BioGraph Capstone Project Report

Jos van Veen, Oscar Duys, Kate Birch

September 2024

Abstract

BioGraph is an easy-to-use, full-stack software workbench for the management, visualisation, manipulation and analysis of SBML models. Researchers and biologists alike can use this tool to automate the fetching of SBML models from the public BioModels database, and the conversion of SBML models into graph data. Moreover, BioGraph connects SBML models with their linked databases and offers multiple schema to choose for models, allowing even more control over the information presented. BioGraph is supported by a Neo4j database, which stores SBML models and their linked databases, and incorporates an interactive GUI, for ease of use. Combined, this provides the user with state-of-the-art graph processing techniques, accessible via the GUI, as well as the flexibility to execute their own Cypher queries. BioGraph is also made with software design principles in mind, with a focus on: fast operations, balance between flexibilty and repeatability, a clean and visually attractive GUI, and having essential, well-made functions instead of many unecessary features.

1 Introduction

The goal of BioGraph is to produce a viable, and complete tool for biologists and researchers to manage, visualise, manipulate and analysis SBML models.

SBML (Systems Biology Markup Language) is the de facto standard for storing and representing computational biological models. SBML models specify certain characteristics that must be associated with biological models, (a more detailed look at this is conducted such as, for example, a unique identifier, relationships such as biochemical reactions between biological entities, units of measurement in such reactions, species, and tags to point to references in other databases. These models are vital to academics and researchers as they allow for reproducibility in research, and facilitate universal communication and documentation of biological models. SBML models are available for free, curated and stored by the BioModels website.

There is a problem that arises, however, from the way SBML models are currently stored. That is, each model is stored and written individually to a separate file. This practice promotes inconsistencies, redundancies and missed opportunities for making connections between models, because of the disconnected and context-less manner in which these models are stored. Naturally, this risks a negative effect on research that relies on computational modelling.

The aim of BioGraph is thus to solve this problem by creating a unified and connected approach to the storing of these models. Furthermore, BioGraph aims to go beyond simply fixing the problem presented, but to improve upon the entire process of working with SBML models and pipelining it into one, user-friendly software workbench.

This project took an agile design approach. Since the team had limited but regular meetings with the client, a focus was placed on establishing the most essential functional requirements first. Then, after each meeting with the client, the team would work on the next incremental step in the project, as well as fixing any features that the identified as problematic. In this way, the project was developed in small, manageable steps that mitigated the impact changing requirements, while still allowing for some version of a working mockup to be presentable (if not incomplete) to the client. In the initial meeting, the team captured user stories, and decided on use cases. The beginning of the project dealt with the basic and essential pipeline of downloading, converting and storing SBML models in a Neo4j database. Then, frontend and backend was created. The frontend had a working GUI, with a rough design of what the final GUI would look like, which was helpful for client meetings. The backend had communication set up with the Neo4j database. All other features were added after this. This agile, iterative design approach gave the team flexibility and a clear idea of priority.

2 Requirements Captured

A variety of design artefacts were produced in order to capture and analyse the requirements of the project. Such artefacts include the functional, non-functional and useability requirements, stakeholders and user stories - illustrated by use case diagrams - and test cases.

2.1 Functional Requirements

Briefly, BioGraph must be a full-stack, user-friendly software workbench that allows users to:

- Download, upload and convert SBML models into a Neo4j database, automatically;
- Visualise the graph data, interact with it through a GUI, and have some control over how it is displayed (e.g. choosing a schema);
- Manually merge and delete nodes;
- Query the Neo4j database, with both pre-made queries and a user's own query;
- Apply complex graph processing techniques.

A special emphasis is to be placed on automation and repeatability in this process. Specific use cases and features are discussed in the architecture analysis.

2.2 Non-Functional Requirements

BioGraph also needs to meet non-functional requirements that ensure that it is secure, reliable and usable. Such requirements include being:

- Well structured, robust and modular. The code should should be designed with OOP principles in mind. It should be well documented, and well encapsulated so that the maintenance and adaptation of the code is made easier and more efficient later on in the future;
- Efficient production. BioGraph should make use of pre-existing code and packages so that the team can have a faster production schedule, and save time by not building a project from scratch. Importantly, the team will need to ensure that these dependencies are carefully managed so that users won't have to have an overly complicated installation guide;
- Portable. BioGraph should be able to run on multiple different architectures provided the necessary dependencies are available;
- Performance. BioGraph should provide a relatively fast response time for graph operations, and for large operations (like merges on many nodes, or loading of many graphs), it should have a large throughput;
- Reliability. Since BioGraph will need to handle message-passing between mutliple different entities, it will need to ensure that this communication is reliable and handles errors well. Error messages and handling should be present at all steps in the design.

2.3 Usability Requirements

Lastly, BioGraph must consider the users of the system. It must have:

- Intuitive GUI. BioGraph should be easy to use, with a GUI that is easy to learn and not overly complicated.
- Comprehensive GUI. The user should be able to perform the entire phase of working with a model (from importing, to converting, to manipulation to analysis and querying and finally to exporting a model) from the GUI alone. This is inherent to the design of a software workbench, as all operations should be packaged as one product.
- Easy installation. Since BioGraph is a software workbench and will necessarily require a lot of dependencies, these dependencies must be well managed and clearly stated such that the user installation is not complicated and time-intensive;
- Good guide. A well-written and thoroughly documented user manual should be readily available;
- Allow for flexibility. BioGraph needs to cater to users of all experience levels, and so users should have the option for both pre-made, automatic querying, and the ability to write their own queries.

2.4 Use Cases

From the meetings held with the client, various artefacts were produced, identifying the stakeholders of the project and generating various user stories for use cases. ThE stakeholders were identified as:

- The client, M. Keet, who will provide final approval on the functionality and specifications of BioGraph
- Primary users: (i) Biology researchers who use SBML models and would benefit from BioGraph and (ii) educational institutions who would use SBML models as educational resources
- Secondary users: (i) Researchers at large who use SBML models for computational modelling and research and (ii) other software developers who are interested in similar problems of graph processing and scientific models
- Partners: (i) The BioModels database administrators and maintainers, (ii) UniProt, ChEBI and other linked database developers, and (iii) other funding partners

Four core user stories were established and refined, across each agile development cycle, and were finalised as documented below:

2.4.1 Automated SBML Model Conversion and Integration

Description: Biologists need to convert SBML models from the BioModels database into a Neo4j graph data store for unified analysis. The system administrator initiates the process by configuring the automated pipeline. The system fetches SBML models from the BioModels database and converts them into graphs using the neo4jsbml converter. The graphs are then stored in the Neo4j database.

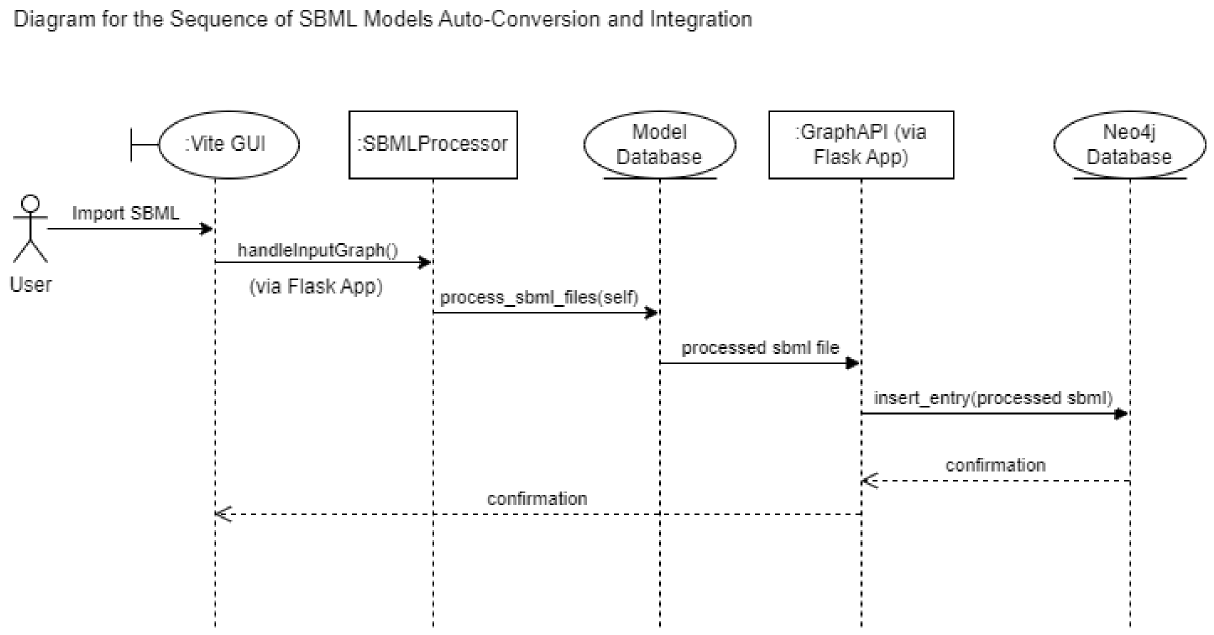

Figure 1: Sequence Diagram for the Upload and Convert Use Case 2.4.1

2.4.2 Graph Merging and Confirmation

Description: Biologists manually identify redundant nodes across multiple SBML models that need merging. They initiate the merging process via the GUI, where the system identifies similar nodes and proposes merges. The system displays the graphs with the suggested merge. Upon confirmation, the system merges the nodes, updating relationships and ensuring data integrity.

2.4.3 Graph Querying and Analysis

Description: The user searches for a model via the GUI, only using a variety of search descriptions, such as organism type, species, compound - all of which are fields specified in a given schema. The application translates these terms into a query and queries Neo4j. Linked databases may also need to be accessed if certain requirements are not explicitly available in the Neo4j database.

2.4.4 Linked Database Integration

Description: The user wishes to see a model with a literal, human-readable tag, instead of an ID pointing to a linked external database. The backend calls a class that will retrieve such identifiers from a database of various IDs and their associated values in the linked databases. The value is returned and displayed in the GUI.
These use cases are illustrated in the diagrams below:

2.5 Test Cases

The last major design artefact in the requirement capturing phase is the test cases. The test cases were used later in production to guide the writing of tests for the code - making the process faster, and informed the team on how to design the system. The tests themselves are detailed in Section 2.5, but the cases involved testing: (i) The system's ability to successfully import, convert, and visualize SBML models in Neo4j. (ii) The functionality to merge graphs within the Neo4j database and manages relationships and the system's ability to merge graphs with conflicting relationships. (iii) Various user interactions with the graph processing tools (e.g.,

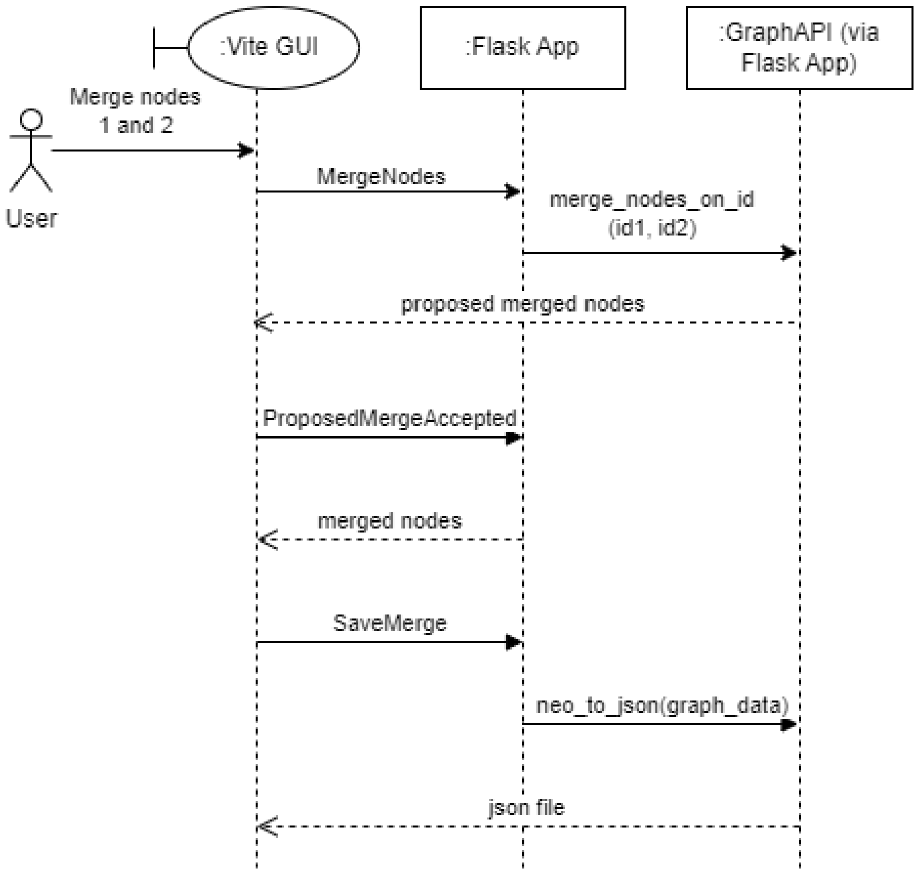

Figure 2: Sequence Diagram for the Merge Use Case 2.4.2

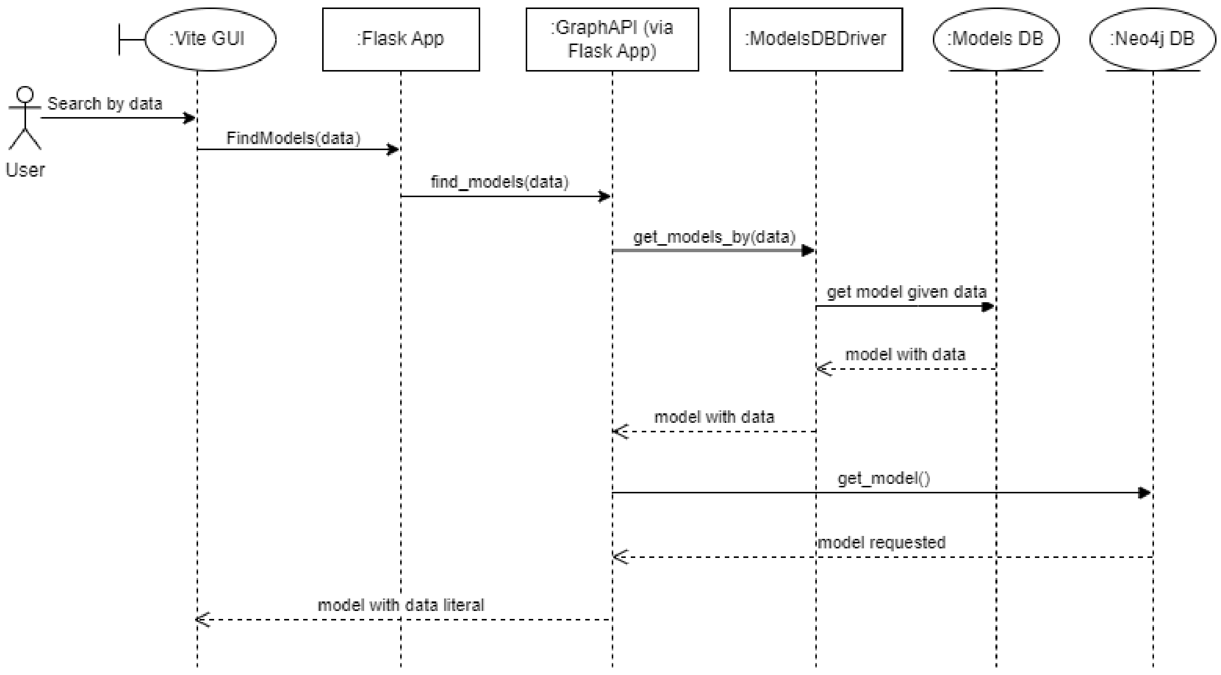

Figure 3: Sequence Diagram for the Query and Link Use Cases 2.4.3 and 2.4.4

finding cliques, calculating similarities). These test cases were vital in the production of tests later on in development.

3 Design Overview

The system design was developed to ensure the expected behaviour of the final project. A strong priority was placed on separateness of the backend, frontend and databases. The backend handles processing logic and connection to databases, while the frontend is the interface the user interacts with. This ensured scalability, robustness, ease of testing and debugging, is in line with OOP development principles, and, because of this, lends itself well to an agile engineering method. BioGraph naturally developed an OO design. This gave the project modular and well-encapsulated classes (evidenced in the class analysis diagrams), which were easy to maintain and change as the requirements of the project evolved. In this way, OOP is well suited to the agile development approach to this project because of the robustness and modularity it affords - allowing the project to respond quickly to change and thus attain the desired behaviour detailed in earlier sections. Furthermore, it is clear from the package dependencies diagram, that BioGraph heavily leverages preexisting packages and libraries. This supports an efficient and scalable development, as time is not wasted on writing code that can easily be adapted to the needs of the project. It also allowed the team to focus on the logic and algorithm design, which were more specific features. Docker is used to manage these package dependencies and allow for a portable and easy-to-install application. The use of a client-server architecture for message passing between layers also promotes encapsulation and a modular design.

3.1 Backend Architecture

The system architecture of the backend layer implements a Flask server, which handles logic and facilitates message passing between it, the frontend and the database layer (i.e. separating the frontend from the database layer.) It is broken up into several key modules:

- At the core of the backend is GraphAPI in routes.py which defines the message passing protocols between the main boundary points of the system (shown in the architecture layer diagram). What it does is register routes for verious necessary operations with the Flask server. Such routes involve importing and fetching schema, fetching graphs from Neo4j, and other graph queries and operations. This is an important layer of abstraction as the logic that supports the Cypher queries which are passed by the API to the Neo4j driver, Neo4jDatabase. The driver is the only class that interacts directly with GraphAPI, and is responsible for handling graph queries and database connections to Neo4j. GraphAPI communicates with the Vite GUI frontend as well as the ModelDatabase class via an interface. By abstracting the database logic and separating it from the logic of the rest of the system, OOP principles are more closely adhered to and there is more flexibility for developers to add and edit graph operations as more features are desired.

- The Model class in models.py provides high-level control over SBML models. Such methods involve uploading SBML models to the Neo4j database, finding models given various, human-readable search terms (such as finding a model by species or organism), and manually manipulation operations (such as merging or deleting nodes). It is responsible for communicating with the Neo4j database with more specific graph processing operations compared to the methods in GraphAPI. It also transitively interacts with the aggregate database interface, ModelsDatabase, to find linked components referenced in SBML models. to ensure that biological models are correctly parsed, stored, and queried. Utility functions in utils.py and FileManager classes aid in file management, JSON conversion, and generating graph files, while ensuring proper handling of file uploads and responses. It

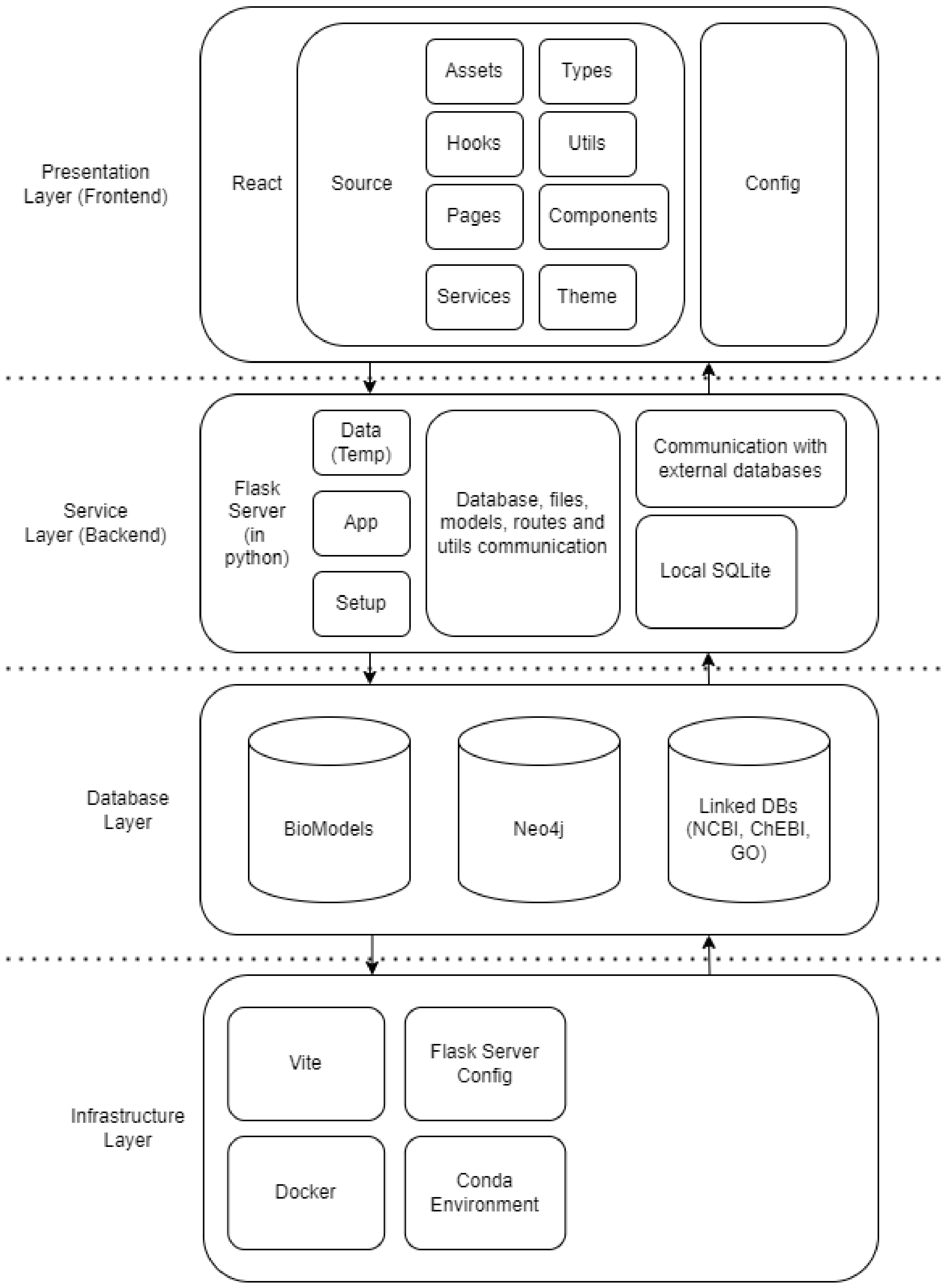

Figure 4: Layered Architecture Diagram

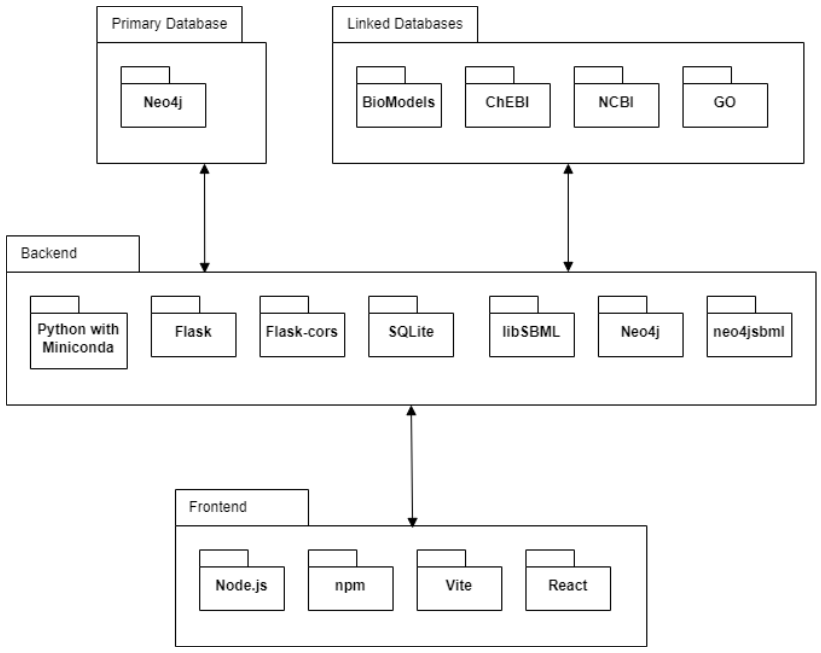

Figure 5: Package Dependencies Diagram

paired with the Utilities class in utils.py and FileManager in files.py provides support for ensuring that SBML models are correctly stored, parsed and queried, as well as proving file management, creating files and conversion to JSON.

- The FlaskApp class in app.py coordinates the entire backend by intialising the Flask server, using the configurations stored in Config. Also, CORS is adhered to because of the use of the flask_cors package, which allows for communication across the external APIs and GUIs.
- The Vite GUI is only related to the GraphAPI via a series of client-server message passing. GraphAPI is the only component in the backend (and no compenent in the database layer) that the GUI communicates with.

3.2 Database Architecture

The architecture of the database layer handles SBML models, taxonomy, gene ontology and chemical models from linked databases like BioModels, GO, ChEBI and NCBI. It is described as follows:

- The ModelDatabase class in model_database.py manages a primary relational database, Models - which implements SQLite, that aggregates the SBML models from BioModels, and connects them with their related taxonomy retrieved from NCBI, their gene ontology from GO and chemical data from ChEBI. There are various get and set methods included, like adding models, taxonomy, and ChEBI data. This allows the user to view a fully populated SBML model, as in their original states they only contain references to entries in linked databases, and so, for example, an entity is represented as a fully annotated diagram instead of being full of unreadable ID numbers, which are impossible to visualise.
- SBMLProcessor in sbml_processor.py manages the bulk processing of SBML files. It calls the SBMLModel class to read individual SBML files. It can also extract taxonomy and chemical compounds so that cross referencing is supported. The BioModelsService class is an interface to the BioModels website, which is responsible for fetching and downloading

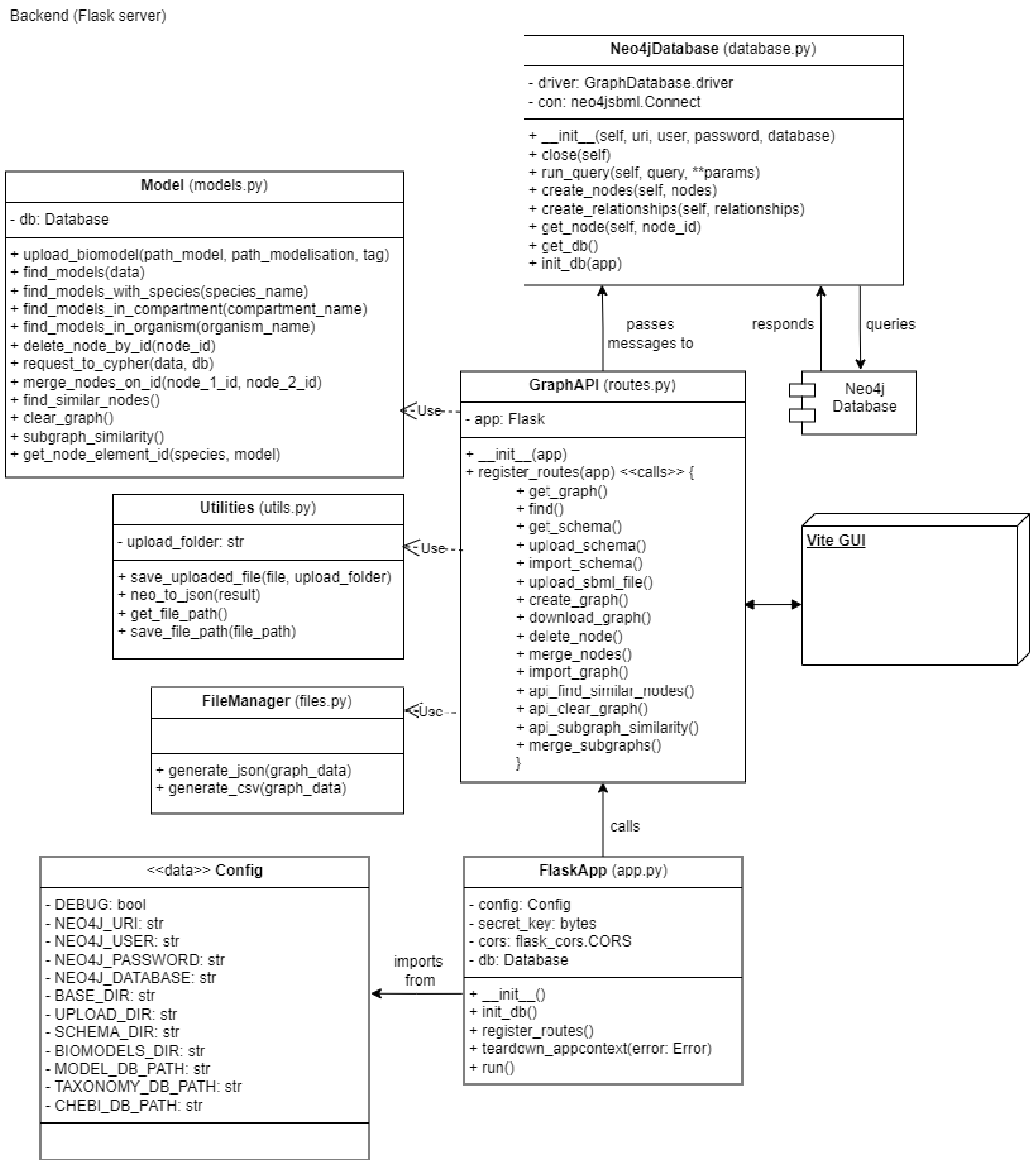

Figure 6: Class Analysis Diagram of the Backend Layer

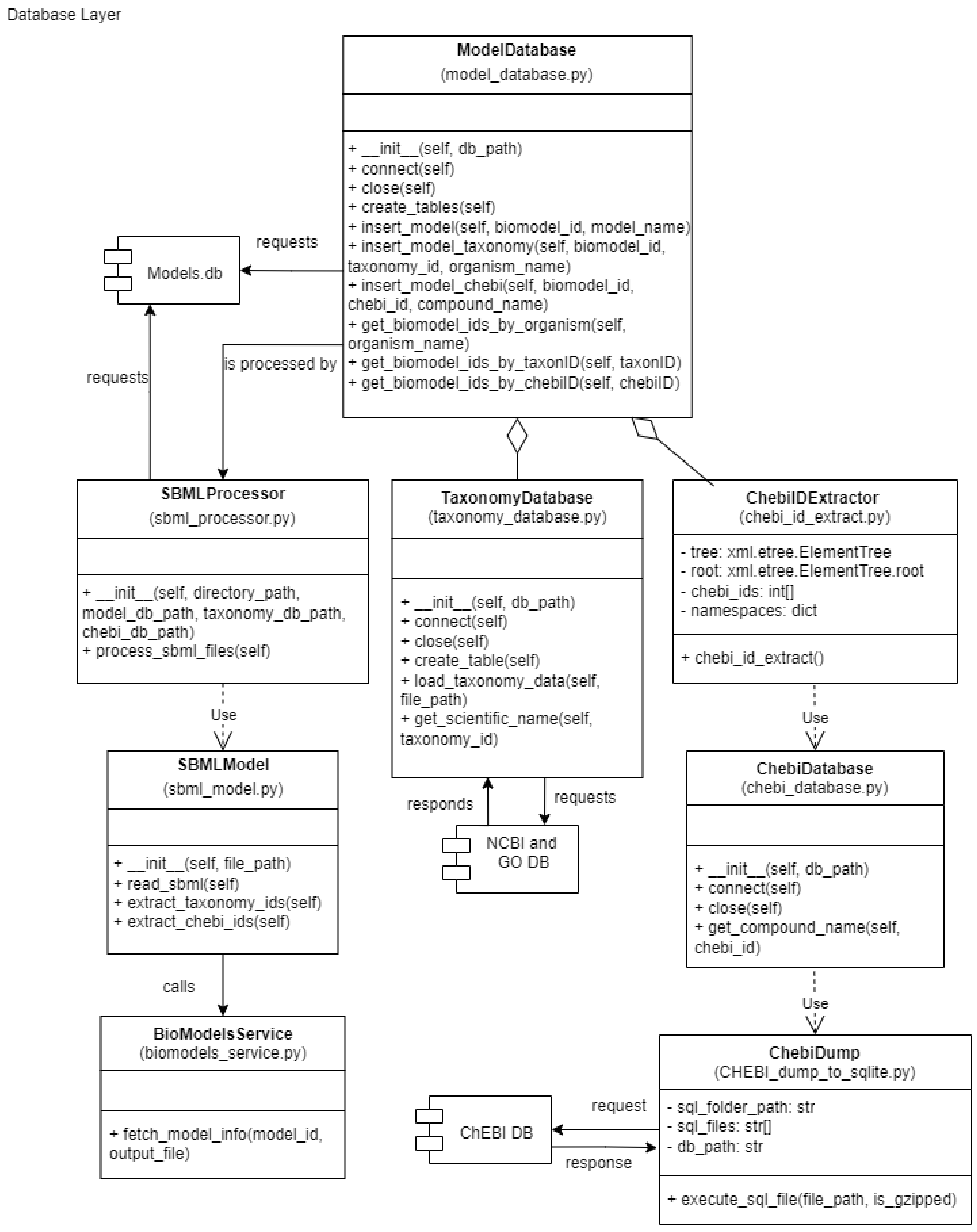
Database Layer
ModelDatabase
(model_database.py)
+ __init__(self, db_path)
+ connect(self)
+ close(self)
+ create_tables(self)
+ insert_model(self, biomodel_id, model_name)
+ insert_model_taxonomy(self, biomodel_id, taxonomy_id, organism_name)
+ insert_model_chebi(self, biomodel_id, chebi_id, compound_name)
+ get_biomodel_ids_by_organism(self, organism_name)
+ get_biomodel_ids_by_taxonID(self, taxonID)
+ get_biomodel_ids_by_chebiID(self, chebiID)
requests
Models.db
requests
is processed by
SBMLProcessor
(sbml_processor.py)
+ __init__(self, directory_path, model_db_path, taxonomy_db_path, chebi_db_path)
+ process_sbml_files(self)
TaxonomyDatabase
(taxonomy_database.py)
+ __init__(self, db_path)
+ connect(self)
+ close(self)
+ create_table(self)
+ load_taxonomy_data(self, file_path)
+ get_scientific_name(self, taxonomy_id)
ChebiIDExtractor
(chebi_id_extract.py)
- tree: xml.etree.ElementTree
- root: xml.etree.ElementTree.root
- chebi_ids: int[]
- namespaces: dict
+ chebi_id_extract()
Use
SBMLModel
(sbml_model.py)
+ __init__(self, file_path)
+ read_sbml(self)
+ extract_taxonomy_ids(self)
+ extract_chebi_ids(self)
responds
requests
NCBI and GO DB
Use
ChebiDatabase
(chebi_database.py)
+ __init__(self, db_path)
+ connect(self)
+ close(self)
+ get_compound_name(self, chebi_id)
calls
BioModelsService
(biomodels_service.py)
+ fetch_model_info(model_id, output_file)
Use
ChebiDump
(CHEBI_dump_to_sqlite.py)
- sql_folder_path: str
- sql_files: str[]
- db_path: str
+ execute_sql_file(file_path, is_gzipped)
ChEBI DB
request
response

Figure 7: Class Analysis Diagram of the Database Layer

all the SBML models. It is important to include this class so that the fetching and downloading of the models from BioModels is repeatable and thus will help users stay up-to-date with the changing entries to the online BioModels database.

- TaxonomyDatabase loads and manages taxonomy from the NCBI and Gene Ontology databases such that the user can query Neo4j with human-readable taxonomic identifiers. Likewise, ChebiDatabase manages chemical data. ChebiIDExtractor is also used to extract ChEBI IDs from SBML files.

Lastly, a brief description of some algorithms and data structures used:

3.3 Recursive Descent Parsing

The backend and database layer utilise XML libraries like libsbml and ElementTree which implement efficient recursive search algorithms so that the searches to through the XML formatted external database dumps.

3.4 Compression

In ChebiDump and File Management classes, the Gzip compression algorithm is used to ensure more space is saved during loading.

4 Implementation

4.1 Data Structures

While the data structures of the database layer were described in detail in Section 3 and given in the diagrams above, it is important to discuss why such data structures were chosen.

- Neo4j was chosen as the DBMS for the graph data store, because it is well supported and documented, it offers scalability and cloud sharing, it is available for free. It also comes with a built in query language and visualisation support and so it allowed an easy adaptation to the required pipeline (which needed visualisation and querying support.)

- An SQLite database was used for the auxiliary ModelDatabase, as it is lightweight and fast which was necessary for storing simple relations of only a few parameters, and having to read multiple database dump files which could take a long time.

- The user was provided the option to choose different schema for their models, and so the different schema allowed a more custom experience for each user, as they could choose what information to prioritise in their visualisation and database management. An example schema is viewable later on in this document.

4.2 GUI

The GUI for BioGraph is implemented in React and uses a client-server architecture to pass messages to the backend GraphAPI. This abstraction is typical of good OOP design practice. The GUI also adheres to design principles and makes it have a positive user experience, as it is both an intuitive, interactive and efficient way to interact with and carry out the entire pipeline of graph conversion, storage, processing, visualisation, management and querying. Without an all-in-one, interactive GUI in which the user can do everything via the GUI, the primary goals of the project would have thus not been met. React is used for building the frontend components quickly and in a consistent way, Chakra UI is used for a stylistic and dynamic look, and Vite is used as a platform for the server on which the frontend will run, and through which the user can access the GUI. Some of the core components of the GUI include:

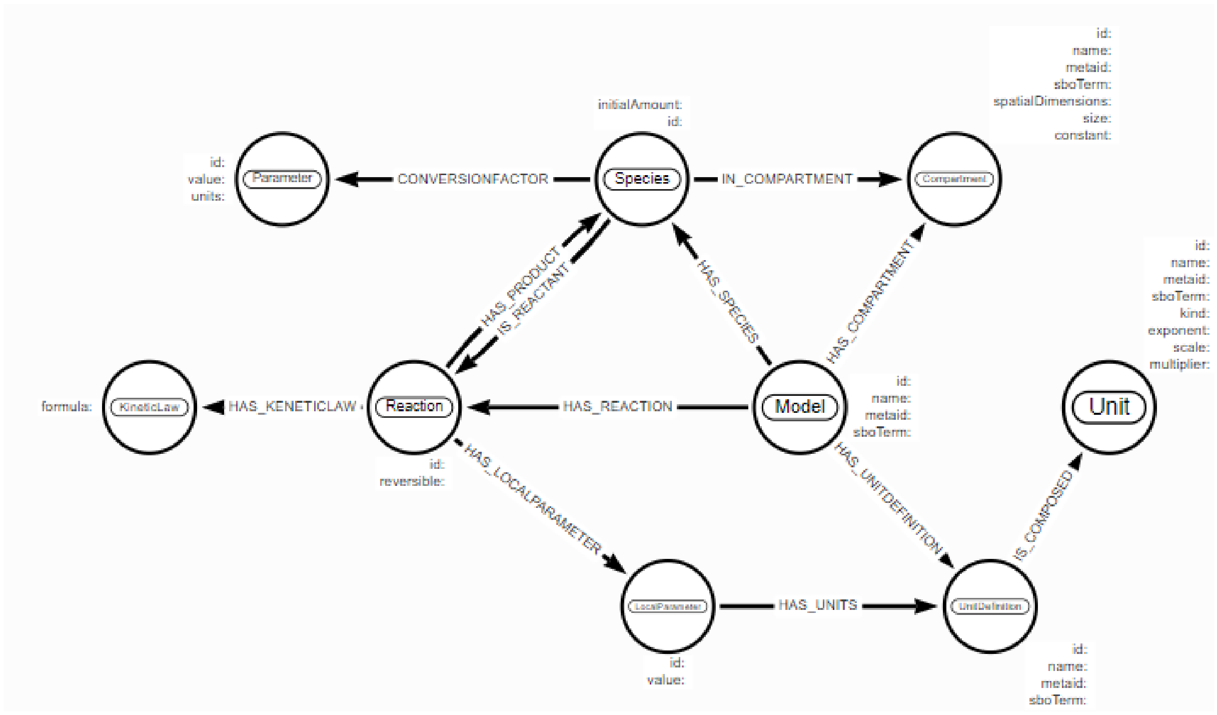

Figure 8: Schema Specification Diagram of an example schema in Arrows

- Routing. React Router is used to manage the navigation between different pages. A side menu is used to move to different functions like Import, Edit and Search by Given Terms
- Sidebar. This houses sections like "Import Model," "Graph Viewing Option," and "Edit Graph." Moreover, these sections are organised and minimalist, since they can collapse. This helps keep the UI clean and unclutters, which still allowing access to many features. This is in line with the 80/20 design principle, of making sure that the less used functionality is hidden in some way.'
- Integrated Graph Visualisation. This is also a core feature of the pipeline of the application. It uses a ForceGraph component to take in graph data and various, customiseable settings (like 3D View toggle, Motion Dynamics, and what arrows to show), which a tool that is well suited to them. The graphs are rendered dynamically using Node.js and users are able to move and manipulate the graphs interactively, even going as far as to click on nodes to show annotations.
- Hooks (State Managers). React hooks are used for managing states across all of the components. The useGraphData hook fetches graph data given the active filters), while simultaneously the useFilters hook tracks the states of the filters, thus allowing the GUI to be more reliable and follow ACID properties.
- API. Since the GUI communicates with the Flask server via an API client, using Axios to handle HTTP requests regarding graph operations. This level of abstraction in the logic of message passing makes sure that the GUI does not need to perform logical processing. Therefore, the GUI is cleaner and more responsive because the GUI only needs to handle rendering and message passing.
- Error Handling/Loading States. According to standard design principles, the app uses error handling mechanisms to communicate feedback upon errors being encountered due to a failed API request. Moreover, the use of loading states improves the user experience as they wait for data to be processed.

- Design. Lastly, the GUI adheres to the 7 Golden Rules of Mobile Design - that being minimalist, allowing for shortcuts with experienced users, obscuring the less used functions.

5 Program Validation and Verification

It is important to mention that testing for the system did not impact the design very much. This is because a lot of the code used came from libraries and packages that are already tested, and so the individual components did not need much testing.

The testing of BioGraph was informed by the test cases generated in the requirements capturing period of the project, updated as necessary. The aims of the project were used as benchmarks for the tests - to make sure that what was being tested had bearing on the final result. The aim of testing was to ensure the project was robust and performed well. Notably, because of the agile development approach, the Quality Management Plan was incremental and iterative to match the building of the project. Also, an emphasis was placed on user acceptance testing because of the agile development and frequent meetings with the client.
Five main tests were composed for the backend: test_get_graph.py, test_clear_graph.py, test_graph_api.py, test_merge_and_similarity.py, and test_subgroup_similarity_endpoint.py.

- **Unit testing** test_get_graph.py. This tests the get_graph endpoint by sending a POST request to the GraphAPI and verifying that the response matches the expected output. It tests all configurations of schema and filters on graph data (i.e. searching with null species and any unit). This is necessary, because message passing to the GraphAPI client is defined from scratch and so it is more prone to errors. This ensures the methods of the class behave as expected.

- **Functional testing** For a similar reason, test_clear_graph.py tests that the API endpoint for clearing graphs works correctly by sending a GET request to the API and the response should be the expected number (200).

- **Integration testing** test_graph.py. This test measures the accuracy of the find_similar_nodes endpoint, and checks that it received a similar response when similar node are queried. This checks that this part of the code is compatible within the wider environment of the application. This script contains unit tests for the Flask web application's graph API endpoint. The main functionalities tested include: Testing the '/api/graph' endpoint: The 'test_graph_endpoint_and_output_to_file' method sends a GET request to the '/api/graph' endpoint, then checks the response is correct (200) and that the data type is application/json, as well as other basic checks on the structure of the json. It verifies that the 'nodes' and 'links' keys are present in the response data and are lists. The test uses the unittest framework for testing.

- **Functional testing** test_subgroup_similarity_endpoint.py. This ensures that the functional requirements of the user is met - by testing that the subgroup similarity is indeed correct.

Moreover, **Validation testing** was conducted continuously over the period of interviews, by presenting the GUI at each stage of development, as well as presenting features in order of priority to the client. This was vital in ensuring that the requirements (both functional, non-functional and usability requirements were met). By asking specific questions, these interviews could be considered user acceptance testing. Consider an extract from the demo meeting held with the client, and the questions and answers exchanged, which gave insight into the GUI:
Question: What schema for conversion should we use for our neo4jsbml converter? Specifically which schema out of the examples listed in the neo4jsbml documentation should we be using? (Note: This question refers to the graphical representation of our models instead of how they are

to be stored.)
Answer: There is no specific schema that is better or worse, so in that sense it doesn't matter, but it might be a better idea to let the user pick which schema they'd like to use in their pipeline, as their requirements or preferences might be different.
Question: What do you want to see in our demo? Would you prefer more functional, fleshed out code that attempts to meet almost all the requirements, or could we rather give a more visual overview?
Answer: A mockup. Focus on fleshing out the high priority deliverables. This showed that the pipeline should be the focus of the project and also that the client did not like the single-schema GUI that was demonstrated, rather wanting the option for many.
Testing results are recorded below:

Table 1: Summary Testing Plan

Process	**Technique**
Class Testing: test_graph.py, test_get_graph.py	Unit tests and assertions on the Flask API endpoints
Integration Testing: Test interaction between API endpoints and Neo4j	Requests sent to `/get-graph`, `/clear-graph`, and merge endpoints, validating returned JSON structure and data correctness
Validation Testing: Confirm system meets user requirements such as in client meetings	Use-case tests to check graph retrieval, node and link validity, similarity function
System Testing: Ensure system-wide behavior, including error handling	Boundary tests, handling of edge cases such as invalid inputs or empty responses, stress testing under multiple requests

The tests focused on ensuring the core functionalities of the system, including graph operations, API integrity, and data consistency.

5.1 Discussion

BioGraph was tested comprehensively, not only for functional requirements and technical integrity but also how much it aligned with the client's expectations. By using the test cases developed early in the development, tests were easily and carefully designed fro specific core functions and points of known weakness. This ensured the backend operated correctly, whereas user validation testing made sure the requirements were satisfied for the client.

Unit tests were for the system to fetch and filter graph data depending on different schema combinations. This proves BioGraph will be able to handle the diverse models in the data set. It confirmed that querying and managing the graph data was reliable. The functional tests done testing the use cases of deleting and editing graphs, which were vital in having a good user experience. Moreover, the system's error handling was tested, and proven successful as the system has to manage many operations at once, and recover from errors quickly. The integration tests, such as test_graph.py, checked that Neo4j and GraphAPI interacted correctly - communicating the efficacy and stability of messages passed between the two. There is potential performance bottlenecks as the number of concurrent queries increased, but more testing is required. Luckily, the BioModels database is relatively small, and so extreme cases are not really relevant at this point. This also promoted adherence to the ACID properties. Lastly, the user acceptance testing was informed by the continuous feedback with the client, and shaped a

Table 2: Summary of tests carried out

Data Set and reason for its choice	**Normal Functioning**	**Extreme Boundary Cases**	**Invalid Data**
test_get_graph.py	Status code 200: Passed; Response content type 'application/json': Passed; Presence of 'nodes' and 'links' keys in response: Passed; 'nodes' and 'links' are lists: Passed; First node contains 'id' and 'label': Passed; First link contains 'source', 'target', and 'label': Passed; JSON response successfully written (0.203sec)	n/a	Fell over
test_clear_graph.py	test clear graph: Passed; Status code 200: Passed; Response message: "Graph deleted" confirmed;	n/a	Passed
test_graph_api.py	Passed	n/a	Fell over
test_merge_and_similarity.py	Passed	n/a	Passed
test_subgroup_similarity.py	Passed	n/a	Passed

lot of the final product. For example, the option to select a schema only came about because of a suggestion from the client after seeing the demo GUI. It also helped prioritise the most essential deliverables, mitigating the risk of a product with too many, badly executed features.

6 Conclusion

BioGraph succeeds both functionally and non-functionally. It meets the use cases generated, satisfies the functional requirements listed and has many additional features. Moreover, it is designed with agile software engineering principles, has a prominent focus on a positive user experience (highlighted by the success of the GUI), and is coded with good OOP principles such that the code is reusable and accessible. This makes BioGraph a robust, modular and well-structured solution to the client's brief.

7 User Manual

7.1 Introduction

BioGraph is a sophisticated tool designed to visualize, analyse, and interact with biological models from SBML files. It bridges the gap between the complex notation of SBML and practical model exploration, offering an intuitive interface for comprehensive visualization and interaction with biological data.

7.2 Getting Started

7.2.1 Understanding the Interface

- **Control Panel:** Located on the left, the control panel provides tools for interacting with and manipulating the graph.
- **Graph Visualisation Tools:** Positioned above the graph, these toggle switches enable users to adjust various visualization settings for enhanced graph analysis.
- **Graph Area:** The central area displays the visual representation of the SBML model.

7.3 Using BioGraph

7.3.1 Graph Visualisation

BioGraph offers six visualization toggles to customize how users view the SBML model:

1. **3D View:** Enables a three-dimensional representation of the graph. When selected, other visualization options are disabled to maintain the integrity of the 3D view.
2. **Text Nodes:** Displays nodes using their SBML names rather than shapes, providing clarity on node identity.
3. **Text Labels:** Shows the types of relationships between nodes, as defined by the schema, allowing for easier identification of connection types.
4. **Show Arrows:** Adds arrows to relationship links to indicate the direction of relationships, distinguishing between source and target nodes.
5. **Show Particles:** Animates particles along the relationship links, visually representing the flow and direction of the relationships.
6. **Highlight on Hover:** Highlights a node and its directly connected nodes and relationships when hovered over, improving visibility and understanding of node connections.

7.3.2 Control Panel

The control panel in BioGraph offers a systematic approach to incorporating SBML models into your workspace. It enables you to select schemas, import models, and configure graph visualizations. The following sections outline the key functionalities.

7.3.3 Import Models

1. **Select Schema:** The initial step is to select a schema, which determines how your graph is visualized. We highly recommend using the default schema, as it ensures optimal compatibility with the visualization and editing tools. Once selected, the schema remains active until you refresh your session or choose a different one.

2. **Import Options:** After selecting a schema, you can either:
   - **Search Biomodels Database:** Search for models in the Biomodels database based on several criteria:
     - **Biomodel ID:** Search by the unique identifier of a model.
     - **NCBI Taxonomy:** Find models based on organism taxonomy.
     - **Gene Ontology (GO):** Search models related to specific biological processes or functions.
     - **ChEBI (Chemical Entities of Biological Interest):** Search by chemical substances relevant to the model.

     The search can be conducted using names or IDs (except for Biomodels, which must be searched by ID). After searching, a list of SBML files will be displayed, allowing you to select which models to import.
   - **Local Import:** Upload an SBML file from your local machine into the workspace. Ensure you click "Import Graph" to add the file.

7.3.4 Graph Viewing Options

This section enables you to specify which types of nodes will be visualized in the workspace. The node types align with the SBML structure defined here. You can adjust the node types to match your analysis requirements.

7.3.5 Search Graph

The search function allows you to explore your workspace based on specific parameters:

- **Species:** Lists all models containing the specified species.
- **Compartment:** Displays models with a given compartment.
- **Organism:** Shows all models related to the specified organism.

To search for multiple species, compartments, or organisms, input multiple parameters in each respective field separated by a comma. The model node of the found graphs will be highlighted.

7.3.6 Edit Graph

The system offers various functionalities for manipulating and analysing graph structures.

1. **Find Similar Subgraphs:** Selecting the *Find Similar Subgraphs* button opens a modal that compares similarities across different models. The system evaluates and matches subgraphs based on their similarity levels. After reviewing the comparisons, users can select a desired match, which will initiate the merging subgraphs workbench. In this workbench, users can designate the specific nodes on which the graphs should be merged, facilitating seamless integration of similar subgraphs.

2. **Delete Node:** The *Delete Node* feature allows users to remove a specific node from the graph. To utilize this function, navigate to the *Delete Node* section, select the node intended for removal, and confirm the action. This process updates the graph by eliminating the chosen node.

3. **Merge Nodes:** The *Merge Nodes* functionality provides the capability to consolidate two specified nodes within the graph. To merge nodes, access the *Merge Nodes* option, select the two nodes to be combined, and execute the merge operation. This results in the integration of the selected nodes into a single entity.

7.4 Export and Delete

The system provides *Export* and *Delete* functionalities for comprehensive graph management.

1. **Export:** The *Export* button allows users to save the graph data in either JSON or CSV format. To export, click the *Export* button, select the desired file format (JSON or CSV), and confirm the export action.

2. **Delete:** The *Delete* option enables users to remove the entire graph from the system. To delete, select the *Delete* button and confirm the action when prompted. This process permanently clears all graph data.