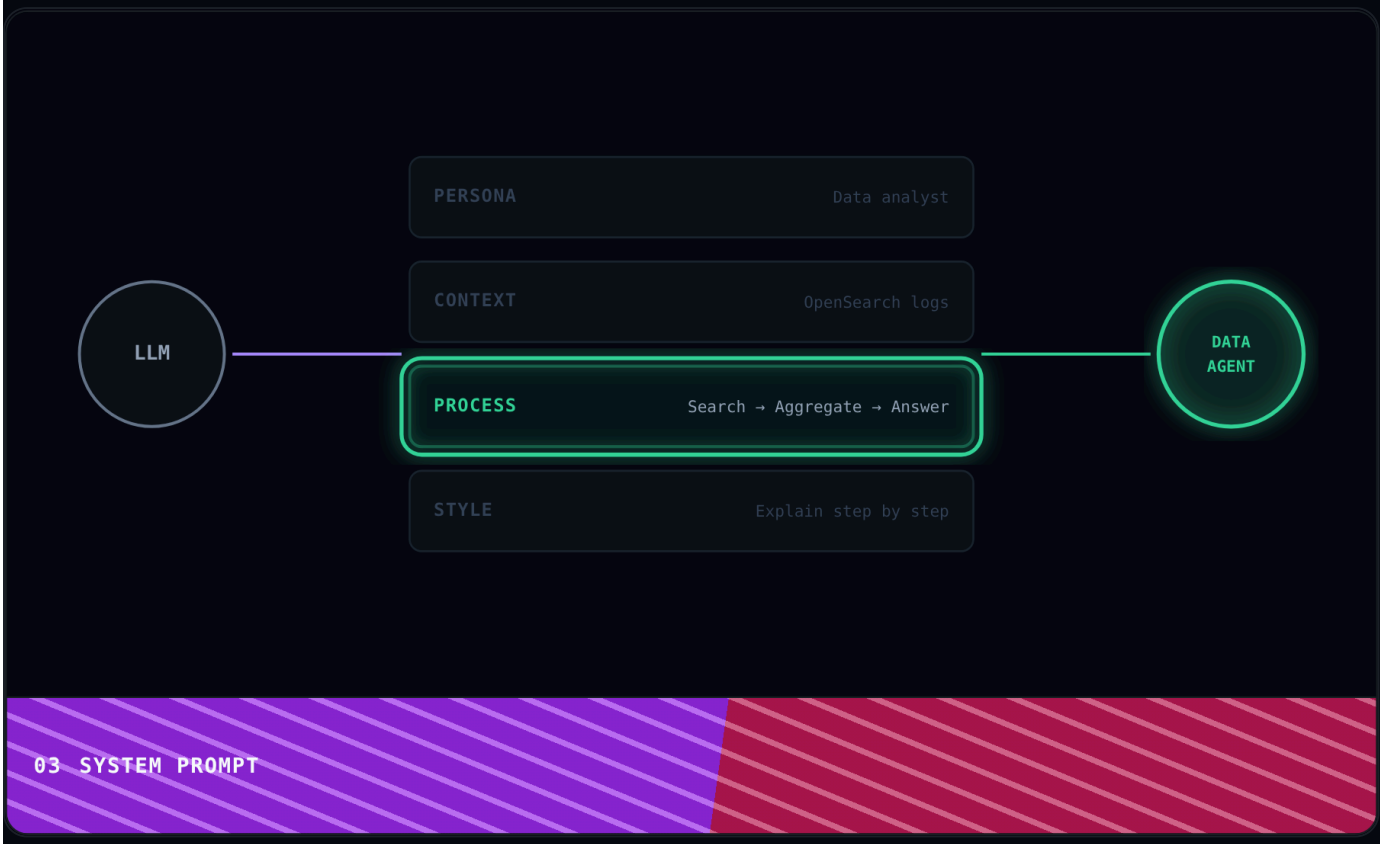
Second: the OpenSearch Java client. This is how your agent actually queries data. When the LLM decides to search for error logs, this client is what executes that search against a real OpenSearch cluster.

Third: the MCP Java SDK. Remember how MCP is like USB for AI tools? This SDK implements that protocol so your agent can expose and discover tools in a standard way.

That is it. Three dependencies, three capabilities: handle JSON for the API, query the data, and wire tools through a protocol. The project structure is simple: one main class for the agent, one class for the tool handler. Next, we write the system prompt that tells the LLM what kind of agent it should be.

STAGE 3 OF 8

System Prompt



03 SYSTEM PROMPT

We have our dependencies. Now we write the first piece of actual agent code: the system prompt. Remember from Part 1 that the harness builds an initial message list? The system prompt is the very first message in that list.

The opening line defines the persona. "You are a data analyst agent." This single sentence shapes how the model approaches every question. It will think like an analyst: look for patterns, quantify things, and explain findings clearly.

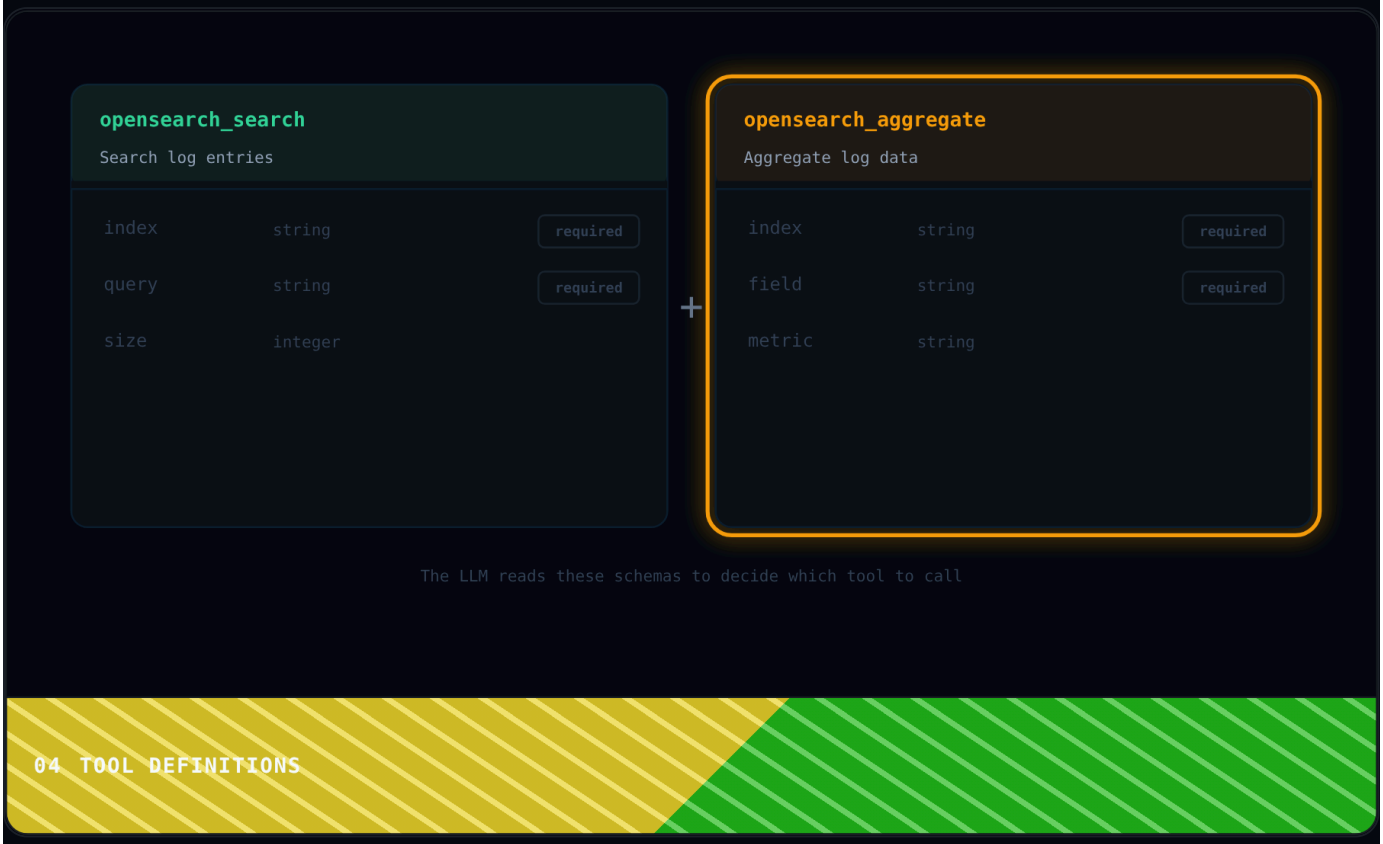
Next we describe what the agent has access to. "You have access to an OpenSearch cluster containing application logs." This tells the model what data exists, so it knows what kind of queries make sense.

The instructions give the model a clear process: search for relevant entries, aggregate data to find patterns, then synthesize a clear answer. This three-step recipe matches the tool call sequence we will build. Try the Persona control to see how a DevOps prompt differs from an analyst prompt.

The final instruction asks the model to explain its reasoning. This is not just for the user. It helps during debugging. When the agent makes a bad tool call, the reasoning shows you why. The system prompt is done. Next, we define the tools it can use.

STAGE 4 OF 8

Tool Definitions



04 TOOL DEFINITIONS

The system prompt tells the model what it is. Tool definitions tell it what it can do. Remember from Part 1 how MCP servers advertise their tools with schemas? We are building those schemas now, one for each OpenSearch operation.

The search tool has three parameters: index (which OpenSearch index to search), query (the search string), and size (maximum results). The name and description are critical. The LLM reads them to decide whether this tool fits the user's question.

Notice the required array: index and query must always be provided. Size is optional and defaults to 10. This tells the LLM it cannot call the search tool without specifying what to search for and where.

The aggregate tool works on grouped data: count errors by type, average response time by endpoint. Its parameters include index, field (what to group by), and metric (count, avg, sum). Switch the Tools control to Search Only to see a simpler single-tool setup.

Two tools give the agent its full analytical power. Search finds specific entries. Aggregate reveals patterns. Together they cover most data analysis questions. The LLM decides which to call and in what order. Now let us write the loop that orchestrates it all.

STAGE 5 OF 8

Agent Loop

05 AGENT LOOP

We have the prompt and the tools. Now we write the code that brings them together. Remember the harness from Part 1? This is that harness, written in Java. The entire agent loop fits in about 20 lines.

The loop starts by creating an input list with the user's question, plus a previousResponseId variable. After the first request, GPT-5.4 can continue the tool workflow by chaining each turn through previous_response_id.

Each iteration builds a JSON request body for the Responses API with the current input items, the system prompt, the tool definitions, and optionally previous_response_id. The response comes back as JSON with output items.

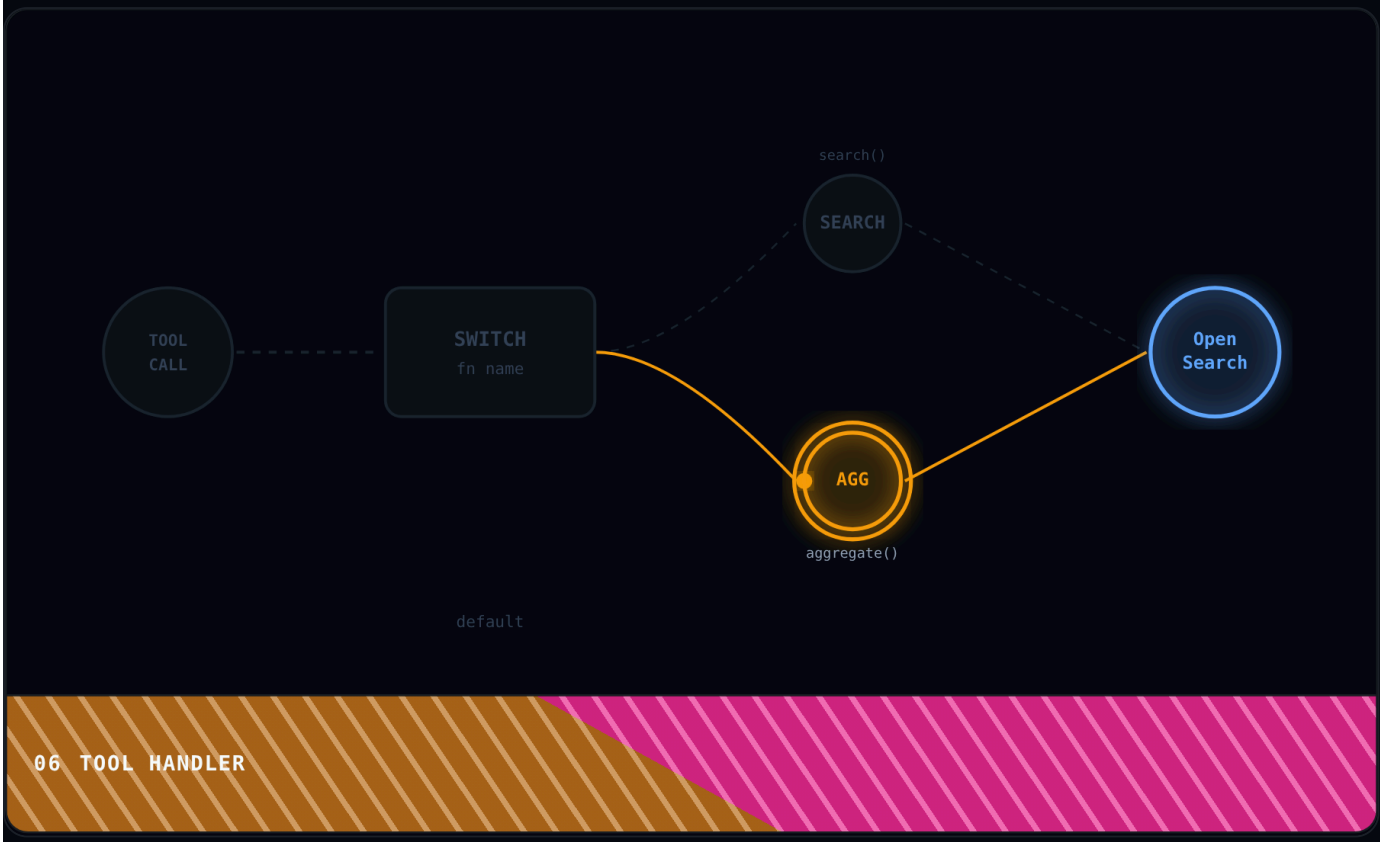
After each response, we look through output for function_call items. If there are none, the model produced its final text answer and the loop exits. This is the model saying "I have enough information now." Try changing Max Iterations to see the safety limit change.

If the response includes function_call items, we iterate over each one, execute it, and send the results back as function_call_output items tied to the model's call_id. Then the loop continues.

That is the complete agent loop. Build a request, send it over HTTP, check for tool calls, execute them, loop. The simplicity is intentional. A few helpers handle the JSON formatting. The LLM handles the reasoning. Your code just drives the cycle. Next, we write the tool handler.

STAGE 6 OF 8

Tool Handler



06 TOOL HANDLER

The agent loop calls executeTool() whenever the model requests a tool. Now we write that method. It receives a JSON node from the API response containing the function name and arguments, and returns a string result.

First we extract the function name and parse the arguments from the JSON. Then a switch expression routes each tool name to its handler. The names must match exactly: "opensearch_search" and "opensearch_aggregate".

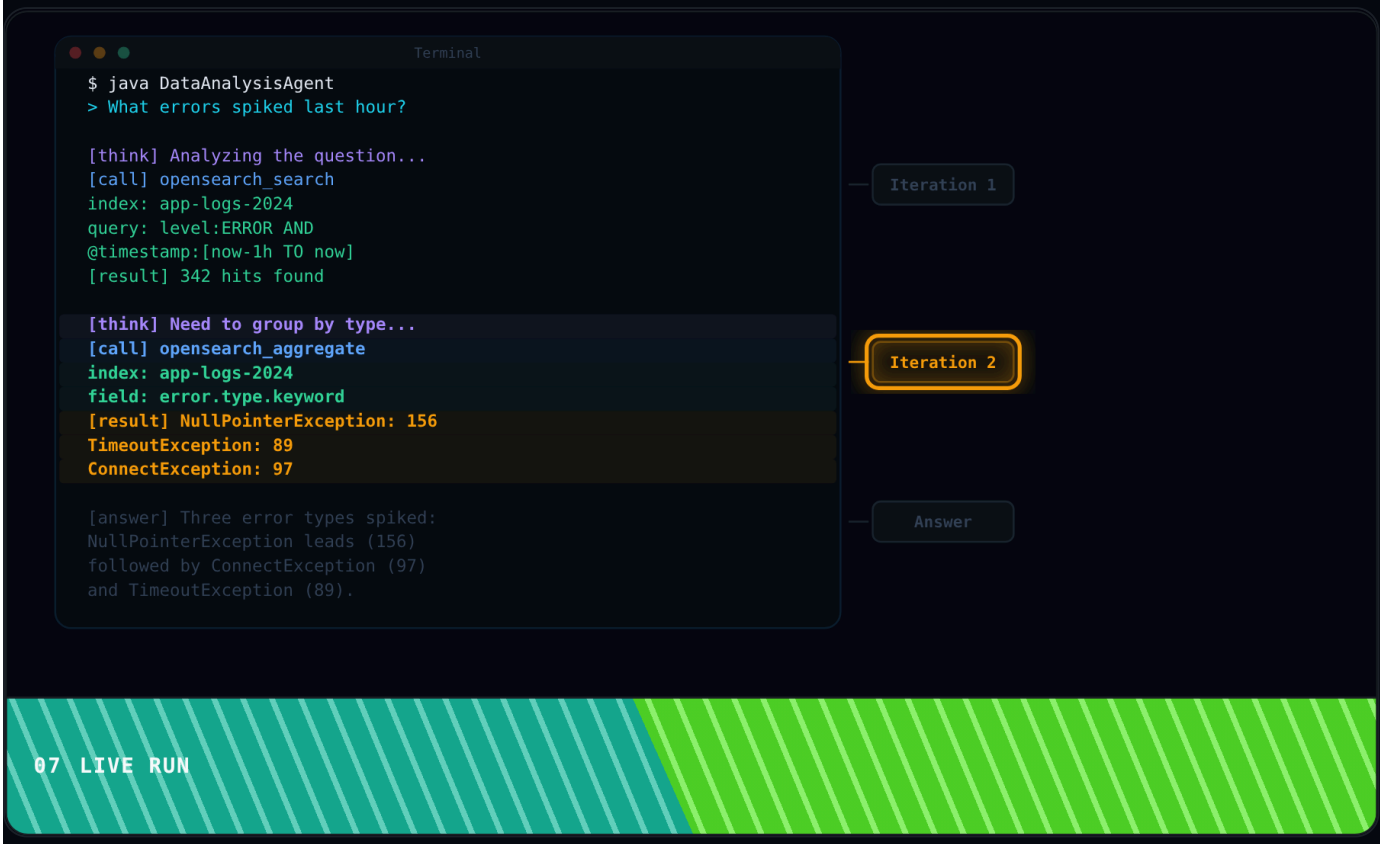
For the search tool, we extract the index and query from the parsed arguments, build an OpenSearch query using the Java client's builder API, execute it, and convert the hits to JSON. This is where your agent actually talks to OpenSearch.

The aggregate case follows the same pattern: extract parameters, build a terms aggregation, execute, and return JSON. Switch the Active Tool control to see each code path highlighted. The two paths share a structure but call different OpenSearch APIs.

The default case handles unknown tool names gracefully. It returns an error message that the LLM can read and reason about. Remember from Part 1 how errors flow back through the conversation just like successes. The handler is complete. Now let us run the whole agent end to end.

STAGE 7 OF 8

Live Run



07 LIVE RUN

Everything we built comes together now. We start the agent and type a question. Watch the terminal output. Each line maps to a concept from Part 1 and a piece of code from the previous stages.

The user asks: "What errors spiked in the last hour?" The agent loop begins. The model reads the system prompt, the tool definitions, and the question. It enters the think phase and decides it needs to search for error logs first.

First tool call: opensearch_search. The LLM provides the index name and a query that filters for ERROR level logs in the last hour. The tool handler executes this against OpenSearch and returns 342 matching hits.

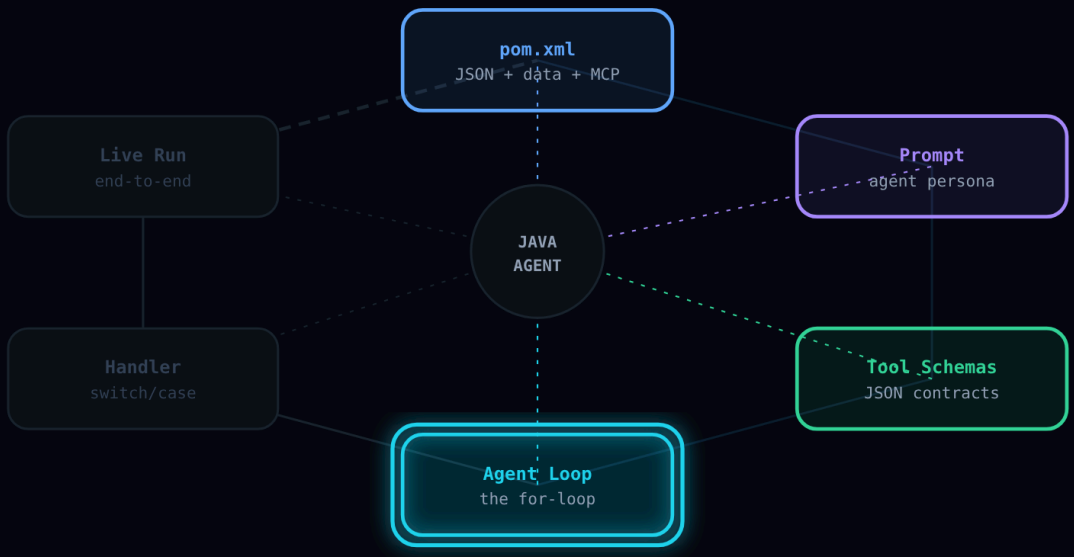
The LLM sees 342 hits but needs to understand the pattern. It calls opensearch_aggregate to group errors by type. This is the second loop iteration. Each call is smarter because the LLM has the previous result in its conversation history.

The aggregation returns three error types with counts. The LLM now has enough information. The stop reason is END_TURN and it produces the final answer. Try switching the Query control to Simple to see a single-step query. Notice how simple queries exit the loop after just one tool call.

That is a working data analysis agent. A system prompt, two tool definitions, a 20-line loop, and a tool handler. The concepts from Part 1 are now running code. Now let us step back and see how all the pieces fit together.

STAGE 8 OF 8

Recap



08 RECAP

We started with pom.xml: three dependencies that give the agent its core capabilities. Jackson for JSON parsing, the OpenSearch client to query data, and the MCP SDK for the tool protocol.

Then we wrote the system prompt: a text block that turns a general-purpose LLM into a focused data analyst. The prompt is the agent's instruction manual, read before every response.

We defined two tools with JSON schemas: opensearch_search and opensearch_aggregate. These schemas are the contract between the LLM and your code. The LLM reads them to decide what to call and with what arguments.

The agent loop tied it together: a 20-line for-loop that sends HTTP requests to the OpenAI API, checks for tool calls, executes them, and repeats. This is the harness from Part 1, written in Java.

The tool handler is where the agent touches the real world: a switch statement that routes tool names to OpenSearch operations and returns results as JSON.

And when we ran it, every piece worked in concert: the loop called the API, the model requested tools, the handler executed them against OpenSearch, and results flowed back until the agent had a complete answer. That is a working data analysis agent in Java. The concepts from Part 1 are now code you can run.