Technical Manual

DocGenie

Shane Power - 21308533
Razvan Gorea - 21306373
Supervisor - Tomas Ward
Last Updated - 07/4/25

Abstract

A common problem encountered in enterprise data storage is the difficulty in unified search and maintenance. As an organization expands, the siloing of data across multiple data stores (e.g., SQL, REST APIs) makes comprehensive search impossible, as users may be unaware of certain stores or their contents, resulting in missed information. We propose a more modular Retrieval Augmented Generation (RAG) approach that separates the functions of serving user context and pulling context from data stores.
By introducing a central vector store as a query point, this architecture enables more granular access control, enhancing security, and increases uptime. This design also leads to a more maintainable system by decoupling the RAG agent from individual data store connectivity and logic.

Motivation

A common problem encountered in enterprise data storage is the difficulty in unified search and maintenance. As an organisation expands, multiple different ‘data stores’ will be created to hold different data. These might be in the form of SQL, a Rest API etc. This siloing of data ends up making search impossible, as a user might not know certain stores exist, what data they store, and any search will be guaranteed to miss information.

Further to this, when one develops a single RAG (Retrieval Augmented Generation) agent to relay data from these stores to the user, you get more problems. If there is downtime on any of the stores, which can be frequent if there is a large number of stores, will result in no context being served related to that at all. Security is also an issue here. You leave yourself prone to leaking sensitive data into users when you use a single agent for all areas. The issue of how you retrieve context still remains, as this agent will need specialised ways of pulling context.

To address these, we propose making a more modular RAG to split the functionality of serving the user context from pulling context from data stores. By adding a central vector store to use as a point of query, you are able to both control access at a more granular level, thus protecting security, and increase uptime for the user. This also results in a more maintainable system.

Research

We shall now discuss some of the underlying concepts that make up this solution.

RAG

RAG (Retrieval Augmented Generation) is important within any LLM (Large Language Model) based application because it grounds the LLM responses in relevant data (in our case the datastore data). The typical process for implementing RAG is indexing the documents, retrieving relevant data based on some user query, e.g. through semantic search of the index, and finally feeding this context back to the LLM to generate a response. Within our solution, this RAG process is largely what we are conforming to. The key difference is at the index process. This is instead independent of the user prompting, and is **event-driven** rather than periodic. The rationale behind event-driven indexing is explained further in the design.

Access Control Models

Access control is important for ensuring that the retrieval agent does not use context that the user is not allowed to access. Generally, access control models are split between role-based and attribute-based. We favour an attribute-based approach. Conceptually, users should have multiple ‘permissions’ associated with, listing what they have permission to access. This is better than trying to apply the role-based model as we can be flexible in what a permission actually means in this case. We can say that a permission can relate to a namespace, or to a datastore, and both can be correct. As such an attribute-based approach seemed logical.

Vector Databases

A vector database is needed as an efficient way to query the large amounts of data being collated from the different datastores. Embeddings are generated from the data, and then upserted to an index. This allows us to perform semantic search, which is well suited towards agents due to its support for English language queries. Outside of query speed, vector databases have the advantage of supporting high dimension vectors, which lets you store semantic meaning well. Choosing different vector databases is largely irrelevant, in this case we use Pinecone.

Multi-Agent Interaction

The actual viability and benefits of writing RAG to be multi-agent are well-documented. By multi-agent, we generally mean that we have taken a single agents responsibilities, and broken them into small and simpler flows, with possible interaction, both direct and indirect.

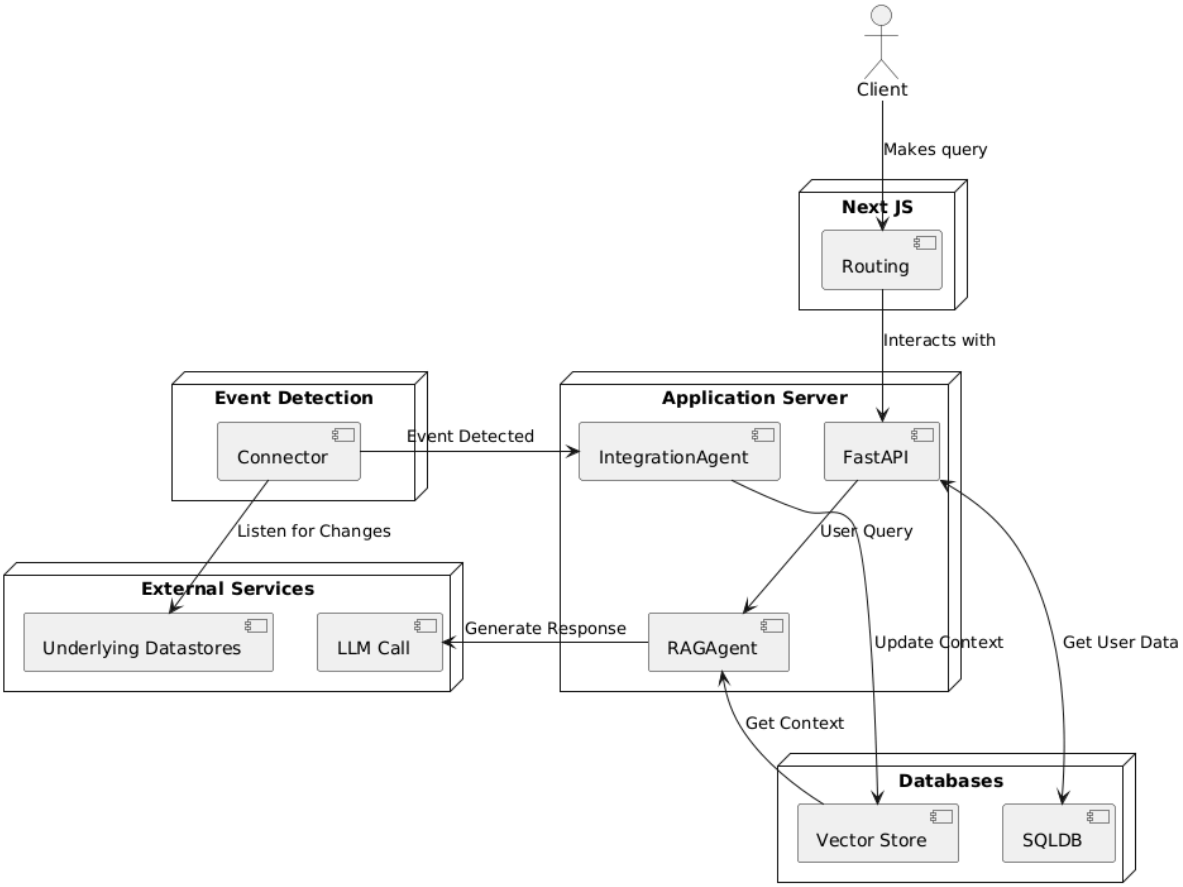
This is still a developing subject in academia, but generally we can say that this almost ‘microservices’ approach to agents is the best way forward. The separation of concerns is the most important outcome. This allows much better hallucination control as each agent is only responsible for one action. Testing is also made much easier, as you can validate outputs much better.

Data Replication

Replication models are typically split into three groups: Full, Incremental and Even-Driven. All three groups have their positives, but event-driven replication is most suited towards an agentic approach. The incremental strategy, where you only make changes since last replication is not flexible enough, as you are still reliant on polling.

Design

System Architecture

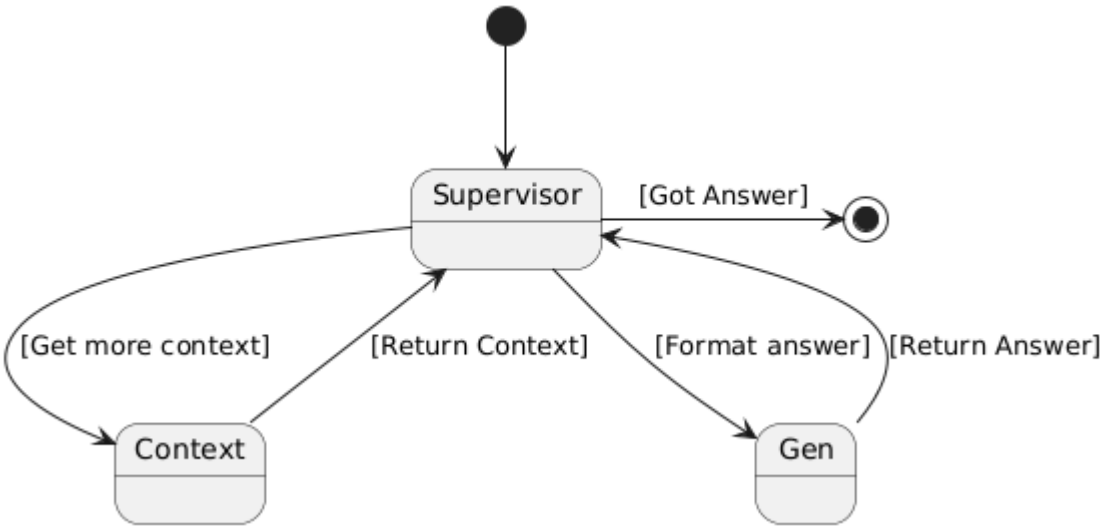


The above diagram shows our system architecture. There is some basic data flow shown here, but do not take it as a full representation of a user query. From this diagram, we can see that the Application Server is of high importance. All main operations are routed through it. Within the server, there are some distinct components.

The most important thing to take away from this image is that the RAG Agent and the Integration Agent are completely distinct from each other. There is no shared functionality with them.

One important thing to note is that the groupings in the diagram do not indicate coupling. For example, Underlying Datastores holding the context does not have any relation to the LLm to generate responses.

RAG Agent

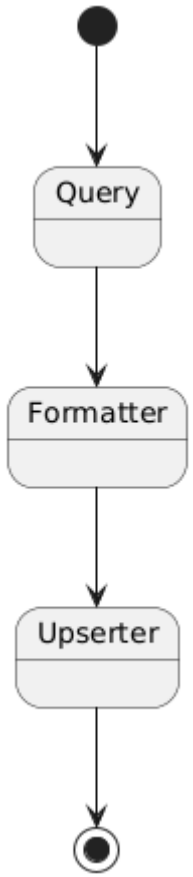


When we think about a LangGraph graph, we are really just thinking about a state machine. As such, we are representing the agents as State Diagrams. In this case, we are following a specific pattern of a Supervisor Pattern. This is where you allow a central node to route itself through the state machine, allowing it to choose what nodes are called. This calling is repeated until some condition is met, either external through some counter, or through the routing node deciding itself to stop.

In our case, the supervisor can decide to either get more context from the Context Node, or get a formatted answer from the Gen Node. This loop is stopped when we hit a limit of calling each node 3 times, to ensure the agent does not take too long. The agent can call less, and decide itself to finish, but it must have called each node at least once to ensure it does not hallucinate an answer.

The reason behind us using the supervisor pattern is to allow us some flexibility in pulling in context. You can't predict the amount of context needed so allowing the agent to dynamically decide to stop calling for more context simplifies the logic exposed to the developer maintaining any of this code.

Integration Agent



Similar to above, we are showing the Integration Agent as a State Diagram. In this case we do not need a supervisor pattern, as there is not a need for dynamically routing between the nodes. In this case, an Event Monitor invokes the graph, wirth the name of the datastore to look at. The graph uses the Query node to query that datastore to get as much data as possible. The formatter nodes formats the data to standardise it into something that can be upserted to the Vector Store, where it can be used in RAG. The upserter then upserts that formatted data and completes the graph.

Importantly here, actions such as “gathering data” are not necessarily linear. In reality, the Query node can do multiple “gather data” actions within one run. This is referred to as tool-calling, and is asynchronous.

Event Detection

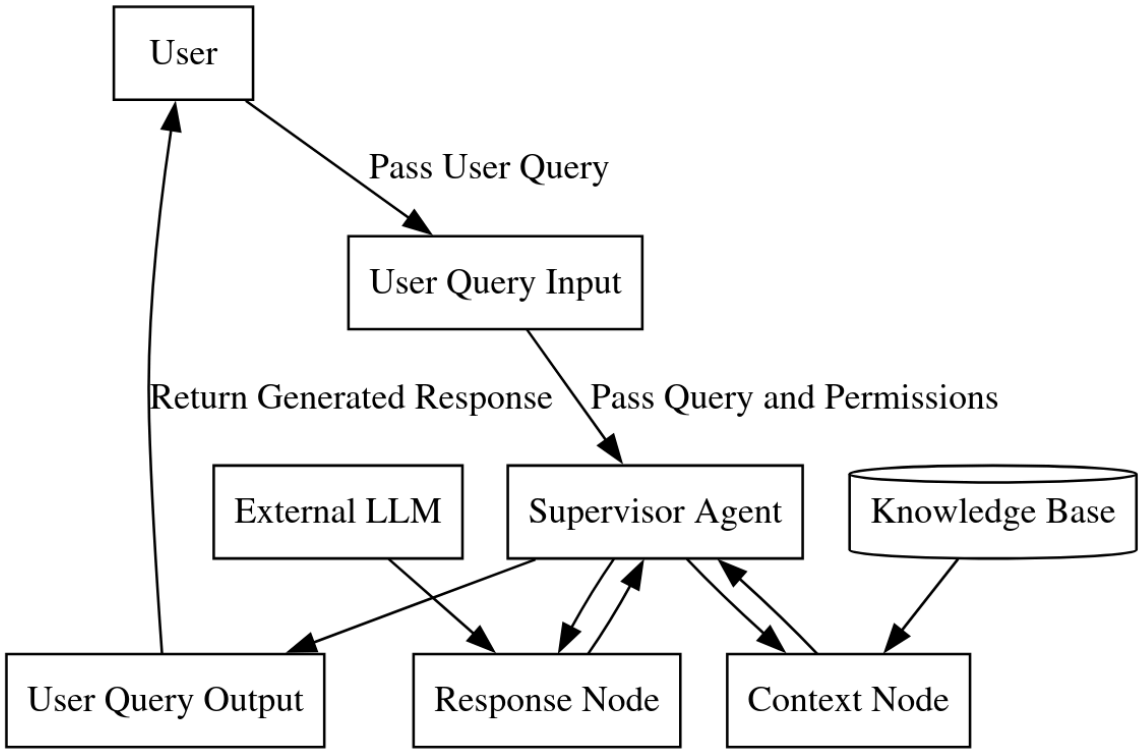
Event Detection is the module where Connectors listen to external datastores, and relay updates to the Integration Agent. Further to this, Connectors must provide an interface for data querying for the Integration Agent to use. The class diagram above shows this. Connectors are organised under a class hierarchy. Connectors inherit basics from the abstract class, such as a startup and shutdown methods for connectors. This is so we can guarantee a few basic methods for use by the ConnectorBuilder class, which manages connections.

The ConnectorBuilder class is responsible for holding a dictionary of these different Connector objects. It is responsible for registering each connector at startup, and is also used by the RAG Agent when it needs to pull context, as it can only pull context that the User has permissions to.

To create new classes of ConnectorBase, it is easy to do so, which is done to ensure scalability.
How connectors invoke the Integration Agent is naturally variable across each subclass.

Data flow

The main data flow to be concerned with is the flow of a user making a query to the RAG Agent. We can show this through a Data Flow Diagram.



We can step through this. User queries are transformed to include permissions associated with the user. This is passed to the Supervisor Agent (Which is the first node of the RAG Agent). At a basic level, this user query is supplemented with context in a series of state transitions across the RAG Agent, represented by the Response and Context Node, until a response is generated. This response can then be returned back to the user.

Implementation

Practical stuff on how the system is built and deployed. code snippets included here
Basically more in depth logic on parts of the design, and justifying why its implementation is related.

Agents

Before going into more detail on how these agents actually work at a lower level, we need to explain some components of LangGraph. We want to treat each langgraph node like its own individual LLM, that shares a global state with every other node. When we say that we invoke a node, we mean that we have triggered the LLM in that node to perform whatever it is prompted to do, influenced by that global state, which each node adds to when it is finished. With this, we can talk more about the code of the agents.

Each LLM inside each LangGraph node is an agent native to LangGraph, the ReAct agent. This agent architecture repeatedly calls predefined functions available to it, called ‘tools’ to try and achieve its task. To get a bit lower-level with this, we will look closer at the Context node of the RAG Agent.

```
result: Any = self.context_agent.invoke(state)
```

We can see from this snippet that we can invoke that ReAct agent with the global state. Further to this, tools can be registered to this specific agent ahead of time.

```
@tool
def query_dbutils(user_query: str, namespace: str) -> Dict[str, Any]:
    """
    Queries the vector database with semantic search.
    """

    query_embedding: Any = dbutils.create_query_embedding(user_query)
    # the agent knows what to do with the tool through the tool decorator and the multi-string.
    results: Any = dbutils.search(
        index_name="quickstart",
        namespace=namespace,
        embedding=query_embedding,
        result_amount=5
    )
    return(results)
```

A tool is just a function to help that agent out. This tool can be called by the agent, which passes its own parameters in which it thinks fits, and gets an answer out. The docstring is interpreted by the agent to give it an idea of what to expect. In this case, this tool helps the agent do semantic search on the vector database.The docstring essentially acts as a guide for the agent to better understand the tool. Tools are bound to a node at init, like so:

```
self.upserter_agent = create_react_agent(
    model=self.llm,
    tools=[create_upserter_tool(dbutils=self.dbutils)],
    prompt=self.upserter_prompt
)
```

All of our nodes use the ReAct agent architecture. While a custom architecture can be developed, in reality there is little need for our use case.

We can force each of these nodes to output to a certain model. In our case we want our formatter node, that formats data to be upserted to Pinecone, to output data that is valid for our schema. In this case we can define a Pydantic model that it must use in its output.

```
class PineconeSchema(BaseModel):
    """A single row in the pinecone."""
    id: str = Field(description="An. Make sure it is not already in use.")
    upsert_text: str = Field(description="This should contain all of the data you collected.")

class PineconeSchemaHolder(BaseModel):
    """Holder for each row in the pinecone."""
    pinecone_items: List[PineconeSchema] = Field(description="A list of pinecone rows.")
```

Event Detection

There is some element of manual work when setting up event detection. This manual work just involves setting up the connection. In our case, we shall look at the Kafka example we have bundled into the application.
A separate Docker container is running a Kafka Connect server. With each of our PostgresConnector objects registered in our DB, they will register their topics into that kafka server.
With this, we can now listen to any events reported by the corresponding Postgres DB. Remember that the whole Postgres end of that replication, i.e. the process of outputting events for our kafka connector to pick up on is not part of the project.

Pinecone

Pinecone is the vector database we are using to store all of our replicated data. As discussed above, the Integration Agent will format data retrieved into a certain Pydantic model. We have a wrapper for a Pinecone Client, called DbUtils, to interact with it. The agent has one of the methods to upsert bound as a tool to it.

This wrapper to the Pinecone DB is also used in our RAG portion. We have methods to both generate query embeddings, to be used is semantic search, and then a method to do the search itself. Once again, the methods are used in bound tool calls. Below is an example of

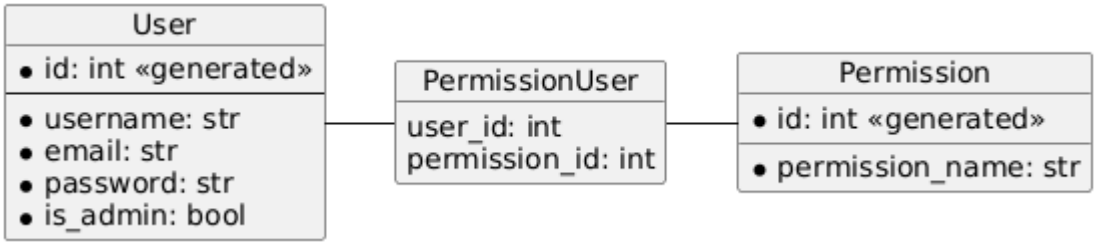
one of our methods:

```python
def upsert_embeddings(self, index_name:str, namespace:str, embeddings:list[dict]) -> int:

    total_upserted = 0
    index: Index = self.get_index(index_name);

    for ids_vectors_chunk in self.upsert_chunker(iterable=embeddings, batch_size=self.upsert_batch_size):
        response: UpsertResponse = index.upsert(
            vectors=ids_vectors_chunk,
            namespace=namespace
        )

        if self.env.DEBUG_MODE:
            print("Embedding upserted")

        total_upserted += response["upserted_count"]

    return total_upserted
```

Permissions

Permissions are our way of saying what context a user is allowed to access. There are two parts to this.

Firstly, the user has permissions bound to him in SQL with a many-to-many relationship. Each row in the Permission table is a one-to-one relation to our notion of Connector objects. In other words you are granted access on a connector-by-connector basis.
The SQL tables in question are like this:



When a user makes a query then to the RAG Agent, we can send their permissions in the query.

```python
answer = supervisor_agent.take_input(chat.body, user.permissions)
```

With these user permissions, the agent is only able to query namespaces on the Pinecone DB containing those permission names. Similarly to this, the Ingestion Agent knows to upsert records to certain namespaces depending on the connector name that created the event.

API

We have a central FastAPI working around all of these different systems. It is primarily there for the purpose of coordination and centralisation between systems like the agents and the PineCone db. We shall cover some of the details of this API. The most obvious usage is as a point of interaction for the user. For example all invocations of the RAG system are routed

through the API. We also use the async context manager to handle startup and shutdown events for things like connectors, letting us gracefully exit on shutdown.
FastAPI is ASGI (Asynchronous Server Gateway Interface), which lets us handle all of our functions asynchronously.
Dependency injection is very important in FAST. We can define all of our wrappers at startup, for example the DBUtils, and then inject that single object as a single dependency across the application. For example, a route can call an annotated method to yield that object whenever they want.

```
ConnectorDep = Annotated[ConnectorBuilder, Depends(get_connector_manager)]
```

SQL

Like the DbUtils class discussed above, we have a similar wrapper for our SQL operations. This is useful as we can abstract things like session management out of the api, giving us the ideal controller level of logic. It also gives us a handy object to inject as a dependency into routes that need it. This class, SQLClient, has a few noteworthy points.
We provide a few easy methods, with the intention of just letting the upper API level **deal only with Pydantic objects**. For example, a method to add objects to the db can just take the object as the table you want to add to.

```
def add_object(self, to_save: T) -> T:
    """Add an object to the db. Takes generics."""

    if hasattr(to_save, "password") and to_save.password:
        to_save.password = self.hash_password(to_save.password)

    with self.get_session() as session:
        session.add(to_save)
        session.commit()
        session.refresh(to_save)
        session.close()
    return to_save
```

Only the object needs to be passed, so the upper level does not need to have any real knowledge of the table, as we can infer the table from the type of the object. This pattern is continued in other methods. One aspect of this model-based approach is it lets us define relationships at the model without knowledge of the underlying tables. Using the previous example of Users and Permissions, we can get these eager relationships within the one

session.

```
def get_by_id(self, table: T, primary_key: int, eager_relationships: list = None) -> Any:
    """Fetch a record on its primary key with optional eager loading of relationships.
       Uses an internal session.
    """
    with self.get_session() as session:
        statement = select(table).where(getattr(table, "id") == primary_key)

        if eager_relationships:
            for relationship in eager_relationships:
                statement = statement.options(selectinload(relationship))

        obj = session.exec(statement).first()
        session.close()
        return obj
```

Selectinload lets us load that eager relationship, so the API doesn't need to care if it's a many-to-many or not, it can just load the relationship by saying the attribute name on the model itself.

For example, our User model can just have its Permission objects related directly on it as an attribute,

```
class User(SQLModel, table=True):
    id: Optional[int] = Field(default=None, primary_key=True)
    username: str = Field(index=True)
    email: str = Field(index=True)
    password: str
    is_admin: Optional[bool] = Field(default=False)
    permissions: List["Permission"] = Relationship(back_populates="users", link_model=UserPermission)
```

Problems solved

This section will briefly list the main problems addressed by this system.
We will also discuss some of the unforeseen problems that came up during the project that we needed to solve.

Main Problems

The first main problem to be addressed is the need to partition data. We can say we accomplished this through the Permissions model and namespaces on Pinecone.

The second main problem is the requirement for the data in the datastores to be available to be queried by the user.
We have achieved this by our system of connector classes, and their ability to invoke the Integration Agent to do further querying.

Other problems

There are a few smaller scale problems that presented themselves during the project.

One such problem is the youth of Langgraph. Langgraph is still a very young project, so there are features that were being changed during our project. For example, the idea of structured output, where a node can conform itself to a certain template, is broken in some versions. There are more problems with the framework like this that made it unpredictable, in particular during the discovery phase of the project.

There are also more theoretical ideas that are still in their infancy during our project. A new idea being covered in papers around January 2025 was around how agentic systems can develop a common format for describing datastores for use in RAG. We were too far into our current system to use this new model, which would have made our system better.

In general, connectors were a big problem and timesink. There is a lot of tedious work involved in setting up a Kafka Connector. In addition, for the purpose of demonstration, the setting up of containers for a postres and rest api was tedious too.

The non-deterministic nature of agents was a major hurdle for testing. Writing tests for something like a simple method, such as an api route is trivial, but writing tests for a non-deterministics system that can essentially spew out anything it wants can be difficult. We solved this by applying the standard pyramid approach at a more granular scale. Write unit tests guaranteeing the return signature of each node, and then build up to checking the output of the total invocation with different permissions.

Future work

There are a few areas that this project can be improved upon.

Firstly, the system of event detection (outside of the connector class system) could be improved upon. The kafka should be more generalised to support more than just postgres, and should be brought more into the main application, instead of living out in a separate container.

While the ReAct agent is suitable for our purposes, a custom agent could be written to get some better tool-calling behaviour. For example the context node of the RAG could be adjusted to get more context.

Testing always has room for improvement. While our code coverage is high (85% - 90%) the actual quality of the tests in this coverage can be inconsistent. Tests could be made more regular.

While the Javascript used for the frontend of the project is acceptable, it could use further componentization. It was not a major consideration for us, as we only really needed one page, but if you expanded the JS it would be needed.

Testing

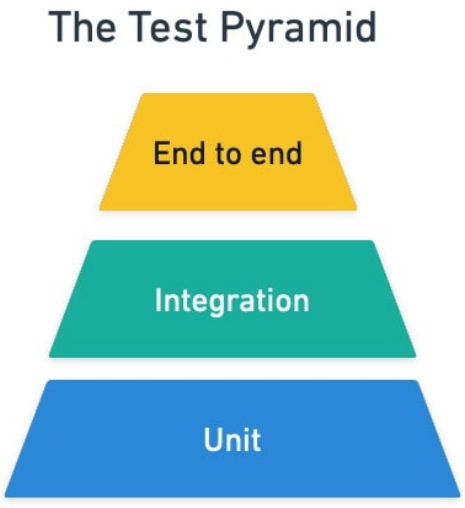
Test Strategy

Our approach is risk-based, where we prioritise testing based on the potential impact and likelihood of defects. A key part of this is to achieve high test coverage. We prioritise high test coverage for a few reasons. Firstly, it is necessary to get a good base of unit tests to conform to the agile methodology. It also allows for growth and ‘filling in the gaps’ later, i.e. writing more complex unit tests. Its also time-efficient, as you are not testing the same area twice.

Following this strategy, our tests are wide-ranging. We test all major modules, such as DbUtils, the SQLClient, the API, the frontend, the Agents, and the Connector classes.

A combination of Unit, Integration, End-To-End and Manual Testing was used.
The vast majority of testing is Automated. The reasoning is simple: time constraints. Manual testing is limited only to the Frontend, and this is to be completed by a group of volunteers. Otherwise, all testing is automated.

Agile testing principles are held in high regard, with us both mirroring the typical Agile test pyramid and the agile practices of continuous testing and coverage metrics. Our pyramid of testing is done to ensure we have sufficient coverage without expending too much energy on fixing E2E tests that break constantly.



PyTest and jest are the two tools used in our testing. Both of these tools provide coverage reports that we can use later in our pipelines, and are generally the standard for Python and JS.

As per Agile methodology, tests are being written to cover features as they are implemented, not after.

Execution

Here we describe how tests are carried out, tools and how we handled results.

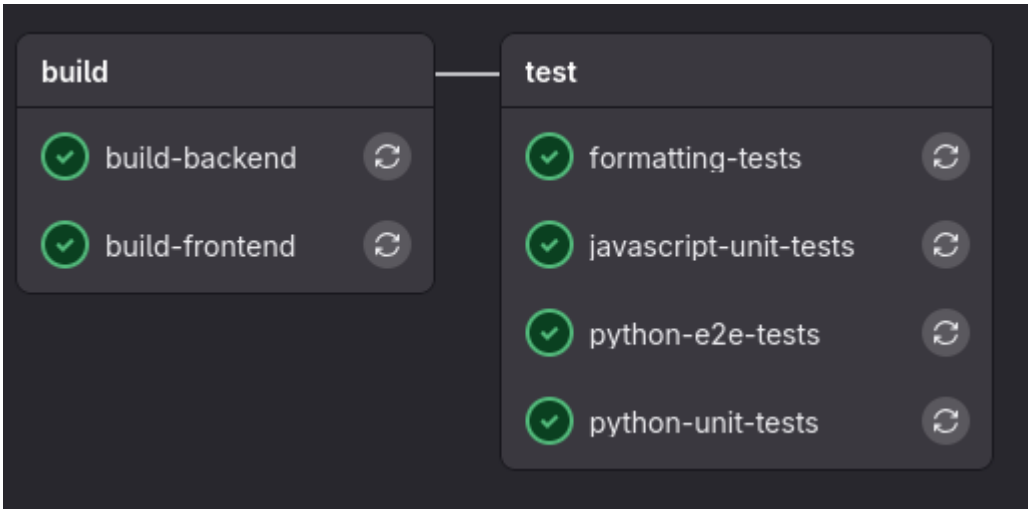
All automated tests are being run on the Gitlab runner environment as part of the CI-CD pipeline. Essentially, this is just a docker container that we install the dependencies to to run the tests.
While unit tests have no real need for external secrets and services, such as api keys, out higher level integration and end-to-end tests do need them, so we have used the Gitlab pipeline variables to populate these. This is similar to a keyvault, and these variables are not visible in the pipeline runs themselves.

This pipeline is run on each commit, and there is a requirement to approve the coverage reports generated by a merge request before it can be merged into main.

The following metrics are generated by these automated tests:

- Total tests executed
- Total tests passed
- Total tests failed
- Test coverage
- Style issues

The following image shows our pipeline stages, and the tests ran in them.



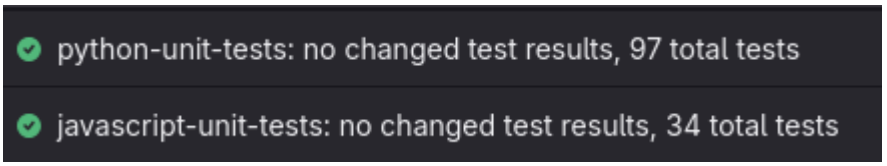
Results

Now we can discuss some of the actual testing results produced from the project.
We shall start with the coverage reports. The following is the PyTest coverage report final metrics:

```
TOTAL                                          1947    165    92%
```

Telling us that out of the 1947 statements in our system, we do not cover only 165. Many of these are non-critical statements in config files, so this is a good result overall.
The coverage that most projects aim for is around 75%, so in comparison we are doing well in this regard.

In terms of total tests executed:



This totals to 131 total tests.
We do not show the cases of these code as there are far too many to go through in this document, but we shall go through some of these tests now to give you an idea of their structure.

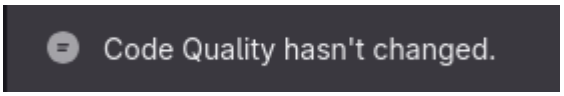
We shall look at some of the FastAPI tests being done. These are located at **/src/tests/unit/api** in the repo.
Below are two tests we shall use as an example.

```python
# get_last_conversation
def test_get_last_conversation_not_found(client: TestClient) -> None:
    response: Response = client.post(url="/conversation/latest?user_id=1")
    assert response.status_code == 200
    assert response.json() is None

def test_get_last_conversation_success(client: TestClient, sqlclient: SQLClient) -> None:
    convo1: Conversation = create_test_conversation(user_id=1, id=1)
    convo2: Conversation = create_test_conversation(user_id=1, id=2)
    convo3: Conversation = create_test_conversation(user_id=1, id=3)
    sqlclient.add_object(to_save=convo1)
    sqlclient.add_object(to_save=convo2)
    sqlclient.add_object(to_save=convo3)

    response: Response = client.post(url="/conversation/latest?user_id=1")
    content = response.json()

    assert response.status_code == 200
    assert content["id"] == 3
```

Firstly, we should note that these tests are written in pytest. As such, we have defined a conftest file to define pytest fixtures, which we can use to simulate some external dependencies. In this case we have a TestClient object, which is a working version of our FastAPI. We also have a mock of a SQLClient object.

Both of these tests are to test two different branches in logic, so we get full coverage on the same endpoint. This pattern of having external fixtures recycled into each test, where multiple tests are written for the same function is reused for every endpoint in the system.

Style issues are also tracked in our tests. On each merge request we can display it if they are a problem:



Our usability testing also resulted in some interesting insights. Out of the 5 users we tested with, We found all users to be able to navigate the chatbot interface fine. Users did find

some finer usability issues, such as the lack of an indicator that the users query is being processed. As such, we added a new loading indicator after this feedback to solve this.

Our final set of results exist in reference to the main use cases of the functional specification. In this case, we are checking the overall functionality of the RAG system. We want to establish that a user can get back data that has correct permissions.

This test simply invokes the agent with a user query, and then we wait to see what it answers with. We can then assert that certain data is present in the answer. This is an undoubtedly flaky way to test, but we can still get an overall high pass rate (80% in total) The non-determinism is a real hurdle here in implementing more.