Building CRD-Xray: An AI Agent for Kubernetes Custom Operators

Author: Vedang Joshi
Published: June 11, 2025 · 5 min read

Github

If you're running Kubernetes seriously, chances are you've built a bunch of custom operators for your organization. These operators work wonders. But I've always felt something was missing and that is context. The kind of context that helps new engineers understand what's happening, or helps me debug when things break. That's where the idea for **CRD-Xray** started.

My goal is simple: collect as much useful data from custom operators as possible, feed it into an AI agent, and use that agent to answer questions like, "What does this CRD do?", "Which controller manages it?", or "Why is it failing?"

The 4 Problems I'm Trying to Solve

To explain what CRD-Xray aims to solve, here are four specific pain points I've faced:

1. No Documentation for Custom Operators

Open-source tools from CNCF are well-documented, and you can find answers on forums and StackOverflow. But when it comes to internal, organization-specific operators, well generally that's not a happy path. There's usually zero documentation or outdated README files that don't help.

2. Debugging is a Nightmare

There's no built-in mapping in Kubernetes between a controller and the CRDs it manages. Tracing logs, understanding how different tools interact, and piecing it all together manually is painful. Imagine if we could debug across all tools and logs in one go, I know it sounds far-fetched, but I wanted to try.

3. MCP Integration

Since I'm already maintaining a database of CRDs, their controllers, logs, and resources, this makes a great base layer of context for future AI agents. By exposing it via MCP (Model Context Provider), I could let even smarter agents use this data later on.

4. CRD Metrics

We have Prometheus exporters for most core components, but custom CRDs often go unmonitored. Having a way to track their versions, usage patterns, and metrics across clusters could be super helpful.

Step 1: Data Collection

To solve any of the above, I had to first collect the right data. I started by building a Kubernetes controller using the **Kopf** framework. It runs inside the cluster and periodically collects:

1. All the CRDs installed on the cluster
2. All events involving those CRDs
3. All the CRs (custom resources) for each CRD
4. All pods — this list gets passed to an LLM to figure out which pod might be the controller for a given CRD
5. Logs from the identified controller pod (fetched every hour for now)

I store metadata like CRD schemas, controller names, and resource names in a **SQLite** database. For everything else like logs, manifests, etc. I use **Qdrant** as a vector database.

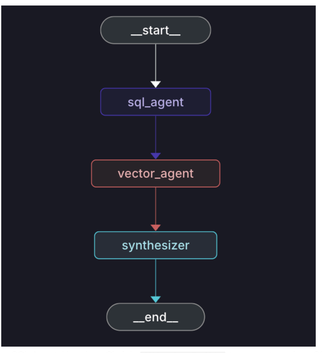
Why Kopf?

Choosing Kopf was an architectural decision. I wanted this controller to live *inside* the cluster. That way, it can continuously monitor and collect data without relying on external scripts or cron jobs. In the future, the idea is that you could just `helm install` this tool, and boom your AI agent is up and running.

Step 2: The LangGraph Agent

Once the controller is up and collecting data, the next step is building an agent that can answer queries. I used **LangGraph** to create a simple state-based AI agent with this flow:

```
user input -> SQL agent -> Vector agent -> Synthesizer -> Final output
```



- **SQL Agent:** Uses LangChain's `SQLDatabaseToolkit` to generate and run SQL queries based on natural language input
- **Vector Agent:** Uses `langchain_qdrant` to search the vector store for relevant logs or manifests
- **Synthesizer:** Combines the results from the SQL and vector search to generate a final answer using Claude (via `langchain_anthropic`)

It's surprisingly effective for questions like:

- "How many CRs exist for `FooBar`?"
- "What does the controller for `FooBar` do?"
- "Show me logs from the controller when resource X failed."

The Agent in Action

The whole point of combining structured (SQL) and unstructured (logs) data is to get deeper, more helpful answers. The agent takes in the query, runs SQL to extract basic info, does a vector search using that context, and then synthesizes a human-readable answer. It's not perfect yet, but it's already saving me time.

What Didn't Work (Yet)

I also played around with **LangGraph Studio** and **LangSmith**. While both are promising, they still need quite a bit of manual setup and tuning. I've shelved that for now, but will revisit once the core system is solid.

What's Next?

There's a lot I want to improve:

- Make the agent more robust and conversational
- Build a proper test suite
- Create a Helm chart for easy deployment
- Test it on more and more custom CRDs in the wild

This is still very much a work in progress, but I'm excited about where it's headed. If you've dealt with the same pain points, or just find this idea interesting, I'd love to hear your thoughts.

Thanks for reading!

AI-Agent Langgraph Kubernetes