Kubernetes AI Agent - Technical documentation

Kubernetes AI Agent - Backend

The Full Stack FastAPI frontend documentation can be found down below

Code Structure

The backend code is structured as follows:

- `backend/app/alembic` - The directory with the Alembic migrations.
- `backend/app/api` - The different API endpoints of the backend.
- `backend/app/core` - The core functionality of the backend, such as the database connection and the security.
- `backend/app/monitoring_agent` - The LLM agent using LangGraph.
- `backend/app/tests` - The tests of the backend.
- `backend/app/web` - The WebSocket connection to retrieve runs in real-time.
- `backend/crud.py` - The CRUD (Create, Read, Update, Delete) utils.
- `backend/main.py` - The FastAPI application creation and configuration.

Monitoring Agent

The monitoring agent code is in the `backend/app/monitoring_agent` directory. It is developed using the LangGraph library.

Code Structure

The agent code is structured as follows:

- `backend/app/monitoring_agent/agent.py` - The agent class.
- `backend/app/monitoring_agent/agent_nodes.py` - The agent nodes.
- `backend/app/monitoring_agent/edge.py` - The router for edges routing.
- `backend/app/monitoring_agent/llm.py` - The LLM class to create the LLM client.
- `backend/app/monitoring_agent/main.py` - The main file to run the agent workflow.

- `backend/app/monitoring_agent/prompts.py` - The prompts for the agent.
- `backend/app/monitoring_agent/state.py` - The state class.

State

A AgentState is a class that inherits from `TypedDict` which allows defining dictionary types with specific keys and values. It contains a list of messages and a sender. This state will then be passed through the workflow of the agent. Each agent will then add messages to the state and pass it to the next agent.

The `operator.add` method is used to add messages to the state. This method is called by the framework to add a message in the list.

The `sender` attribute is used to store the sender of the message. This is useful to know to which node respond to when a tool call is made so the response can be sent to the correct node.

Here is an example of a state :

```
{'metric_analyser':
    {'messages': [
        AIMessage(id='run-34618012-fba9-4380-a745-346b4a2ac44e-0',
                  name='metric_analyser',
                  content='',
                  tool_calls=[{'name': 'get_pod_names', 'args': {'namespace': 'testing-apps'
                  additional_kwargs={'tool_calls': [
                      {'id': 'call_Y8mXlniyV8sdV28AHnk55mjI',
                       'function': {'arguments': '{"namespace": "testing-apps"}', 'name': 'g
                       'type': 'function'}
                  ]},
                  response_metadata={
                      'token_usage': {'completion_tokens': 34, 'prompt_tokens': 1005, 'tota
                      'model_name': 'gpt-4o-2024-05-13', 'system_fingerprint': 'fp_400f27fa
                      'finish_reason': 'tool_calls', 'logprobs': None
                      },
                  usage_metadata={'input_tokens': 1005, 'output_tokens': 34, 'total_tokens':
        'sender': 'metric_analyser'
    }
}
```

Nodes

Nodes are created in the `backend/app/monitoring_agent/agent_nodes.py` file.

The `parse_config` method is used to parse the prompts to a single string. The promps are defined in the `prompts.py` file and are a dictionary with the name of the prompt as the key and the prompt is defined with a role, a goal, a backstory, a description, an expected_output and a list of examples.

The `agent_node` method is used to create a node. It takes a state, an agent and a name. This method will be called by the framework to execute the given task to the agent. Agents are created with the `functools.partial` which partially creates the object. Firstly nodes are created with the `agent_node` method and only agent and name are passed. The state is passed when the node is executed. This mechanism is needed as we need to compile the graph to validate the nodes and the edges but the state is not available at this time.

The agent objects needed to the previous function are made with the `create_agent` method. It takes an LLM object (which currently is either ChatOpenaiAI, OllamaFunctions or Ollama), a list of available tools and the system_message. This method defines the main prompt explaining the overall goal of the agent, the available tools and the system_message which is the task to be executed by the agent.

Edges

The agent only uses conditional edges wich enables the agent to choose the next node to execute based on the response of the previous node. The router is defined in the `backend/app/monitoring_agent/edge.py` file. It takes the state, select the last message and firstly checks for the `tool_calls` attribute. If it is present, the router will return the node with the name of the tool call. Otherwise, it will continue to check for keywords such as "DIAGNOSTIC NEEDED" or "GENERATE SOLUTIONS" and return the keyword "continue". If keywords "UNSUCCESSFUL" or "FINISHED" are present, the router will return the keyword `__end__`. There only keywords are used. The next node is then defined during the graph generation.

Flow

In the `generate_graph` method in the `main.py` file, We define the workflow using the `StateGraph` with the `AgentState` in parameter.

In this worflow each node created with the `agent_node` method is added to the graph using the `add_node` method. It takes in parameter a string for the name of the node and the agent object. The tool_node is created using the `ToolNode` method and taking a list of tools in parameters.

Then the edges are added to the graph using the `add_conditional_edges` method. It takes in parameters the name of the node, the router function, and a mapping dictionnary with the next node to execute based on the router output.

```
workflow.add_conditional_edges(
        "metric_analyser",
        router,
```

```
        {"continue": "diagnostic", "call_tool": "call_tool", "__end__": "incident_reporter"}
    )
```

For the `call_tool` node, to define the next node to execute, the function retrieves the sender name of the current state. The path map is defined in the `path_map` dictionnary. The next node to execute is then defined based on the sender name to return the tool response to the correct node.

```
workflow.add_conditional_edges(
        "call_tool",
        lambda x: x["sender"],
        {
            "metric_analyser": "metric_analyser",
            "diagnostic": "diagnostic",
            "__end__": "incident_reporter"
        },
    )
```

As all nodes and edges are defined, one last edge needs to be defined to indicate the end of the workflow. This is done with the `add_edge` which creates a normal edge between `incident_reporter` and the `END` node which is a node provided by the framework.

With the `set_entry_point` method, the entry point of the workflow is defined. The entry point is the first node to be executed. In this case, it is the `metric_analyser` node.

Finally, the workflow is compiled to check if all nodes and edges are correctly defined.

Run

To run the agent, the `run` method is called. It takes a WebSocket manager to send json to users, a session to save the events in the database and a run id.

The `run` method initialise the graph, defined the namespaces to monitor and the input message.

Then the graph can be executed using the graph.astream method which will stream all events that occurses during the execution of the graph. The `stream_mode` is defined to `Updates` which will return an event as soon as it is generated. Another option is to use `Values` which will return the final state of the graph.

Each event generated by the graph is then parsed to json and sent to the user using the WebSocket manager as well as saved in the database. When the workflow is finished, the status of the run is modified to `finished`. If any exception occurs during the execution of the graph, the status is modified to `failed` and the exception is saved in the database.

Setting the run status and creating an event is made using the `set_run_status` and `create_event` methods defined in the `crud.py` file. These methods are used to interact with the database using the session in the parameters which allows to save the events and the run status in the database.

Tools

The tools are defined in the `backend/app/monitoring_agent/tools` directory.

The tools for Kubernetes are defined in the `kubernetes_tool.py` file.

To access the Kubernetes cluster, the KubernetesConfig class has been created in the `backend/app/monitoring-agent/config/k8s_config.py` file. This class is used to create a Kubernetes client using the `kubernetes` library. The `authenticate`method is used to read the service account file and create a client. The `get_client` method is used to authenticate the client if it is not already done and return the client. For a better security the authentication should use the SSL certificate.

Another similar class has been created to access Googl Cloud Logs. The `GoogleCloudLogging` class uses the `google-cloud-logging` library to create a client. The `get_client` method is used to authenticate the client if it is not already done and return the client. The authentication is using the service account file.

The tools created for Kubernetes are * `list_namespaced_pod` : List all pods in a namespace. * `get_pod_resources` : Get the resources of a pod. * `get_nodes_resources` : Get the resources of all nodes. * `get_pod_logs` : Get the logs of a pod. * `get_pod_yaml` : Get the yaml of a pod.

When creating a tool that can be used with the LLM agent, the functions should be annotated with the `@tool` decorator and containing a description of the tool in the comments. The framework will use the name, the paramteres, the return type and the description of the tool to parse it in json and send it to the LLM.

The `get_pod_logs` tool is a little bit different. All other tools take a defined number of parameters with specific types. This function takes a logs filter which is of string type. But the format of the string is very specific as it should match the Google Cloud Logging filter format.

Here is an example of Google Cloud Logging filter format :

```
resource.type="k8s_container"
resource.labels.project_id="plenary-stacker-422509-j4"
resource.labels.location="europe-west6-a"
resource.labels.cluster_name="gke-monitoring-agent"
resource.labels.namespace_name="boutique"
labels.k8s-pod/app="adservice"
severity>=DEFAULT
timestamp>="2024-07-08T16:41:00Z"
```

`timestamp<="2024-07-08T16:42:00Z"`

For Prometheus tools, the `prometheus_tool.py` file has been created.

The `execute_prometheus_query` tool is also special. The string query passed in parameter should follow the PromQL format. The function will then execute the query and return the result in a json format. This is interesting as the LLM will generate the query and the agent will execute it. The Prometheus queries are executed with the PrometheusQuery client. During many tests, there was a need to sanitize the query to avoid issues with the Prometheus API. The sanitization simply removes double backslashes which are not needed in the query.

Another tool was created only for the use to get HTTP metrics for the `sock-shop` test app. The query is hardcoded using the name `request_duration_seconds_count` which is the metric used to get the number of requests. This tool was created when `execute_prometheus_query` was not able to use this metric as its name is dedicated to the `sock-shop` app.

LLM

In the `llm.py` file the functions `get_llm` retrieves the environment variable `LLM_MODEL` and creates a client using the `ChatOpenaiAI` class if the LLM starts with `gpt` for `gpt-4o` and `gpt-3.5-turbo` models. Otherwise, if the model is Llama3, we create an `OllamaFunctions` if tools are available or an `Ollama` client if no tools are available. Currently, using Ollama LLM is not working as wanted which may be caused by the `OllamaFunctions` which is still an experimental feature.

When developing the agent, the `OllamaFunctions` has an issue parsing the tools which forced to create a parsing method for tools. This method is defined in the `tool_binder.py` file. It takes a function in the parameter and returns the name of the function, the parameters and the description of the function.

DB

Models

The models are defined in the `backend/app/models.py` file. The models are defined using the SQLModel library which is a library to define SQL tables using Python classes.

The two tables that are created are the `AgentRun` and the `Event`. The `AgentRun` table is used to store the runs of the agent. The `AgentEvent` table is used to store the events of the agent.

The id of both tables are UUID fields. The `event_data` from the `Event` table is a JSON field which stores the event data in a JSON format. The `run_id` is a foreign key to the `AgentRun` table. The `status` field is an Enum field which can take the values `running`, `finished` or `failed`.

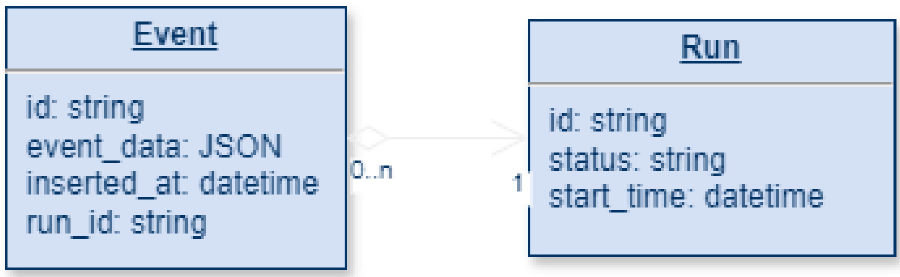

Figure 1: Database schema for agent runs and events

Here is the example of the event in JSON format detailed in the State section :

```
{
  "metric_analyser": {
    "messages": [
      {
        "id": "run-50a3a733-d765-4d5a-9e35-0fa9d83ea82e-0",
        "name": "metric_analyser",
        "type": "ai",
        "content": "",
        "tool_calls": [
          {
            "name": "get_pod_names",
            "args": {
              "namespace": "boutique"
            },
            "id": "call_JJ4WETCdsEMXMKPmmFVqaD90"
          }
        ],
        ...
      }
    ],
    "sender": "metric_analyser"
  }
}
```

Other models where created to match the data of the agent that need to be send to the frontend. The `AgentRunAndEventsPublic` model is used to store the data of a run and its events. The `AgentRunPublic` model represents a single run without its events. The `AgentRunsPublic` model is used to store a list of runs without their events.

CRUD

The CRUD operations are defined in the `backend/app/crud.py` file. The `create_run` method is used to create a run. It is used when the user asks for an agent run. The `create_event` method is used to create an event. It is used to save in the database all events produces by the workflow and different tasks. The `get_run_events` method is used to get all events of a run. It is used to return to the frontend all events of a run. The `set_run_status` method is used to set the status of a run. It it used when the run is finished or failed.

API

The API is built with FastAPI. In `backend/app/api` you can find the different endpoints of the API. Routes concerning the agent are in the `agent.py` file. There are 3 routes in this file:

- `POST /agent/run` - To run the agent.
- `GET /agent/runs` - To get the list of runs.
- `GET /agent/run/{run_id}` - To get the details of a run.

The route for websocket is in the `main.py` file. The route is `/ws` and is used to connect to the websocket to get the events in real-time. It was placed in this file as it would not be possible to connect to the websocket if it was in the `agent.py` file.

All these routes are only available if the user is authenticated. The verification is made by passing the `CurrentUser` object to the route. This object is created by the `get_current_user` method in the `backend/app/api/deps.py` file. This method uses the `OAuth2PasswordBearer` to get the token from the request and then the `jwt.decode` method to decode the token and get the user information.

All available routes are documented using Swagger UI. You can access the documentation at `http://localhost:8080/docs`.

FastAPI Project - Backend

Requirements

- Docker.
- Poetry for Python package and environment management.

Local Development

- Start the stack with Docker Compose:

```
docker compose up -d
```

- Now you can open your browser and interact with these URLs:

Frontend, built with Docker, with routes handled based on the path: http://localhost

Backend, JSON based web API based on OpenAPI: http://localhost/api/

Automatic interactive documentation with Swagger UI (from the OpenAPI backend): http://localhost/docs

Adminer, database web administration: http://localhost:8080

Traefik UI, to see how the routes are being handled by the proxy: http://localhost:8090

Note: The first time you start your stack, it might take a minute for it to be ready. While the backend waits for the database to be ready and configures everything. You can check the logs to monitor it.

To check the logs, run:

```
docker compose logs
```

To check the logs of a specific service, add the name of the service, e.g.:

```
docker compose logs backend
```

If your Docker is not running in `localhost` (the URLs above wouldn't work) you would need to use the IP or domain where your Docker is running.

Backend local development, additional details

General workflow

By default, the dependencies are managed with Poetry, go there and install it.

From `./backend/` you can install all the dependencies with:

```
$ poetry install
```

Then you can start a shell session with the new environment with:

```
$ poetry shell
```

Make sure your editor is using the correct Python virtual environment.

Modify or add SQLModel models for data and SQL tables in `./backend/app/models.py`, API endpoints in `./backend/app/api/`, CRUD (Create, Read, Update, Delete) utils in `./backend/app/crud.py`.

Enabling Open User Registration

By default the backend has user registration disabled, but there's already a route to register users. If you want to allow users to register themselves, you can set the environment variable `USERS_OPEN_REGISTRATION` to `True` in the `.env` file.

After modifying the environment variables, restart the Docker containers to apply the changes. You can do this by running:

```
$ docker compose up -d
```

VS Code

There are already configurations in place to run the backend through the VS Code debugger, so that you can use breakpoints, pause and explore variables, etc.

The setup is also already configured so you can run the tests through the VS Code Python tests tab.

Docker Compose Override

During development, you can change Docker Compose settings that will only affect the local development environment in the file `docker-compose.override.yml`.

The changes to that file only affect the local development environment, not the production environment. So, you can add "temporary" changes that help the development workflow.

For example, the directory with the backend code is mounted as a Docker "host volume", mapping the code you change live to the directory inside the container. That allows you to test your changes right away, without having to build the Docker image again. It should only be done during development, for production, you should build the Docker image with a recent version of the backend code. But during development, it allows you to iterate very fast.

There is also a command override that runs `/start-reload.sh` (included in the base image) instead of the default `/start.sh` (also included in the base image). It starts a single server process (instead of multiple, as would be for production) and reloads the process whenever the code changes. Have in mind that if you have a syntax error and save the Python file, it will break and exit, and the container will stop. After that, you can restart the container by fixing the error and running again:

```
$ docker compose up -d
```

There is also a commented out `command` override, you can uncomment it and comment the default one. It makes the backend container run a process that does "nothing", but keeps the container alive. That allows you to get inside your running container and execute commands inside, for example a Python interpreter to test installed dependencies, or start the development server that reloads when it detects changes.

To get inside the container with a `bash` session you can start the stack with:

```
$ docker compose up -d
```

and then `exec` inside the running container:

```
$ docker compose exec backend bash
```

You should see an output like:

```
root@7f2607af31c3:/app#
```

that means that you are in a `bash` session inside your container, as a `root` user, under the `/app` directory, this directory has another directory called "app" inside, that's where your code lives inside the container: `/app/app`.

There you can use the script `/start-reload.sh` to run the debug live reloading server. You can run that script from inside the container with:

```
$ bash /start-reload.sh
```

…it will look like:

```
root@7f2607af31c3:/app# bash /start-reload.sh
```

and then hit enter. That runs the live reloading server that auto reloads when it detects code changes.

Nevertheless, if it doesn't detect a change but a syntax error, it will just stop with an error. But as the container is still alive and you are in a Bash session, you can quickly restart it after fixing the error, running the same command ("up arrow" and "Enter").

…this previous detail is what makes it useful to have the container alive doing nothing and then, in a Bash session, make it run the live reload server.

Backend tests

To test the backend run:

```
$ bash ./scripts/test.sh
```

The tests run with Pytest, modify and add tests to `./backend/app/tests/`.

If you use GitHub Actions the tests will run automatically.

Test running stack If your stack is already up and you just want to run the tests, you can use:

```
docker compose exec backend bash /app/tests-start.sh
```

That `/app/tests-start.sh` script just calls `pytest` after making sure that the rest of the stack is running. If you need to pass extra arguments to `pytest`, you can pass them to that command and they will be forwarded.

For example, to stop on first error:

```
docker compose exec backend bash /app/tests-start.sh -x
```

Test Coverage When the tests are run, a file `htmlcov/index.html` is generated, you can open it in your browser to see the coverage of the tests.

Migrations

As during local development your app directory is mounted as a volume inside the container, you can also run the migrations with `alembic` commands inside the container and the migration code will be in your app directory (instead of being only inside the container). So you can add it to your git repository.

Make sure you create a "revision" of your models and that you "upgrade" your database with that revision every time you change them. As this is what will update the tables in your database. Otherwise, your application will have errors.

- Start an interactive session in the backend container:

```
$ docker compose exec backend bash
```

- Alembic is already configured to import your SQLModel models from `./backend/app/models.py`.
- After changing a model (for example, adding a column), inside the container, create a revision, e.g.:

```
$ alembic revision --autogenerate -m "Add column last_name to User model"
```

- Commit to the git repository the files generated in the alembic directory.
- After creating the revision, run the migration in the database (this is what will actually change the database):

```
$ alembic upgrade head
```

If you don't want to use migrations at all, uncomment the lines in the file at `./backend/app/core/db.py` that end in:

```
SQLModel.metadata.create_all(engine)
```

and comment the line in the file `prestart.sh` that contains:

```
$ alembic upgrade head
```

If you don't want to start with the default models and want to remove them / modify them, from the beginning, without having any previous revision, you can remove the revision files (`.py` Python files) under `./backend/app/alembic/versions/`. And then create a first migration as described above.