ENGN3712
A competition web service for on-line attitude estimation algorithms

Author
Callum Woods
u5162452

Supervisor
Jochen Trumpf

November 12, 2014

Abstract

Attitude estimation is a method of estimating an object's orientation in space, and making adjustments to maintain a desired orientation. Filters, or algorithms, are used to convert sensor measurements into control feedback. To develop algorithms, the code is tested against datasets of recorded sensor information.

A web service that contains many datasets would be beneficial to attitude estimation algorithm developers, as it allows them to test their algorithms against many different datasets. A ranking of tested algorithms on the web service would be useful for members of the attitude estimation community, as it allows them to easily see the leaders in the field.

Contents

List of Figures

List of Tables

1 Introduction

An object's attitude is its orientation in space, with respect to an inertial frame. To be able to control attitude, an aircraft must be able to know its own orientation, and perform adjustments in real time. Various sensors are used to make measurements relating to the attitude. Filters, or 'algorithms' are then applied to the data to obtain useful information.

In order for effective algorithms to be developed, testing must be performed. Drones are flown in controlled environments, and measurements are made by mounted sensors. These measurements are stored as 'datasets', containing arrays of data corresponding to the various measurements. Multiple cameras track the drone in the testing area, and record its true attitude. By comparing the true attitude against the information obtained by applying algorithms to the datasets, the accuracy of the algorithms can be measured.

Developers of attitude estimation algorithms may not always have access to large sets of datasets to test their algorithms. Researchers who collect attitude measurements might not be able to test their data with many different algorithms. Also, those interested in the field of attitude estimation may want to keep up to date with the latest information, but not be able to perform research themselves.

A method for all of these groups to interact and collaborate is desired. An online web service can allow datasets and algorithms from different sources to be tested together. By ranking the algorithms against each other, members of the community can easily find the best algorithms for the criteria required, and the measurements made by the sensors.

The contributions made by the project are detailed in section 1.1.

1.1 Contributions and Project Description

The purpose of the project is to design an online web service for the purposes of ranking attitude estimation algorithms. The main contributions of this project are the finite state machine, the API, and the implementation selection for the design of the web service.

In the planning stage, various user requirements were identified and scored to develop design specifications. These specifications were then used to design the web service, so that no requirements were overlooked. The user requirement analysis and the design specification formation are shown in sections 3 and 4 respectively.

From the specifications found in section 4, a list of design metrics was developed. These metrics were used to perform implementation selections, in section 6.

The possible user actions for the web service were formulated in section 5.2, and used to create multiple possible implementations, shown in use flow diagrams. The structure of the web service was also designed, and is shown in section 5.3. A ranking of the various possible implementations was performed against the design metrics. A final selection for the implementation was made for each component. The implementation selection is shown in section 6.

The design for the database was developed. All necessary columns for each table were identified, and are shown with descriptions of the field, as well as the required type for the entry. This is shown in section 7.

From the selected use flow diagrams and composite structure diagram in section 5, a finite state machine was developed. The finite state machine gave a complete description of the use of the web service, showing every possible action considered. The finite state machine diagrams were separated by the action, in a similar manner to the use flow diagrams in section 5.2. Every state necessary for the completion of the actions is included in the state machine. The finite state machine diagrams are shown in section 8.

From the finite state machine, an API was developed for the web service. The API describes the back-end of the web service, detailing exactly what is performed for each state of the service. The API was separated by module, as determined in the composite structure shown in section 5.3. The API is shown in section 9. Each function in the API is shown with input variables, function, and returned/output variables described.

The project focused only on the design of the web service, with no actual implementation performed.

2 Background

2.1 Attitude Estimation

An object's attitude is its orientation in space, measured in reference to an inertial frame. Attitude of aircraft is often described in terms of pitch, roll, and yaw, measured on three axes as shown in fig. 2.1.

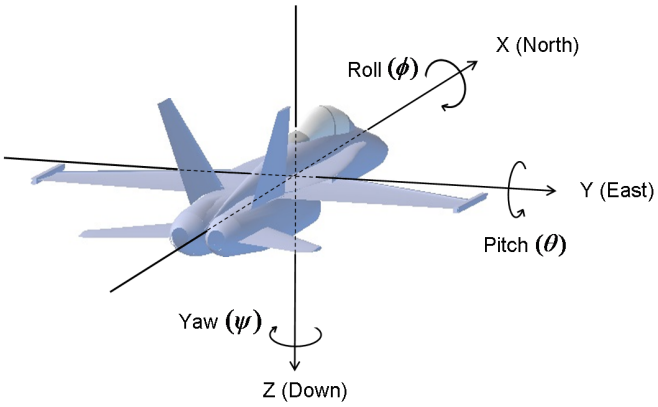

Figure 2.1: Roll, pitch, and yaw axes for aircraft (CHRobotics)

The attitude is measured using sensors on the aircraft. These sensors might include gyroscopes, accelerometers, horizon sensors, and star trackers, among others. (Crassidis et al., 2007). Information from these sensors feed back into the control systems to maintain attitude control. This control should occur in real time, so that the aircraft can maintain constant control.

Accurately measuring the attitude is especially important for unmanned aircraft, since there is no pilot to make manual adjustments if necessary. Because the reliability of the measurements is so important, multiple sensors are often utilised (Yadlin).

Attitude estimation is the process of taking information about the present and previous attitude measurements, and applying filters to determine the future attitude (Welch and Bishop, 1995). These filters, or 'algorithms', need to be able to quickly and accurately convert the measurement into useful data.

Among the most well known attitude estimation algorithms are the Kalman filter and its derivatives (Yadlin; Lefferts et al., 1982). The Kalman filter uses a recursive method of convergence to approximate an accurate value by minimising error. The Kalman filter is widely used because it provides a set of mathematical equations which can be used to estimate past, present, and future states of attitude (Welch and Bishop, 1995).

Developers of attitude estimation algorithms use datasets to test the algorithms. The datasets are recorded outputs from the different sensors. Drones are flown in a controlled environment. Sensors on the drone make measurements of the attitude, and record these measurements in data arrays known as 'datasets'. The specific syntax of the dataset is important for the algorithms, as the algorithms are developed to be able to apply to certain data formats. Also recorded is the true attitude of the drone, measured by

cameras tracking the drone in the controlled space. The data array of this measurement is called the 'observer' data. By comparing the outputs from the algorithms to the observer data, developers can quantify the accuracy of the algorithms (Crassidis et al., 2007).

2.2 Web Services

Information online is disseminated by web pages and web services. While web pages display only static content, web services are online services that display non-static content. A web service can perform functions, take input, store data, and display results.

A common web service design uses many states, and a web API to control state transfer. This type of web service is known as a 'Representational State transfer', or REST web service (Booth et al., 2004).

Web services are developed using web development modules. For this project, the python web development framework *flask* was chosen as the framework to build the web service. Flask is introduced in more detail in section 2.2.1. Also required for a web service to work online is a deployment program. For this project, the open source web server software *Apache* was chosen. Apache is discussed in more detail in section 2.2.2.

A web service that ranks data is known as a competition web service. Data on competition web services is most often displayed in a leaderboard. For the attitude estimation community, a competition web service ranking attitude estimation algorithms will provide a useful source of well-rated algorithms that meet the requirements of the researcher.

Some features of the web service for this project include user handling for user accounts and a database for storing dataset and algorithm information. Information regarding user handling is discussed in more detail in section 2.2.3. For this project, the relational database management system *MySQL* was chosen to manage the database. MySQL is discussed in more detail in section 2.2.4.

2.2.1 Flask

Flask is a python-based web developement framework. It is sometimes referred to as a 'microframework', because the level of complexity is low, lacking some components that other frameworks provide. However, it is possible to add these components to flask in the form of extensions. (Flask, 2014)

Flask is based on the Web Server Gateway Interface (WSGI) toolkit, and uses the Jinja2 template (Flask, 2014). WSGI is a python-based interface between web servers and web services, used for facilitating communication between the server and the program. Jinja2 is a template engine for python. It can be used to code in non-python languages

inside a python script. For example, in web development pages are most often coded using HTML. Using Jinja2, a developer can write a HTML page within a larger python script or function. Python variables can easily be parsed into HTML this way. (Jinja, 2014)

Flask was first developed in 2009 by Armin Ronacher. The first release was April 1st, 2010. (Flask, 2014)

2.2.2 Apache

Apache, also known as Apache HTTP Server, is an open source web server software. First developed in 1995, it is the most commonly used web server software in the world (The Apache Software Foundation, 2014). According to a web survey performed in June 2013, over 50% of top-level web servers on the internet use Apache (**?**).

2.2.3 User Handling

User handling is the capacity for a web service to manage users.

There are 4 main attributes of user handling: identity, user tracking, user permissions handling, and credentials. Some possible implementations for these attributes are outlined below, with a brief description. The implementation decision for the user handling components of the web service are shown in section 6.3.

(a) Identity

The identity of the user allows actions to be attributable. For example, without user identity, ownership of uploaded datasets and algorithms could not be established. Some possible implementations for user identity include:

- To assign an ID number based on join order
- The allow the user to choose their own username
- To let the server use an ID number, but to display usernames for user convenience
- To let the user sign in using an external ID.

  This is convenient for the user, as they will not be required to create an account to use the web service. The authentication is passed off to the external service for verification. However, users without these external IDs will still need to create an account, and so this implementation is not sufficient by itself.

(b) Tracking Users

Tracking users is necessary for the web service to be able to deliver the requested information to the correct user, as well as allowing the user to navigate the service. Some possible implementations for user tracking include:

- User agent downloads cookie to local machine, with user data stored on it.

  For this implementation, we must consider the possibility of certain browsers restricting cookies, or cookies being deleted.

- Using a stateless protocol for the service.

  Using this implementation requires additional information to be parsed for every action the user makes, since the web service does not keep track of previous information.

- Not tracking users, but requiring sign in for every upload.

  This would not require the service to keep track of the user between initial log in and taking some action which requires user attribution. Instead, when the action is taken, the user is required to verify their identity by signing in.

  This may be sufficient for uploading, but would be inconvenient for users if log in was required for every possible action (e.g. viewing some data).

(c) Permissions

User permissions are necessary if the capability to restrict access to certain data is desired. For example, if uploaders do not want their data availble to the general public but only a select few other users, a permissions implementation must be used that allows access to those chosen. Some possible implementations for permissions include:

- Permissions Table

  A permissions table is a table stored in the database, which contains a field for users, and a field for files. Each user has a relation to every field, in that they are or are not permitted to view, modify, or download it.

- A column in users database table with permitted datasets and algorithms.

  This approach is additive one, meaning users can only view pieces of data if the uploader of the data has given them permission.

- A column in the dataset and algorithm database tables with permitted users.

  This implementation is similar to the previous one, but with permissions acting on a per file basis instead of a per user basis.

  There is also the possibility of separate columns for viewing, modifying, and downloading datasets and algorithms.

- UNIX file permissions

  Files in UNIX-like systems have classes which affect who can view and modify the file. The three classes are the 'owner' class, which includes the owner or creator of the file; the 'group' class, which is a collection of users designated as part of the group (which may or may not include the owner); and the 'others' class, which includes all users not in the other classes.

  For each class, there are three permissions attributes: 'read', which allows a user to view the file; 'write', which allows a user to modify a file; and 'execute' which allows a user to run an executable file. These attributes are all independent, and can be set for any class in any combination.

  Permissions attributes are not inherited, meaning that files created inside a directory will not automatically have the same permissions as the parent directory.

(d) Credentials

User credentials allows for secure user accounts. Without credentials, user accounts could be accessed by anyone who know the identity of the account. Some possible implementations for credentials include:

- A user-selected password (possibly with requirements such as minimum length, included characters, etc.)
- A piece of hardware that works with the software to provide credentials.

  While very secure, this implementation has the drawback of requiring construction and distribution of hardware, which is both expensive and slow.

- Other physical credentials, such as fingerprint or iris scanning.

  Again, this implementation requires specialised equipment to be able to be recognised by the web service.

- No credentials at all.

  This implementation requires no additional coding, however it also provides no security against unauthorised access.

2.2.4 MySQL

MySQL is a relational database management system (RDBMS). Many graphical user interfaces (GUIs) exist for using MySQL to manage databases, as well as a set of command line tools. There are two versions of MySQL: an open-source edition, and a proprietary edition called MySQL Enterprise Server (MySQL, 2014a).

MySQL was first developed in 1995 by Michael Widenius and David Axmark, and is now owned by the Oracle corporation. As of 2014, MySQL is the second-most used database management system in the world (DB-Engines, 2014).

3 Customer Requirements

The customers for the project were identified, and are shown below:

1. Suppliers of Algorithms - the researchers developing the attitude estimation algorithms
2. Suppliers of Datasets - the researchers making measurements to create datasets
3. Users of Service - users that perform testing on algorithms and datasets, view the leaderboard of ranked algorithms, and download data
4. Jochen/Server
5. Callum/Developer
6. Development Community
7. Attitude Estimation Community

For each identified customer, the requirements were found. These were identified in this way to minimise the probability of missing some requirements.

As an example, the identified requirements for the suppliers of algorithms are shown here:

- Able to upload algorithms
- Unique user accounts
- Other users can comment on algorithms
- The service must have no cost

For the list of all of the identified requirements, see appendix A.1.

After all of the requirements were found, the list of identified customer requirements was organised into groups based on the type of the requirement. The groups were for Utility, Extra Features, Security, Implementation, and Documentation. The full list of sorted requirements in shown in appendix A.2. Each group of requirements was given a score out of five based on the importance of the group to the web service.

3.1 Ranking Customer Requirements

Each group was given a weighting based on the importance of the group. In addition, every requirement within each group was given a score of either 5, 3, 1, or 0, according to the following scheme:

- 5 – vitally important, service cannot exist without requirement
- 3 – important, service should include requirement
- 1 – useful, requirement would be good for service
- 0 – not required

The full list of customer requirements, with the group weightings, requirement raw scores, and the total scores, is shown in table 3.1.

The requirements were given a rank based on the score, out of 5. Under this ranking scheme, the lower values were more important than the higher values.

Table 3.1: Customer Requirements with Scores

No.	Requirement	Raw Score	Score	Rank
	Utility	**Weighting**	**5**	
1	Algorithms can be uploaded quickly	5	25	1
2	Large datasets can be uploaded quickly	5	25	1
3	Algorithms and Datasets can be downloaded quickly	5	25	1
	Leaderboard	**Weighting**	**5**	
4	Algorithms are ranked	5	25	1
5	Information on leaderboard is trustworthy	3	15	2
	Security	**Weighting**	**5**	
6	User actions are attributable to identified users	1	5	4
7	Upload and storage of data is secure	3	15	2
8	Suppliers can control permissions on datasets	3	15	2
9	The service is secure from unwanted access	3	15	2
	Implementation	**Weighting**	**3**	
10	Service uses Debian, flask, Apache, and mysql	5	15	2
11	Service can store large datasets	5	15	2
12	Project complies with the project boundaries	3	9	3
13	Scope is completable on time (hard deadline)	5	15	2
14	The service has no cost to users	3	9	3
15	The running costs are minimised	1	3	5
	Documentation	**Weighting**	**3**	
16	Documentation for assessment shows process for completion	5	15	2
17	Documentation for developer's use is clear and comprehensive	3	9	3
18	Documentation explains function and purpose	3	9	3
19	Standard terminology is used for ease of comprehension	3	9	3
	Extra Features	**Weighting**	**3**	
20	Other users can easily comment on algorithms/datasets	3	9	3
21	Users have reputation based on reliability	1	3	5
	User Accounts	**Weighting**	**5**	
22	New users can create accounts	5	25	1
23	Users can adjust account info	5	25	1

4 Design Specifications

The customer requirements were identified and given a score based on the relative importance. This can be seen in table 3.1. The customer requirements were converted into design metrics, with metric units.

Each metric was assigned a score for the importance, which was based on the rank of the customer requirement that the metric is based on. The metrics were given a score out of 5 which was inversely proportional to the rank of the requirement it was based on by the formula $Score = 6 - rank$. metrics was developed for these requirements. The full list of design metrics are shown in table 4.1.

In table 4.1, in the 'Units' column, the parameter 'List' means that the metric implementation will be selected from a list of possible options. 'Binary' means that the implementation either passes or fails, with no range in between.

Table 4.1: A list of metrics, including score and units

Met No.	Need No.	Metric	Score	Units
1	1, 2	Average upload speed	5	MB/s
2	1, 2	Worst case upload speed	5	MB/s
3	3	Average download speed	5	MB/s
4	3	Worst case download speed	5	MB/s
5	1, 2, 3	Service availability	5	Uptime
6	4	Leaderboard ranking criteria supported	5	List
7	6	User authentication measures	2	List
8	6	User actions attributable	2	List
9	7	Upload encryption standard supported	4	List
10	7	Storage encryption standard supported	4	List
11	8, 9	Users have list of datasets accessible	4	Binary
12	8, 9	Datasets list users which can access	4	Binary
13	8	Suppliers can approve access to data	4	Binary
14	9	Location for execution of code (algorithms)	4	List
15	10	Service implementation uses Debian	4	Binary
16	10	Service implementation uses flask	4	Binary
17	10	Service implementation uses Apache	4	Binary
18	10	Service implementation uses mySQL	4	Binary
19	11	Storage for datasets & algorithms	4	GB
20	1-3	Average time to complete requests	5	s
21	1-3	Worst case time to complete requests	5	s
22	13	Project deadline is met	4	Binary
23	14	Cost to user	3	AUD
24	15	Aggregate running cost per month	1	AUD
25	15	Overall cost of ownership	1	AUD
26	16-19	Documentation structure	3	List
27	16-19	Percentage of functions documented	3	%
28	20	Previous comments shown when replying	3	# comments
29	20	Comment character limit	3	Chars

5 Implementation Diagrams

This section outlines some of the possible implementations for the web service.

5.1 Labelling Scheme

The implementation diagrams are labelled according to the following scheme:

Type[A-Z] . Case[xxxx] . Implementation[a-z]

where

- Type[A-Z]

  Gives the type of diagram:

  - A: Use Flow Diagram
  - B: Composite Structure Diagram
  - C: Finite State Machine Diagram (shown in section 8)

- Case[xxxx]

  Describes the cases of the diagram:

  - supp: Use Flow for Supplier - Uploads datasets and/or algorithms, controls uploaded data
  - test: Use Flow for Tester - Tests algorithms against datasets
  - comm: Use Flow for Community - Views leaderboard, views/downloads datasets and algorithms
  - land: Use Flow to Landing Page
  - lead: Use Flow from Leaderboard
  - view: Use Flow from View Info state
  - resu: Use Flow from Results state
  - user: Adjust user account information

- Implementation[1-9]

  Gives the implementation of the diagram. Different values refer to alternate implementations.

e.g. A.supp.1 indicates a Use Flow Diagram showing the 'Supplier' use case.

5.2 Use Flow Diagrams

In the Use Flow diagrams, a rectangular node indicates an action, which may contain many states. An elliptical node indicates a state.

The flow between the states and actions are represented by arrows, labelled with the action being undertaken.

5.2.1 Use flow to Landing State

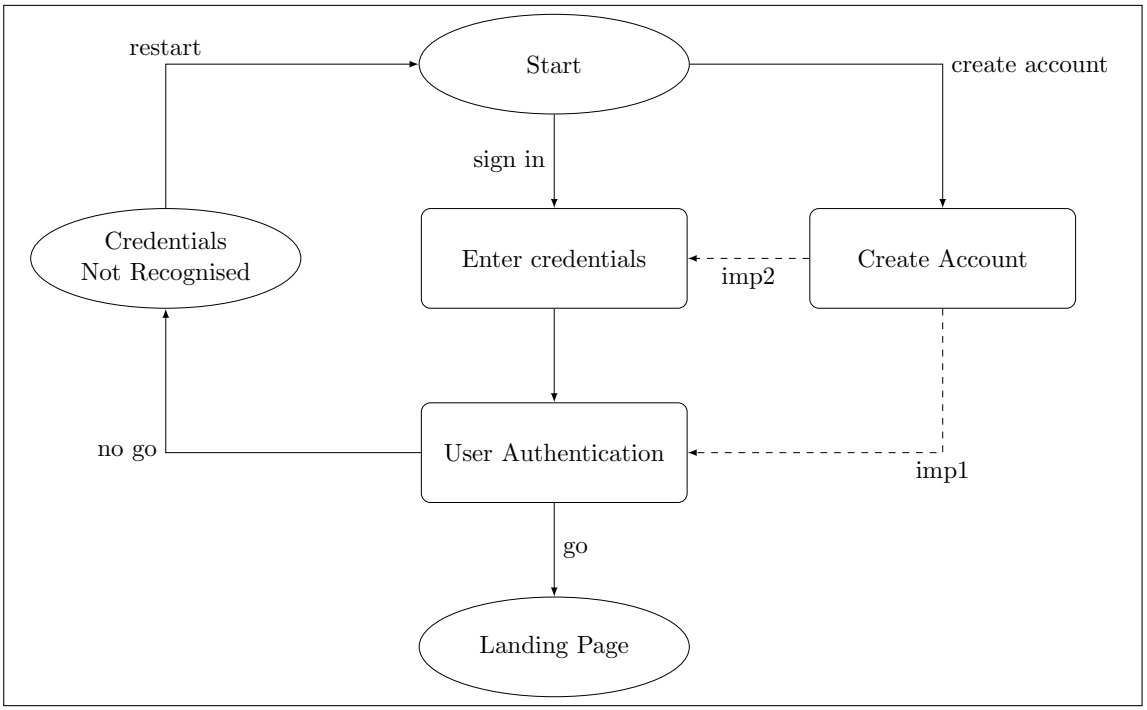

Figure A.land.1: Use Flow Diagram showing use flow to Landing Page

The diagram shown in fig. A.land.1 shows the high-level use flow from the start page of the web service to the landing page.

From the starting state, the user has two possible actions depending on whether or not the user has an account created.

If the user already has an account, then they take the 'sign in' action, from which they enter their credentials and the web service checks them in the 'User Authentication' block.

If the user does not already have an account, then they take the 'create account' action from which they enter the 'Create Account' block. Here the user can create an account by

supplying the relevant details and credentials. There are two possible implementations from this block: imp1 moves the user into the 'Enter Credentials' block, where the user must reenter the credentials used to create the account; imp2 moves the user directly to the 'User Authentication' block.

If the user credentials are authenticated, the user is moved to the 'Landing Page' state. If not, the user is moved to the 'Credentials Not Recognised' state, and hence back to the 'Start' state.

The state 'Landing Page' is the state from which many actions can be made to use the web service. These actions are shown in figs. A.supp.1 to A.comm.1. Specifically, the actions available from the landing page include:

- User: Adjust user account information (see section 5.2.2)
- Upload: Upload datasets and algorithms (see section 5.2.3)
- Test: Execution of algorithms on datasets (see section 5.2.4)
- View: Viewing the leaderboard of algorithms (see section 5.2.5)
- Search: Search datasets and algorithms (see section 5.2.5)

5.2.2 Use flow for user accounts

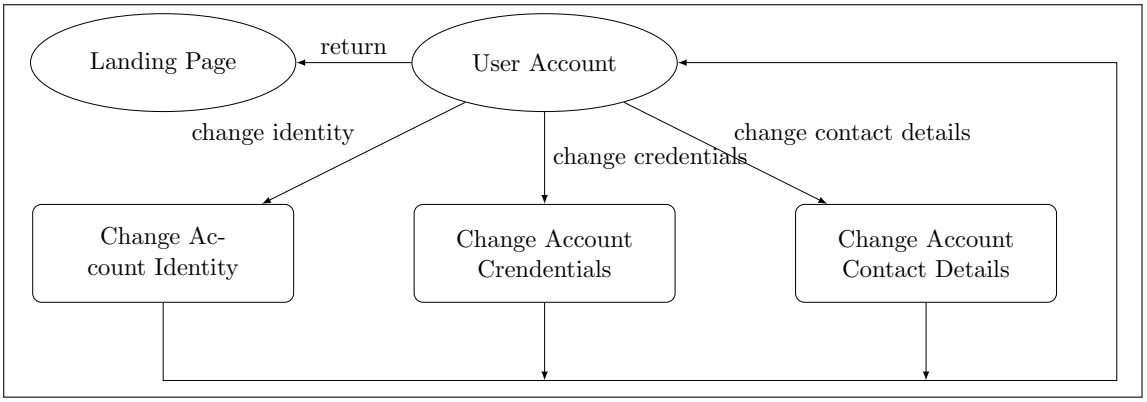

Figure A.user.1: Use Flow Diagram showing use flow for user accounts

From the landing page, users can enter the 'User Account' state.

From this state, users can adjust information relating to their account. The exact content that is changeable is implementation dependent: identity (e.g. username), credentials (e.g. password), and contact details (e.g. email address).

After any user information is changed, the user is returned to the user account page.

5.2.3 Use flow for Suppliers

The diagram for this use case is shown in fig. A.supp.1 in appendix B.1, and shows the high-level use flow for suppliers using the web service.

From the 'Landing Page' state, suppliers can upload either datasets or algorithms. The processes for each are both contained within the 'Upload Dataset/Algorithm' block.

From these processes, the uploaded data is committed to the mySQL database, and the user is returned to the landing page.

Another process from the landing page gives the supplier control over their own uploaded data.

The 'Control' state allows suppliers to control their own data which they have uploaded.

Users can adjust permissions on their datasets and algorithms in the 'Adjust permissions on uploaded data' block. This allows suppliers to control who has access to view and download their data.

From the control state, users can also return directly to the landing page.

5.2.4 Use flow for Testers

The diagram for this use case is shown in fig. A.test.0 in appendix B.2, and shows the high-level use flow for testers using the web service.

From the 'Landing Page' state, an algorithm to test, and a dataset to test the algorithm on, are selected. The tester has the opportunity to select the criteria against which to compare the result to previous tests. If the tester does not select any criteria, the default is to test against all applicable criteria.

The algorithm is executed on the dataset in the 'Execution of Algorithm' block. Depending on the implementation, this might include downloads/uploads by the user. Three possible implementations for this process are outlined in figs. A.test.1 to A.test.3.

After execution is completed, the results are compared to previous algorithms so that the tested algorithm can be ranked against them for the leaderboard. This occurs in the 'Comparison to Previous Tests' process. The comparison is made on the criteria selected in the 'Choose' block.

If at any of these stages and error occurs, the user is sent to the 'Testing Failed' state. From here, the user can see what the error that occurred was, and return to the landing page.

If no errors occur, the result of the test is inserted into the database, so that the algorithm can be ranked against all other tested algorithms. This occurs in the block 'Insert test

result into Database'.

Finally, the algorithm is ranked by applying an ordering to the database. The algorithms are ranked against a default criteria. This process occurs in the 'Produce ranked results list' block.

The user is then sent to the 'Leaderboard' state, with the default settings for that state (see section 5.2.7).

Local (Native) Execution The diagram for this use case is shown in fig. A.test.1 in appendix B.2, and shows the high-level use flow for testers using the web service, where the execution of the algorithm occurs locally (server-side).

This case is the most efficient in terms of speed and allows for testers to execute algorithms without first downloading them, but is the most vulnerable to malicious code.

A possible solution to this security vulnerability is to force datasets and algorithms to comply to a specific format. However, this may not be feasible for the algorithms, as they are executable scripts.

Local (Sandboxed) Execution The diagram for this use case is shown in fig. A.test.2 in appendix B.2, and shows the high-level use flow for testers, where the execution of the algorithm occurs locally (server-side), but sandboxed.

The 'Execution of Code' block has been expanded to include the additional steps of creating a box, importing the data, exporting the result back to the service, and deleting the box.

This case still allows testers to execute algorithms without first downloading them, while providing some protection from malicious code. However, a new sandbox must be generated for every algorithm that is executed, which may slow execution.

This is the most difficult implementation.

Client-Side Execution The diagram for this use case is shown in fig. A.test.3 in appendix B.2, and shows the high-level use flow for testers, where the execution of the algorithm occurs on the client-side.

The 'Execution of Code' block has been expanded to include the additional steps of downloading the algorithm and dataset ('Download Algorithm and Dataset' block), and the uploading of the results back to the web service ('Upload of Results' block). The actual execution in this implementation occurs on the client computer, and the process occurs in the 'Client-Side Execution of Algorithm' block.

This is the most secure case, as there is less opportunity for malicious code to affect the web service. However, this case requires that testers download the algorithm before they can execute it. The results must then be uploaded back to the web service.

The client computer must be able to run the tested algorithm. It may be that some client's computers can not execute the algorithm quickly, and this can slow down the process.

We must also account for the possibility of spoofing results. A check might be required to verify results, and this case is the least trustworthy for this reason.

Possible remedies for this include: peer review of results; requirement for multiple independent confirmations of results; or a reputation system to enable users to judge a tester's trustworthiness.

5.2.5 Use flow for the Attitude Estimation Community

The diagram for this use case is shown in fig. A.comm.1 in appendix B.3, and shows the high-level use flow for the attitude estimation community using the web service.

From the 'Landing Page' state, the user can view the 'Leaderboard' state, which shows the ordered algorithms sorted by a default criteria. The leaderboard state is shown in more detail in section 5.2.7.

An alternate action from the 'Landing Page' state allows the user to search the database for specific datasets or algorithms. First, the user selects the criteria by which they plan to find algorithms or datasets. This takes place in the 'Select Search Criteria' block. Next, the database is searched using the criteria chosen. This occurs in the 'Search Database' process. Finally, the user is shown a list of results, in the 'Display Result List' state. For more details about this state, see section 5.2.6.

5.2.6 Use Flow from Results State

The diagram for this use case is shown in fig. A.resu.1 in appendix B.4, and shows the use flow from the 'Results' state. This state displays a list of datasets and algorithms returned from searching the database (this is shown in section 5.2.5). Note that this list may be empty.

From this state, the user has the option to return to the landing page.

The user also has the option to select a result shown in the list. This action will place them in the 'View Info' state corresponding to the result selected. The possible use flows from this state are shown in section 5.2.8.

5.2.7 Use Flow from Leaderboard State

The diagram for this use case is shown in fig. A.lead.1 in appendix B.5, and shows the high-level use flow from the leaderboard state of the web service.

The actions available from this state include:

- Return: Return to the landing page
- View: View dataset/algorithm info. This state is shown in more detail in section 5.2.8.
- Change: Change criterion by which leaderboard is ranked. Users can select from different criteria, which will recreate the 'Leaderboard' state with the algorithms sorted by the new criterion.

5.2.8 Use Flow from View State

The diagrams in this section show the high-level use flow from the view state of the web service. There are two possible implementations shown in this section: fig. A.view.1 and fig. A.view.2, shown in appendix B.6.

An alternate implementation shown would be to have two View Info states; one for entry from the Leaderboard state, and one for entry from the Results List state. These two states would be identical to either implementation shown in fig. A.view.1 or fig. A.view.2, but instead of returning to the 'Landing Page' state, they would instead return to either the 'Leaderboard' or 'Results List' state, depending on the 'View Info' state. A use flow diagram for this implementation is shown in fig. A.alt.2

The diagram for this use case is shown in fig. A.view.1 in appendix B.6, and shows a simple use flow from the 'View Info' state.

The 'View Info' state displays information about the dataset or algorithm, including supplier, and date uploaded.

The actions available from this state include:

- Return: Return to the landing page
- Display: Display the dataset or algorithm. This places the user in a state in which the data or code is displayed. From this state, users can return to the 'View Info' state.
- Download: Download the dataset/algorithm. This process starts a file transfer from the web service to the user's computer, then returns to the 'View Info' state.

This implementation is only useful if we include a permissions check in the implementation set out in fig. A.comm.1.

The diagram for this use case is shown in fig. A.view.2 in appendix B.6, and shows a possible implementation of the use flow from the View Info state.

In this implementation, the View Info state displays information about the dataset or algorithm, including uploader, and date uploaded. Not shown in this state is the actual dataset or algorithm itself.

From the View Info state, users can return to the landing page, or attempt to display the dataset or algorithm. Before the data is displayed, the web service will check whether the user has permission to view the dataset, which occurs in the Check User Permissions block.

If the user does not have sufficient permissions to view the data, then they are redirected to the Not Permitted state, which informs the user about their lack of permissions and allows the user to return to the View Info state.

If the user does have sufficient permissions to view the data, then they are directed to the Display state. In this state, the dataset or algorithm is displayed. From here, the user has the option to download the data (in the Download block), or to return to the View Info state.

This implementation allows for a simpler implementation of fig. A.comm.1, with no need to include a permissions check.

5.2.9 Alternate Use Flow for multiple View Info states

The use flow shown in fig. A.alt.2 in appendix B.6 displays an alternate use flow covering the use flow for the attitude estimation community (fig. A.comm.1), the use flow from the 'Leaderboard' state (fig. A.lead.1), and the use flow from the 'Results List' state (fig. A.resu.1).

This alternative has two 'View Info' states; 'View Info (L)' for the 'Leaderboard' state, and 'View Info (R)' for the 'Results List' state. These two states are identical to each other apart from the state entered when taking the 'return' action.

Not included in the diagram is the use flow from the two 'View Info' states. The possible implementations are displayed in figs. A.view.1 and A.view.2, with one difference: instead of entering the 'Landing Page' as specified, 'View Info (L)' returns to the leaderboard, and 'View Info (R)' returns to the results list.

5.3 Composite Structure Diagrams

Nodes indicate components of the design. The area contained within the dashed boundary is the system. A solid arrow indicates flow of data. A dashed arrow indicates flow of control. For the data flow, the data contained inside square brackets is the data that is passed, and the arrows show the path that the data takes.

Implementation 1 is shown here. For implementation 2, see appendix B.7.

The data flow for implementation 1 is outlined below:

- User Sign-in

  User[id, credentials] → Webservice → User Authentication → Database

  Database[y/n credentials met] → User Authentication → Webservice → User

- Uploading Datasets and Algorithms

  User[data] → Webservice → Database

  Database[confirmation] → Webservice → User

- Testing Datasets and Algorithms

  A.test.1, A.test.2: User[request] → Webservice →Database

  Database[dataset, algorithm, previous results] → Webservice

  Webservice[result, comparison to previous] → Database

  Database[ranked list] → Leaderboard → Webservice → User

  A.test.3: User[request] → Webservice →Database

  Database[dataset, algorithm, previous results] → Webservice

  Webservice[dataset, algorithm] → User

  User[result] → Webservice

  Webservice[result, comparison to previous] → Database

  Database[ranked list] → Leaderboard → User

- View Leaderboard

  User[request] → Webservice → Database

  Database[ranked list] → Leaderboard → Webservice → User

- View Dataset or Algorithm

  User[request] → Webservice → Database

  Database[dataset, algorithm] → Webservice → User

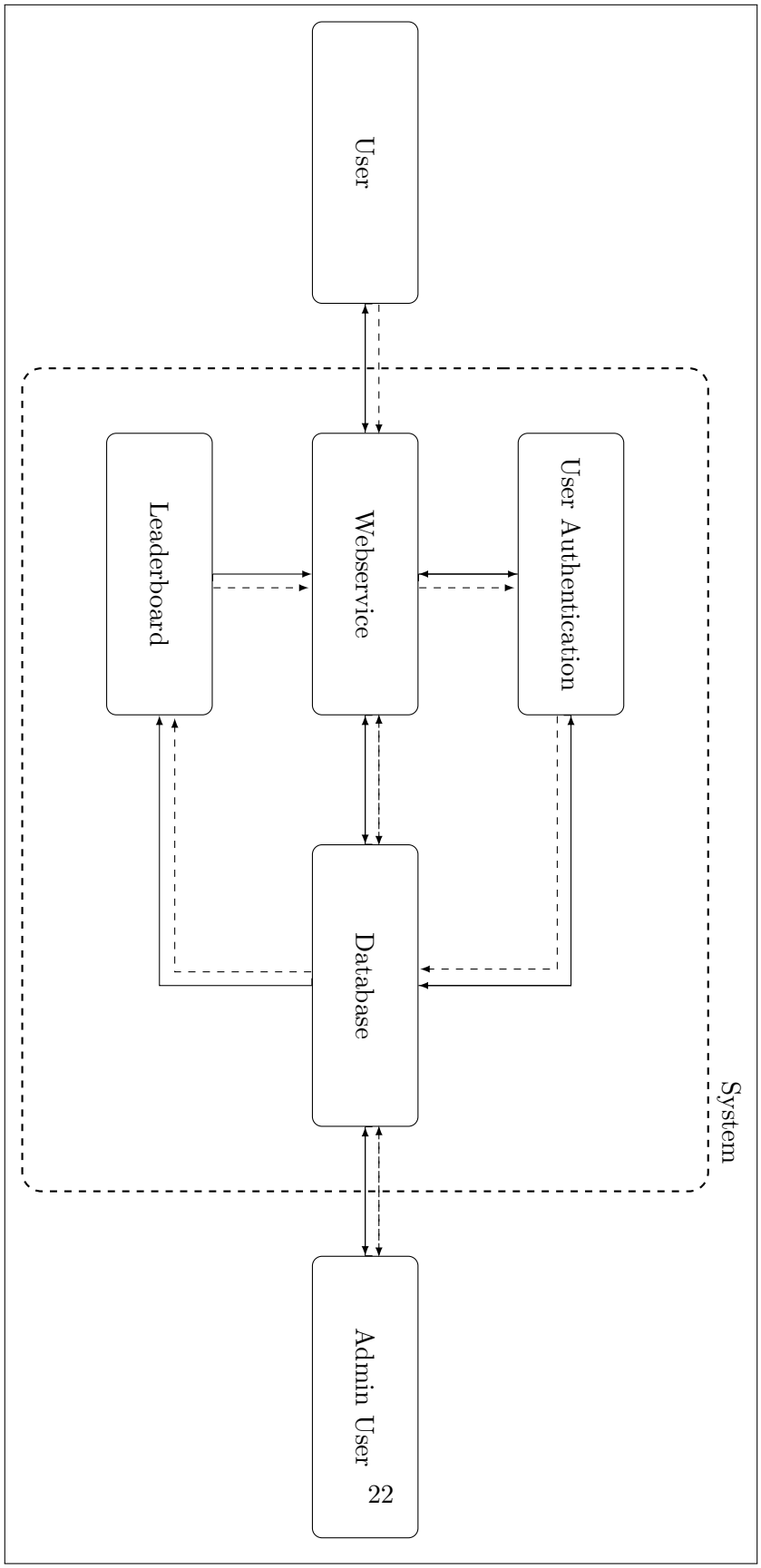

Figure B.1: Composite Structure Diagram Implementation 1

6 Implementation Selection

6.1 Implementation Decisions for Use Flow

In this section the possible implementations for different use flows are ranked against each other.

6.1.1 Use Flow to Landing Page

See fig. A.land.1 in section 5.2.1 for the use flow diagram showing the different implementations.

The difference in implementations for the use flow to the landing page occurs after the 'Create Account' action.

Implementation 1 has the user move directly from the 'Create Account' action to the 'User Authentication' action, while implementation 2 has the user move to 'Enter Credentials' first.

The fundamental difference between these two implementations one of different philosophies in user handling.

Implementation 1 is more convenient for the user, since it requires less input, and has one action less than implementation 2. However, it may pose a security risk, since it creates a code path from the start directly into the web service. This could potentially be exploited to allow access without entering credentials. This should also be the behaviour for the implementation passing straight through to the web service after signing up, but poor coding may allow such access.

Implementation 2 is less convenient for the user, but is more secure, as it prevents users from accessing the web service without entering credentials. Additionally, the inconvenience to the user only applies once, when the user first creates the account.

For these reasons, the chosen implementation is implementation 2.

6.1.2 Use Flow for Testers

See figs. A.test.1 to A.test.3 in appendix B.2 for the use flow diagrams showing the different implementations.

The implementations for the use flow for testers differ in the 'Execution of Algorithm' block. The difference is the location in which the chosen algorithm is executed on the chosen dataset.

The first implementation has the execution occur natively on the server. This case is the most efficient in terms of speed, and allows for testers to execute algorithms without first downloading them.

This implementation is also the most vulnerable to malicious code, as a poor management may allow algorithms to affect the web service. However, proper coding should minimise the chances of this occurring.

The second implementation has the execution occur in a sandbox on the server. This implementation still allows for testers to execute algorithms without first downloading them. The fact that the execution occurs in a sandbox should prevent malicious code from affecting the web service.

This implementation includes the additional steps of generating a new sandbox, moving the algorithm and dataset into the box, moving the result out of the sandbox, and deleting the sandbox. Because of this, the time taken for execution will be longer than implementation 1. This is also the most difficult implementation to code.

The third implementation has the execution occur on the client computer. This is the most secure case, as there is less opportunity for malicious code to affect the web service.

However, this implementation requires the user to download the algorithm and dataset, and upload the result back to the web service. This could negatively affect the total time for the execution process. We would also need to take into consideration the possibility of users uploading spoofed results. This possibility can be minimised by having peers review the results, having a requirement for multiple independent confirmations of results, or having a reputation system for users.

The implementation decision here was made by evaluating the possible implementations against the design metrics, which are shown in section 3. The full ranking can be seen in appendix C.

From our evaluation, we found that implementations 1 and 2 had the same total score. Since implementation 1 had the lowest single score, we chose implementation 2.

6.1.3 Use Flow from View Info State

See figs. A.view.1 and A.view.2 in appendix B.6 for the use flow diagrams showing the different implementations.

The first implementation has three options available from the 'View Dataset/Algorithm Info' state: return to the landing page, display the dataset/algorithm, or download the dataset/algorithm. Each of these actions will move the user directly to the relevant block.

Choosing this implementation means that the implementation for the use flow for the attitude estimation community requires a permissions check action.

This implementation is the easier to code.

The second implementation has only two: return to the landing page, or display the dataset/algorithm. If the user elects to display, then the web service checks the user permissions set to the data and compares it the the user id. If the user is not permitted to view the dataset/algorithm, then they are directed to a 'Not Permitted' state, and subsequently returned to the 'View Info' state. If the user does have permission to view the data, then the dataset/algorithm is displayed. The user also has the opportunity to download the dataset/algorithm.

Choosing this implementation allows for a simpler implementation of the use flow for the attitude estimation community, since we include the permissions check action here instead.

Of the two possibilities, this is the more difficult to code.

The fundamental difference between these two implementations is one of where a feature is implemented, as well as a philosophy regarding transparency vs. secrecy.

For a first implementation, the accounts permissions are a low priority. As a result, choosing the second implementation offers no major benefits over the first.

For this reason, the first implementation was selected for the web service.

6.1.4 Use Flow for Attitude Estimation Community, Leaderboard, and Results List States

See figs. A.comm.1 to A.lead.1 and A.alt.2 in appendices B.3 to B.6 for the use flow diagrams showing the different implementations.

The fundamental difference between these two possible implementations is the inclusion of multiple 'View Info' states. These states are identical apart from the state the user is sent to if the take the 'return' action. In the first implementation, the user is always sent back to the landing page. In the second implementation, the user is sent back to the state from which they entered the 'View Info' state.

The first implementation is slightly simpler to code, since it requires only one 'View Info' state which will have multiple states leading to it. However, it may inconvenience users by requiring them to search the database, or sort the leaderboard, every time they wish to view a different dataset or algorithm.

The second implementation is slightly more complex to code, since we require a distinct 'View Info' state for each state that leads to it. Since these states are all functionally identical apart from the state the user is sent to when they return, much of the code can be repeated for each state.

This implementation will also be more convenient for users who wish to view multiple datasets or algorithms, as they do not have to redo their actions to arrive at the state every time they wish to view a new one. For users who wish only to view one dataset, this implementation will require only one additional state to return to the landing page, a minor inconvenience.

Since the potential increase in convenience is much greater than the potential decrease in convenience, and because the additional coding is not significant, we selected the second implementation for the web service.

6.2 Implementation Decisions for Structure

In this section the possible implementations for composite structure are ranked against each other. The two possible implementations are shown in figs. B.1 and B.2 in section 5.3.

There were two possible implementations for the web service structure. The main difference between them is the inclusion of a 'Filter' state between the components and the database. This state prevents all access to the database unless the appropriate user authentication measures are met.

The additional steps of passing through the 'Filter' state may add some extra time to the use of the web service. However, it will make the web service more secure in that data stored in the database can not be accessed without permission. This should also be the behaviour for the structure without the filter, but poor coding may allow such access.

However, the inclusion of the filter means that every action that accesses the database includes multiple additional steps. This could affect the time taken for requests to be returned, as well as being more difficult to code.

Since the project was behind schedule, it was decided that the first implementation would be used. Since a working service is a requirement of the project, a simpler implementation will reduce the time taken to code. Proper coding should overcome the shortcomings of the implementation, namely, the risk of unauthorised access to the database.

6.3 Other Implementation Decisions

In this section implementation decisions that do not relate to the use flow or the structure of the web service are made.

6.3.1 User Authentication

The four main attributes of user authentication, and the possible implementations, are listed below:

(a) Identity

Options here include a unique ID number, a username, and external IDs. External IDs are identifications from other services, such as Facebook or Google accounts.

Possible implementations are:

- To assign an ID number based on join order
- The allow the user to choose their own username
- To let the server use an ID number, but to display usernames for user convenience
- To let the user sign in using an external ID.

The selection was made that users can choose their own (unique) username, as well as being assigned a user ID number for internal use. This allows for easily remembered identification for users, as well as simple handling from the server.

This could cause a security problem, where users might guess id numbers, which may allow for unauthorised access if coded poorly.

(b) Tracking Users

Options here include using cookies, using a stateless protocol, or not tracking but requiring additional verification steps.

Possible implementations are:

- User agent downloads cookie to local machine, with user data stored on it.
- Using a stateless protocol for the service.
- Not tracking users, but requiring sign in for every upload.

The most simple to code implementation that gives us the functionality we require is that of using cookies with user information, stored on the user's computer. For this reason, this was our chosen implementation.

(c) Permissions

Options here include using a permissions table stored in the database, using UNIX system file permissions, and using consolekit.

Possible implementations are:

- A column in users database table with permitted datasets and algorithms.
- A column in the dataset and algorithm database tables with permitted users.
- Using groups for access, as in UNIX systems.

  UNIX-based file systems have different classes of access: the 'owner' class, the 'group' class, and the 'others' class.

  We could combine the philosophy here with either of the previous two possible implementations, for example 'owner' could modify, 'group' could view or download, and 'others' could not view the data.

The second implementation here makes more sense, because suppliers will be adjusting data permissions on their own data, so it is more straightforward to adjust directly in the dataset or algorithm database.

Having separate fields for different possible actions is also desirable, so that users can be granted different levels of access to data.

It is also desirable to be able to group users, so that all users in a large group can easily be given permissions without being required to add each user individually.

(d) Credentials

Options here include a password, certificates, physical credentials, or no credentials.

Possible implementations are:

- A user-selected password (possibly with requirements such as minimum length, included characters, etc.)
- A piece of hardware that works with the software to provide credentials.
- Other physical credentials, such as fingerprint or iris scanning.
- No credentials at all.

It was decided that a password would be the simplest to implement, and that using a password would provide sufficient security against unauthorised access, provided it was chosen well.

7 Database Design

The web service will use four separate tables: a Users table, a Groups table, a Dataset table, and an Algorithm table. For each table, the columns are listed, with the data stored, explanations in parentheses, and the datatype in square brackets.

The tables of the database are linked together, with connections between the different tables. Specifically, the Groups table has a field for users, which contains relations to the Users table. The Dataset and the Algorithm tables have columns for users or groups with specific permissions. These columns contain relations to either the Users table or the Groups table. A diagram showing the relationship between the database tables is shown in fig. 7.1. In this diagram, arrows indicate relations.

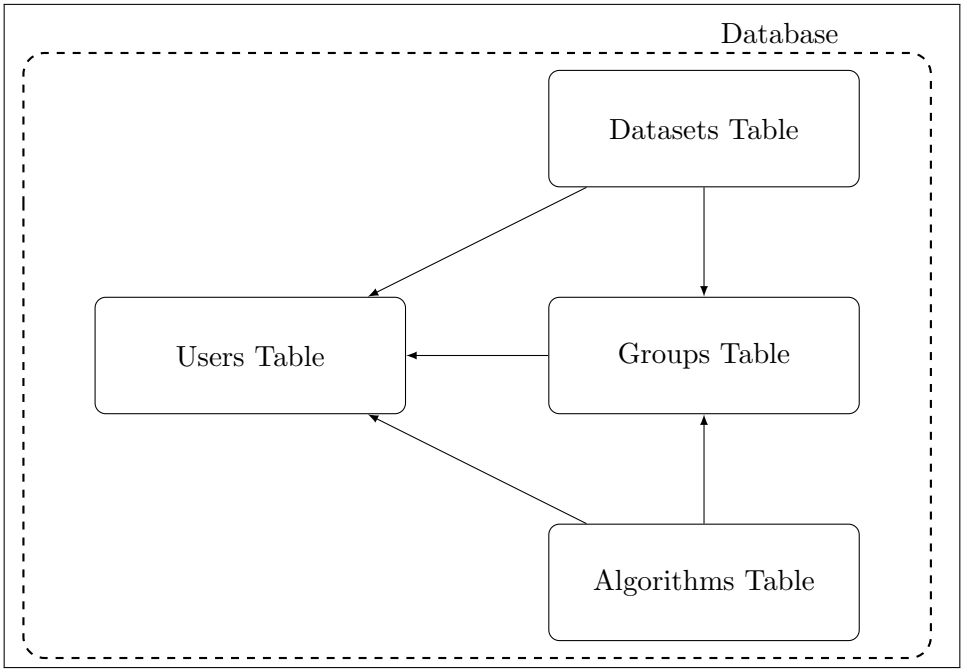

Figure 7.1: Relationship between Tables in the Database

The database is designed to be implemented using mySQL.

The web service is designed to store dataset and algorithm files in directories on the server. To make the files secure, it is necessary to obfuscate the file names. As a result, the true name is stored in the relevant database table, as is the pathway to the file, with the encrypted name.

The 'type' parameters are standard MySQL data types, as described in MySQL (2014b).

7.1 Users Table

This table lists all registered users of the web service. Each record contains a user ID for use by the server, a username for ease of identification by human users, the credentials required for the user to log in, and information used for verification and tracking of users. The user ID is assigned by the server when the user first creates an account.

The username, credentials, and user information are supplied by the user themselves upon sign up. The credentials and user information can be changed later by the user, but the username cannot be changed. The design for the columns for this table are shown in table 7.1.

Table 7.1: Columns for Users Table

Col #	Field	Description	Type
1	User ID	(number)	[INTEGER]
2	User name	(unique)	[CHAR]
3	Credentials	(password)	[CHAR]
4	Information	(email, etc.)	[VARCHAR]

7.2 Groups Table

This table contains lists of users, which are grouped together. Each record contains a group ID number for use by the server, a group name for use easy calling by users of the web service, and a list of all users in the group.

A separate database table for groups was chosen instead of a groups column in the users table to simplify the design. With this implementation, permissions for datasets and algorithms can be relations to either the users table, the groups table, or both.

The group ID is assigned by the server when the group is created. The group name, and the users in the group, are assigned by the creator of the group upon creation. These can be changed later. The design for the columns for this table are shown in table 7.2.

Table 7.2: Columns for Groups Table

Col #	Field	Description	Type
1	Group ID	(number)	[INTEGER]
2	Group name	(unique)	[CHAR]
3	Users in group	(list of User Table records)	[RELATION]

7.3 Dataset Table

This table contains information about datasets stored on the server. Each record contains the dataset name (which should be unique), the location of the dataset file on the server, the user who uploaded the dataset, information parsed from the dataset itself, and lists of users or groups with permission to perform actions on the dataset.

When the supplier of the dataset uploads the dataset to the web service, they supply the dataset name, and set the users/groups with permissions to view, download, or modify the dataset. The dataset file location, and the supplier of the dataset, are added by the web service. The description of the dataset, and the parameters time_series_length, num_time_series, time_series_semantics, and time_series_types, are parsed from the dataset itself.

The design for the columns for this table are shown in table 7.3.

Table 7.3: Columns for Dataset Table

Col #	Field	Description	Type
1	Dataset name	(unique)	[CHAR]
2	Dataset file location	(pathway to directory)	[VARCHAR]
3	Supplier	(User Table record)	[RELATION]
4	Description	(text, incl. punctuation)	[VARCHAR]
5	time_series_length	(number)	[INTEGER]
6	num_time_series	(number)	[INTEGER]
7	time_series_semantics	(list of strings, length is num_time_series)	[LIST]
8	time_series_types	(list of strings, length is num_time_series)	[LIST]
9	Users w. permission to view	(list of Group and/or User Table records)	[RELATION]
10	Users w. permission to download	(list of Group and/or User Table records)	[RELATION]
11	Users w. permission to modify	(list of Group and/or User Table records)	[RELATION]

7.4 Algorithm Table

This table contains information about algorithms stored on the server. Each record contains the algorithm name (which should be unique), the location of the algorithm on the server, the user who uploaded the algorithm, information parsed from the algorithm itself, and lists of users or groups with permission to perform actions on the algorithm.

The algorithm file should have two classes: *1*) the compatibility class, detailing the semantics and syntax required from the dataset for the algorithm to be able to run; and *2*) the algorithm class, containing the code for the algorithm itself.

When the supplier of the algorithm uploads the algorithm to the web service, they supply the algorithm name.

The algorithm file location, and the supplier of the algorithm, are added by the web service. The description and the purpose of the algorithm, and the parameters *Methods* and *Imports*, are parsed from the algorithm file itself. The parameters *required_semantics* and *required_syntax* are parsed from the compatibility class of the algorithm file.

The design for the columns for this table are shown in table 7.4.

Table 7.4: Columns for Algorithm Table

Col. #	Field	Description	Type
1	Algorithm name	(unique)	[CHAR]
2	Algorithm file location	(pathway to directory)	[VARCHAR]
3	Supplier	(User Table record)	[RELATION]
4	execution_time	(float)	[FLOAT]
5	Description	(text, incl. punctuation)	[VARCHAR]
6	Purpose	(text, incl. punctuation)	[VARCHAR]
7	required_semantics	(list of strings)	[LIST]
8	required_syntax	(list of strings)	[LIST]
9	Required imports	(required python modules)	[LIST]
10	Users w. permission to view	(list of Group and/or User Table records)	[RELATION]
11	Users w. permission to download	(list of Group and/or User Table records)	[RELATION]
12	Users w. permission to modify	(list of Group and/or User Table records)	[RELATION]

8 Finite State Machine

8.1 Labelling Scheme

The finite state machine diagrams in this section show the states (nodes) and the actions (paths) for using the web service.

Diagrams The finite state machine diagrams are labelled according to the following scheme:

Type[A-Z] . Action[xxxx] . Implementation[1-9]

where

- Type[A-Z]

  Gives the type of the diagram:

  - C: Finite State Machine Diagram

- Action[xxxx]

  Describes the user actions for the diagram:

  - sign: Sign In and Sign Up actions
  - user: Change User Account Information action
  - lead: View the Leaderboard action, including changing sorting criteria
  - data: Upload Dataset to web service action
  - algm: Upload Algorithm to web service action
  - ctrl: Control Uploaded Data (datasets/algorithms) action
  - test: Test Dataset and Algorithm action

- Implementation[1-9]

  Gives the implementation of the diagram. Different values refer to alternate implementations.

8.2 Finite State Machine Diagrams

The finite state diagrams, as well as a brief explanation of the states involved, are shown in this section. The states that are visible to the user have solid borders, while those states which are invisible have dashed borders.

For the user to move from a visible (solid) state, they must manually undertake an action. For invisible (dashed) states, the user is automatically moved once the server action is completed.

The functions associated with the state transfer actions are shown in section 9.

In this section, a full explanation of the finite state machine diagram is included for fig. C.sign.1, as an example to demonstrate how the finite state machine works. For the other state machine diagrams, only basic explanations are supplied.

8.2.1 Sign In, Create Account

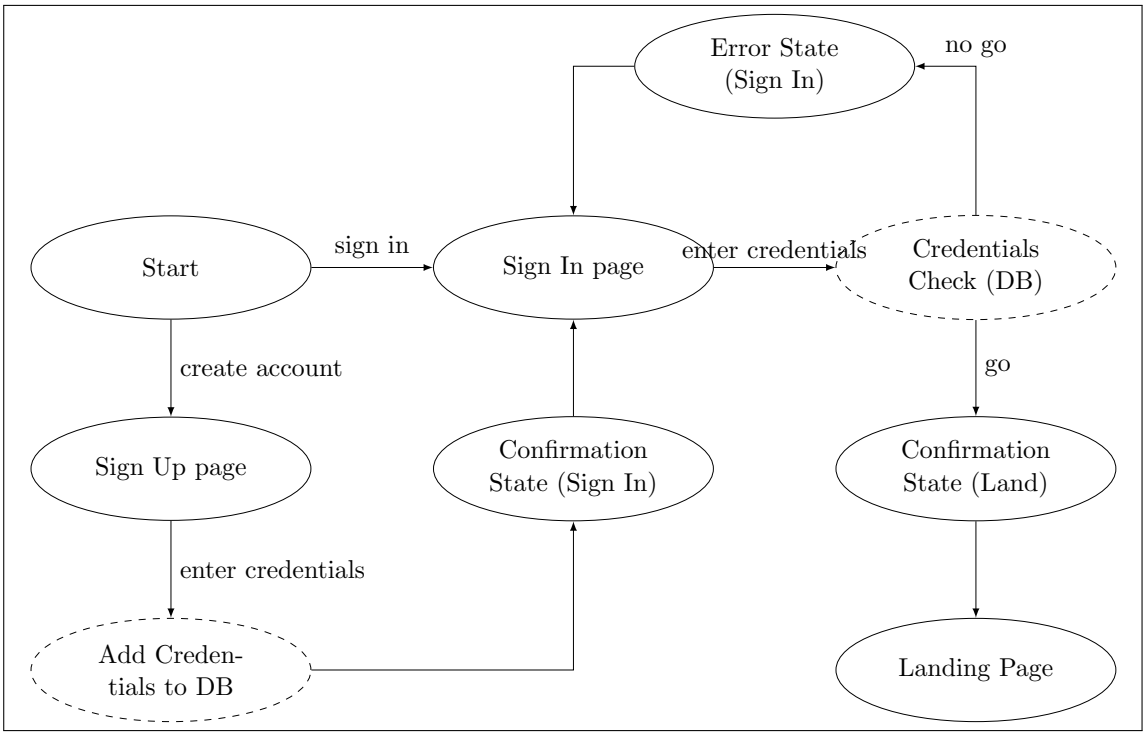

Figure C.sign.1: Finite State Machine for Sign In and Sign Up user actions

The 'Start' state is the state that is shown to the user when they navigate to the URL corresponding to the web service. From here, users have to option to sign in using an existing account, or to create a new account.

To create an account, the user is sent to the 'Sign Up Page' state. The user enters information for user identity and credentials. This information is added to the database in the 'Add Credentials to DB' state, and a confirmation of successful account creation is displayed to the user in 'Confirmation State (Sign In)'. Upon acknowledgement by the user, they are sent to the 'Sign In Page' state.

The user is also sent to the 'Sign In' state when they choose the sign in action from the 'Start Page' state. In this state, the user enters their user identity and credentials. The web service checks the Users table of the database for a matching entry. If no match is found, the user is sent to the 'Error State (Sign In)', where an unsuccessful sign in message is displayed. Upon acknowledgement by the user, they are sent to the 'Sign In Page' state.

If a match is found in the Users table, a confirmation of successful sign in is displayed to the user in 'Confirmation State (Land)'. Upon acknowledgement by the user, they are sent to the 'Landing Page' state.

The state 'Error State (Sign In)' returns the user to the 'Sign In Page' state. The state 'Confirmation State (Sign In)' returns the user to the 'Sign In Page' state. The state 'Confirmation State (Land)' returns the user to the 'Landing Page' state.

8.2.2 Change Account Info

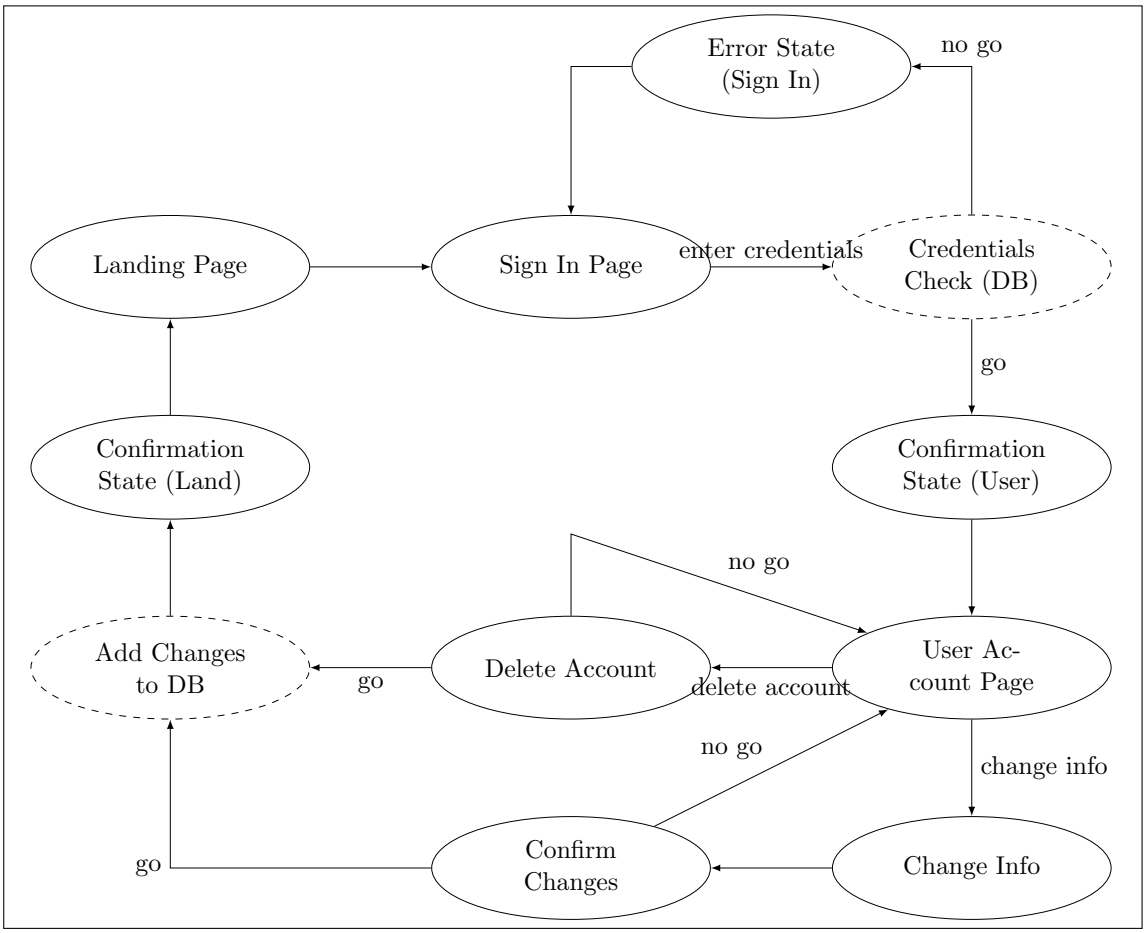

Figure C.user.1: Finite State Machine for Change User Account Info user action

The state 'Sign In Page' is included here so that users cannot have their account information altered unless they can authenticate themselves (i.e. prevents changing info from say leaving account logged in).

8.2.3 View Leaderboard

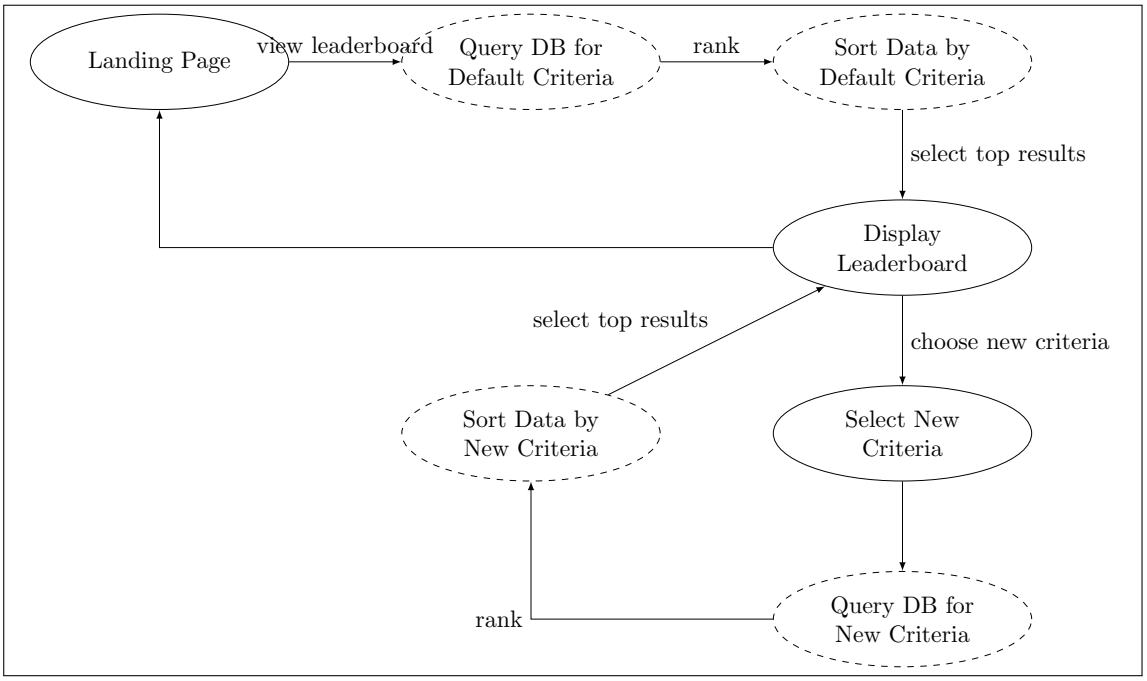

Figure C.lead.1: Finite State Machine for View Leaderboard user action

The 'Query DB for Default Criteria' action searches the database for all datasets or algorithms that meet the default criteria (this can be changed later). The 'Sort Data by Default Criteria' action ranks the returned datasets/algorithms in order of the default ranking criteria (). The best of dataset/algorithms is then displayed in the 'Display Leaderboard' state.

8.2.4 Upload Datasets

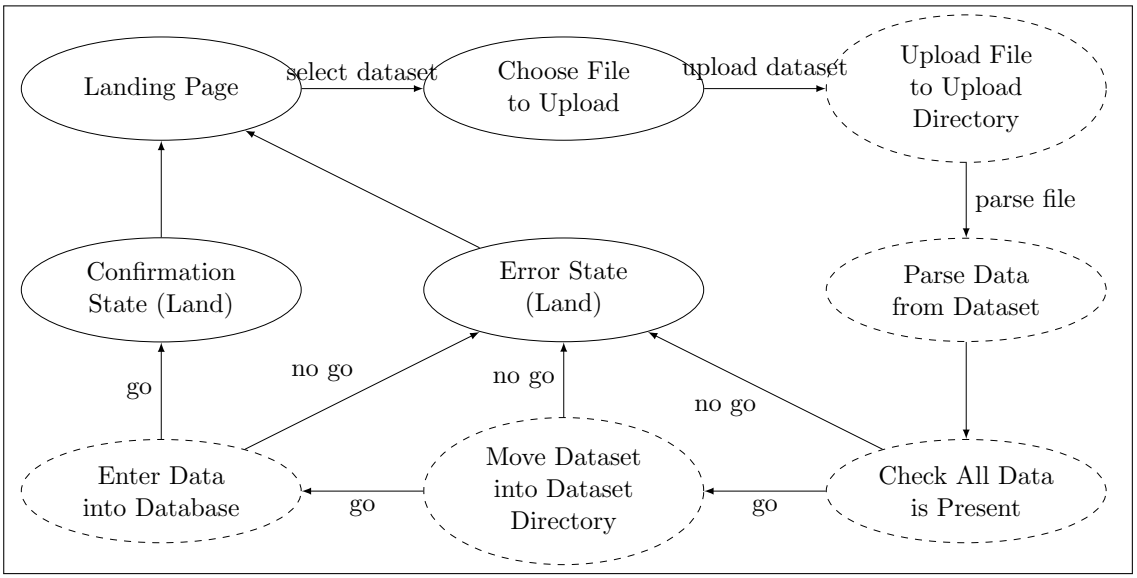

Figure C.data.1: Finite State Machine for Upload Dataset user action

The state 'Error State (Land)' returns users to the 'Landing Page' state. When an error occurs ('no go'), the user is moved to this state, along with information concerning what caused the error to occur.

For the 'Check All Data is Present' state, the information expected is listed below:

- 'START FILE' and 'END FILE' flags
- time_series_length
- num_time_series
- time_series_semantics
- time_series_types
- series_1, series_2, ..., series_(num_time_series)

The following checks are also performed:

- The lengths of the series is equal to time_series_length
- Each series has 'data_start' and 'data_end' flags
- Each series gives semantics and syntax
- Given semantics and syntax matches those in time_series_semantics and time_series_types respectively

8.2.5 Upload Algorithms

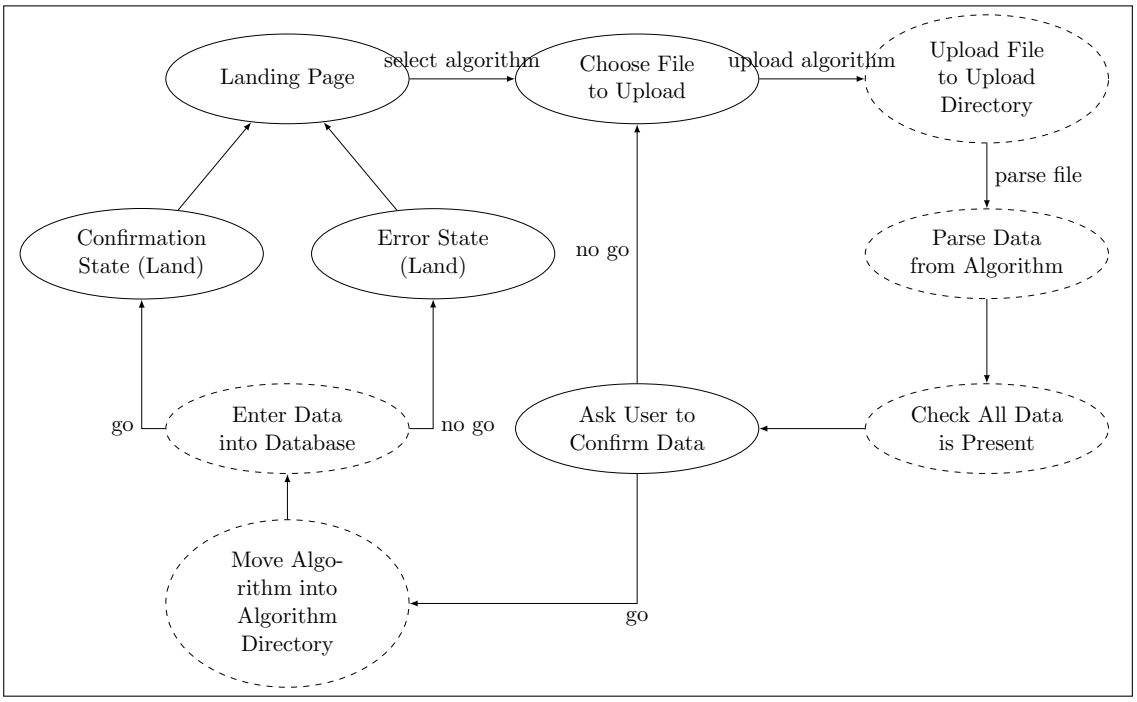

Figure C.algm.1: Finite State Machine for Upload Algorithm user action

For the 'Check All Data is Present' state, the expected data is listed below:

- required_semantics
- required_shapes
- algorithm_purpose
- algorithm_methods
- algorithm_imports

8.2.6 Control Uploaded Data

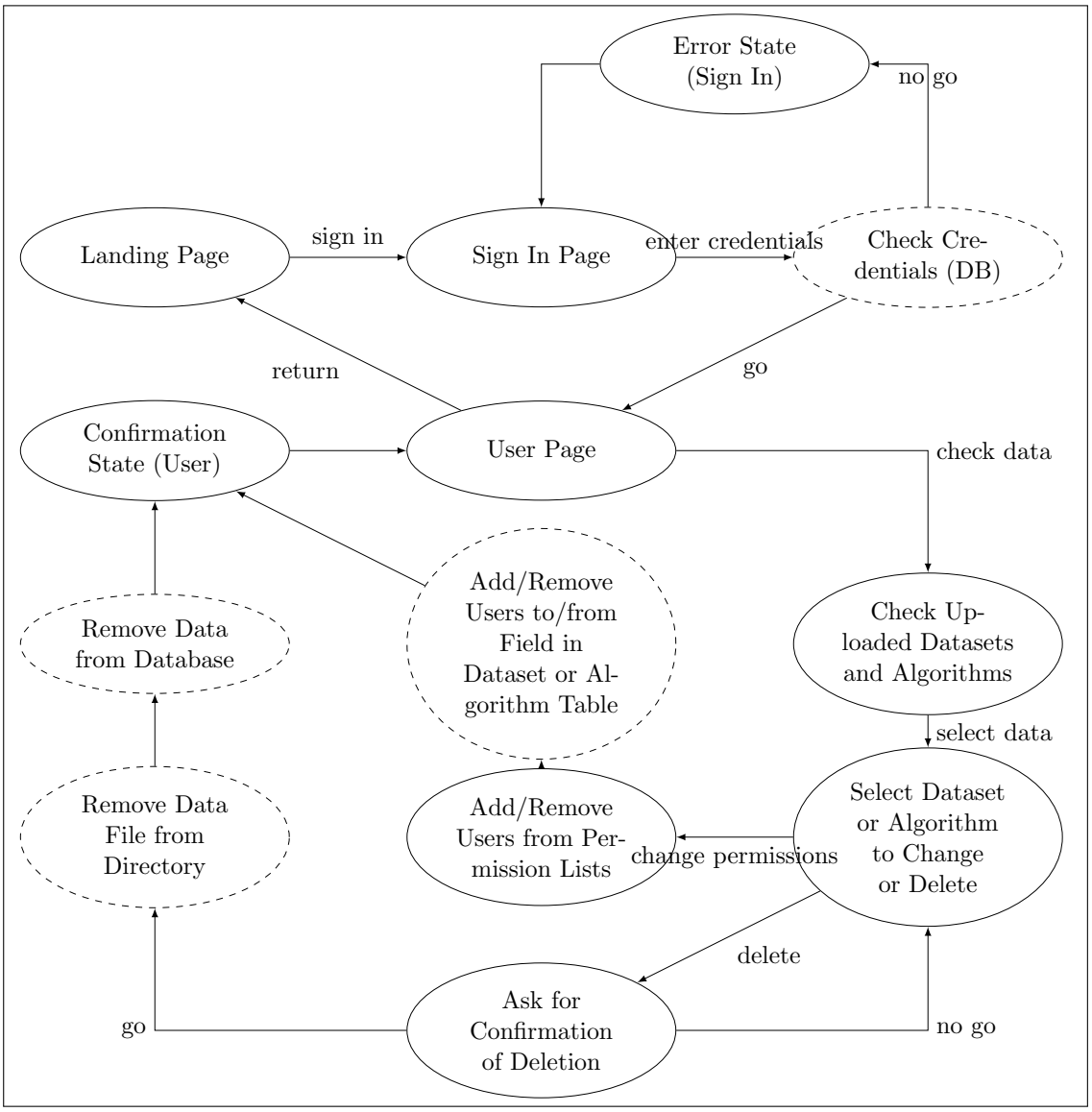

Figure C.ctrl.1: Finite State Machine for Controlling Uploaded Data actions

The user is able to change user permissions on who can view, modify, or download data. The user can also delete their uploaded data. This removes the file, as well as the information from the database, and hence the leaderboard.

The state 'Confirmation State (User)' returns users to the 'User Page' state.

9 API

An Application Programming Interface (API) describes the relationship between different components of a piece of software. For the web service, the APIs will describe the functions that run the service, including the states involved, the variables transferred, and the results returned.

In this section is a list of all user actions that might be undertaken in each state of the web service. Each component of the service has its own API attributed to it.

In the data flow part of the API, the data being passed is contained inside square brackets. The components that the data moves through are shown by arrows.

Following is a list of all user actions that might be undertaken while using the web service:

- sign in
- create account
- change account details
- upload dataset
- upload algorithm
- control uploaded data
- test data
- view leaderboard
- select result from list
- change criterion
- search
- view dataset/algorithm
- display dataset/algorithm
- download dataset/algorithm

Each of these actions will have a function or functions attributed to it. These functions form the API of the web service, and are sorted by the component from which they are called.

9.1 User Authentication

Actions calling the user authentication module:

- Sign in
- Create account
- Change account details

9.1.1 Sign In

The data flow for the 'sign in' action is described below:

User[id, credentials] → Webservice → User Authentication → Database
Database[y/n credentials met] → User Authentication → Webservice → User

The functions associated with this action are shown below.

def sign_in():

"'The function has no inputs. The function is initialised when the URL argument is 'sign in'.

The function moves the user to the 'Sign In' page. "'

def check_credentials(*username*, *password*):

"'The *username* and the *password* variables are strings parsed from the URL.

The function searches the User table of the database for a record with a username matching the *username* variable exactly. If no such record exists, the webservice sets the variable *error_message* to "Username not found", and the user is moved to the 'Error (Sign In)' state.

If the username strings match, the function then compares the 'password' field of the record with the user-entered *password* variable. If the strings don't match, then the variable *error_message* is to "Password does not match", and the user is moved to the 'Error (Sign In)' state.

If the password strings match exactly, then the variable *confirmation_message* is set to "Sign in successful", and the user is moved to the 'Confirmation (Land)' state. "'

def sign_in_error(*error_message*):

"'The string *error_message* contains a message to display to the user to inform the user which error occurred.

A page is displayed to the user which contains the string contained in the variable *error_message*. Upon confirmation by the user , the user is then moved to the 'Sign In' state. "'

def land_confirm(*confirmation_message*):

"'The string *confirmation_message* contains a message to display to the user for confirmation of function success.

A page is displayed to the user which contains the string contained in the variable *confirmation_message*. Upon confirmation by the user, the user is then moved to the 'Landing Page' state. "'

def return_to_start():

"'This function clears all variables and moved the user to the 'Start Page' state. "'

9.1.2 Create Account

The data flow for the 'create account' action is described below:

User[id, credentials, information] → Webservice → User Authentication → Database
Database[confirmation] → User Authentication → Webservice → User

The functions associated with this action are shown below.

def create_account():

"'The function has no inputs. The user initialises the function when they want to create an account to use the webservice.

The function moves the user to the 'Create Account' state. "'

def enter_credentials(*username*, *password*, *information*):

"'The variables *username* and *password* are strings entered by the user. The *information* variable contains the information for the account.

The function searches the Users table of the database for a record with a username matching the *username* variable exactly. If such a record is found, the variable *error_message* is set to "Username already exists. Please choose new username", and the user is moved to the 'Error (Sign In)' state.

If no matching username is found, then a new record is created in the User table of the database with the variables *username*, *password*, and *information*, and the server-assigned *user_id*.

The variable *confirmation_message* is then changed to "New account created", and the user is moved to the 'Confirmation (Sign In)' state. "'

def sign_in_confirm(*confirmation_message*):

"'The string *confirmation_message* contains a message to display to the user for confirmation of function success.

A page is displayed to the user which contains the string contained in the variable *confirmation_message*. Upon confirmation by the user, the user is then moved to the 'Sign In' state. "'

9.1.3 Change Account Details

The data flow for the 'change account details' action is described below:

User[information] → Webservice → User Authentication → Database
Database[confirmation] → User Authentication → Webservice → User

The functions associated with this action are shown below.

sign_in

check_credentials

def user_account_confirm(*confirmation_message*):

"' The string *confirmation_message* contains a message to display to the user for confirmation of function success.

A page is displayed to the user which contains the string contained in the variable *confirmation_message*. Upon confirmation by the user, the user is then moved to the 'User Account' state. "'

def change_account_info():

"'The function has no input. The user initialises the function when they want to take the 'change account info' action.

The function moves the user to the 'Change Info' state. "'

def confirm_account_changes(*username*, *password*, *information*):

"'The input variables default to the existing values in the database. The user changes only the data that they want to alter, and the other data is left as before.

A page is displayed to the user which contains the strings associated with the variables *username*, *password*, and *information*.

Upon confirmation by the user, the user id column for the record in the Users table is set to the variable *record_number*, and the user is the moved to the 'Add

Changes to DB' state. If the user does not want to change the shown data, then they are returned to the 'User Account' state. "'

def add_account_changes_to_database(*record_number*, *username*, *password*, *information*):

"'The variable *record_number* is an integer indicating the record in the Users table for the information being changed.

The function finds the record associated with the variable *record_number*, and replaces the fields for username, credentials, and information with the variables *username*, *password*, and *information*.

The variable *confirmation_message* is set to "User information changed", and the user is moved to the 'Confirmation (Land)' state. "'

def delete_account():

"'The function has no inputs. The user initialises the function when they want to take the 'delete account' action.

A page is displayed to the user which displays "Are you sure you want to delete this account? This action cannot be undone." Upon confirmation by the user, the variable *record_number* is set to the user id value, and the variable *db_table* is set to the string "users". The user is then moved to the 'Add Changes' state.

If the user does not want to complete the action, they are returned to the 'User Account' state. "'

def delete_database_record(*db_table*, *record_number*):

"'The variable *db_table* is a string indicating the table from which the record should be deleted. The variable *record_number* is an integer corresponding to the ID of the record to be deleted.

The function searches the table indicated in *db_table* for the record indicated by *record_number*. If the record is not found, then the string *error_message* is set to "Database record not found", and the user is moved to the 'Error (Land)' state. If the record is found, the the string *confirmation_message* is set to "Record deleted". If *db_table*=='users': the user is moved to the 'Start Page' state. Else: the user is sent to the 'Landing Page' state. "'

9.2 Other Components

See appendix D for the API for the other components of the web service.

10 Conclusions

10.1 Project Summary

This report outlines a detailed design for a competition web service for online attitude estimation algorithms. A complete finite state machine has been developed, as well as an API describing the various use cases for the service.

To reduce space in the report, some example diagrams are shown for the use flow diagrams, with the rest appearing in appendix B. This is also true for the API, with the User Authentication API appearing in the body of the report, while the API for the other components are in appendix D.

The implementation selection was ranked by the design metrics for only some of the more important decisions, with others being justified without a formal ranking.

The process of designing the web service was sequential. In order to develop the state machine, first the user actions and the service structure were be identified. To select the best implementation, design specifications had to be formulated from the customer requirements. Each step was iterative, and as each new section was developed, the previous sections were constantly being improved.

The design contained in this report will be helpful in implementing a working web service for attitude estimation algorithms.

10.2 Future Work

This project focused only on the design of the web service. The next step for the project would be taking the design outlined in this report and implementing a web service on a test server.

In terms of the design, possible additional features include a comment capability for the web service, which would allow users to make comments about datasets and algorithms. A discussion or forum section could also be implemented, for attitude estimation discussion that is not directly related to specific datasets or algorithms.

A reputation system for users could be added. This would allow users to gain reputation by uploading reliable datasets, or obtaining reliable results as a result of testing. Additionally, algorithm rankings can be given a reliability value, corresponding to the number of testers with similar results, or more reputable testers obtaining the result.

10.3 Project Reflection

The project summary form completed at the commencement of the project stated that a web service for attitude estimation algorithms would be designed and implemented. This was the aim of the project for the first semester, until time constraints necessitated a reduction in scope. Thus, it was decided that the project would focus on the design of the web service instead of both design and implementation. This allowed for a more complex and in-depth design, instead of a simple design and token implementation.

Several of the sections in the design process took a longer time than expected to complete. The use flow diagrams in particular required several weeks longer than predicted to complete to a level of satisfaction. However, the use flow diagrams were integral to the design, and the finite state machine could not be developed until these were finished.

Due to the length of time taken to complete the design, there was not enough time left in the project for a complete implementation.

The composite design and the state machine of the web service are among the most useful results of the report. With these designs, as well as the developed API, an implementation for the web service requires only the coding, without worrying about the design.

During the project, a test web service was developed in a virtual box and deployed locally. This was beneficial for learning and understanding exactly how flask and Apache work together, and to get them to actually perform as desired. Also required was learning how to manage a Debian operating system, and how to create a connection between a host machine and a virtual machine, so communication could occur, including accessing the deployed web service from the host machine.

Also learned during the project was a basic management of databases using MySQL, identifying and accurately portraying the user actions in the use flow diagrams, linking composite state diagrams and user actions to create a finite state machine, and hence developing a simple comprehensive API.

References

David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture. *BT Technology Journal*, (February), 2004.

CHRobotics. Understanding Quaternions. URL `http://www.chrobotics.com/library/understanding-quaternions`.

John L Crassidis, F. Landis Markley, and Yang Cheng. Survey of Nonlinear Attitude Estimation Methods. *Journal of Guidance, Control, and Dynamics*, 30(1):12–28, January 2007. ISSN 0731-5090. doi: 10.2514/1.22452. URL `http://arc.aiaa.org/doi/abs/10.2514/1.22452`.

DB-Engines. Db-engines ranking, 2014. URL `http://db-engines.com/en/ranking`.

Flask. Flask web development, 2014. URL `http://flask.pocoo.org/`.

Jinja. Jinja, 2014. URL `http://jinja.pocoo.org/`.

E J Lefferts, F L Markley, and M D Shuster. Kalman Filtering for Spacecraft Attitude Estimation. *Journal of Guidance, Control, and Dynamics*, 5(5):417–429, September 1982. ISSN 0731-5090. doi: 10.2514/3.56190. URL `http://dx.doi.org/10.2514/3.56190`.

F. Landis Markley and John L Crassidis. Unscented filtering for spacecraft attitude estimation. *Journal of Guidance, Control, and Dynamics*, 26(4):536–542, 2003.

MySQL. Mysql, 2014a. URL `https://www.mysql.com/`.

MySQL. Chapter 11: Data types, 2014b. URL `https://dev.mysql.com/doc/refman/5.0/en/data-types.html`.

Netcraft. June 2013 web server survey, 2013. URL `http://news.netcraft.com/archives/2013/06/06/june-2013-web-server-survey-3.html`.

The Apache Software Foundation. The apache software foundation, 2014. URL `https://www.apache.org/`.

Greg Welch and Gary Bishop. An introduction to the Kalman filter. pages 1–16, 1995. URL `http://clubs.ens-cachan.fr/krobot/old/data/positionnement/kalman.pdf`.

Roni Yadlin. Attitude Determination and Bias Estimation Using Kalman Filtering.

A Customer Requirements

A.1 Identifying Customer Requirements

To identify requirements, each customer was analysed individually. The requirements were identified in discussion with the project supervisor, Jochen.

A.1.1 Suppliers of Algorithms

The Suppliers of Algorithms requirements identified were:

- Able to upload algorithms
- Unique user accounts
- Other users can comment on algorithms
- The service must have no cost

A.1.2 Suppliers of Datasets

The Suppliers of Datasets requirements identified were:

- Able to upload large datasets
- Secure upload/storage
- Can control permissions on datasets
- Unique user accounts
- Other users can comment on datasets
- The service must have no cost

A.1.3 Users of Service

The Users of Service requirements identified were:

- User accounts
- Able to download datasets and/or algorithms
- The service must have no cost
- The service should be fast

A.1.4 Jochen/Server

Jochen's identified requirements were:

- Service must use Debian, flask, Apache, and mysql
- Documentation - for assessment

The Server requirements identified were:

- The service must be secure from unwanted access/control
- Various computational load, cost, memory, etc. requirements

A.1.5 Callum/Developer

Callum's identified requirements were:

- Limitations from lack of knowledge
- Compliance with the project boundaries
- Scope completable on time (hard deadline)

The Developer requirements identified were:

- Documentation - for own memory, understanding code

A.1.6 Development Community

The Development Community requirements identified were:

- Documentation - for similar projects in the future

A.1.7 Attitude Estimation Community

The Attitude Estimation Community requirements identified were:

- Use the correct/standard terminology for ease of comprehension

A.2 Sorting Customer Requirements

In this section we organise the identified customer requirements into groups based on the type of requirement.

A.2.1 Utility

- Able to upload algorithms
- Able to upload large datasets
- Able to download datasets and/or algorithms

A.2.2 Extra features

- Unique user accounts
- Other users can comment on algorithms/datasets

A.2.3 Security

- Unique user accounts
- Secure upload/storage of data
- Suppliers can control permissions on datasets
- The service must be secure from unwanted access/control

A.2.4 Implementation

- Service must use Debian, flask, Apache, and mysql
- Various computational load, cost, memory, etc. requirements
- Limitations from lack of developer knowledge
- Compliance with the project boundaries
- Scope completable on time (hard deadline)
- The service must have no cost
- The service should be fast

A.2.5 Documentation

- Documentation for assessment
- Documentation for developer's use
- Documentation for similar future projects

- Standard terminology used for ease of comprehension

B Use Flow Diagrams

B.1 Use Flow for Suppliers

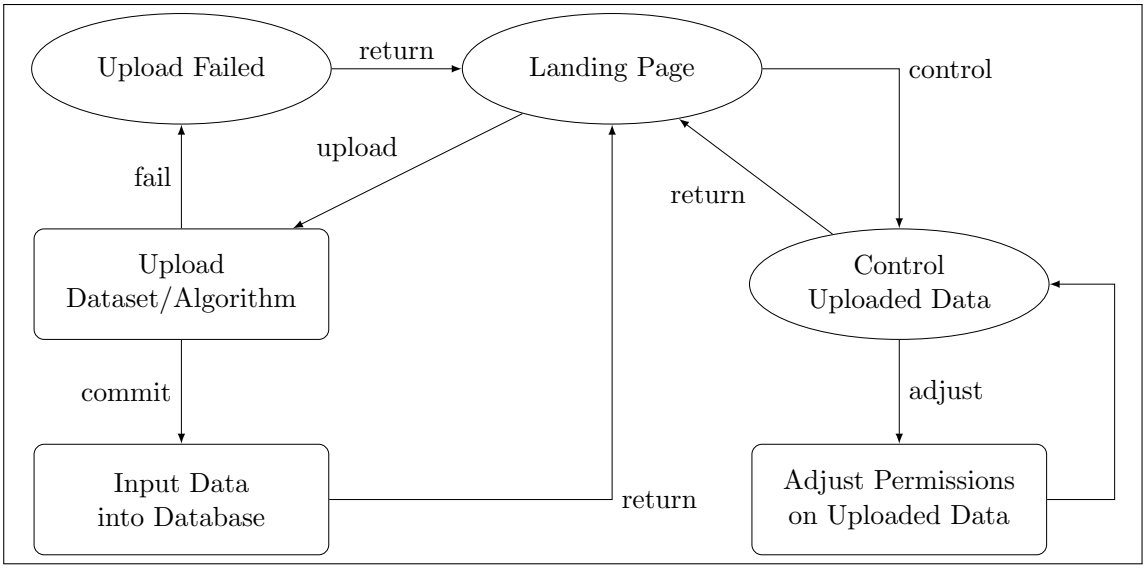

Figure A.supp.1: Use Flow Diagram showing use flow for Suppliers

B.2 Use Flow for Testers

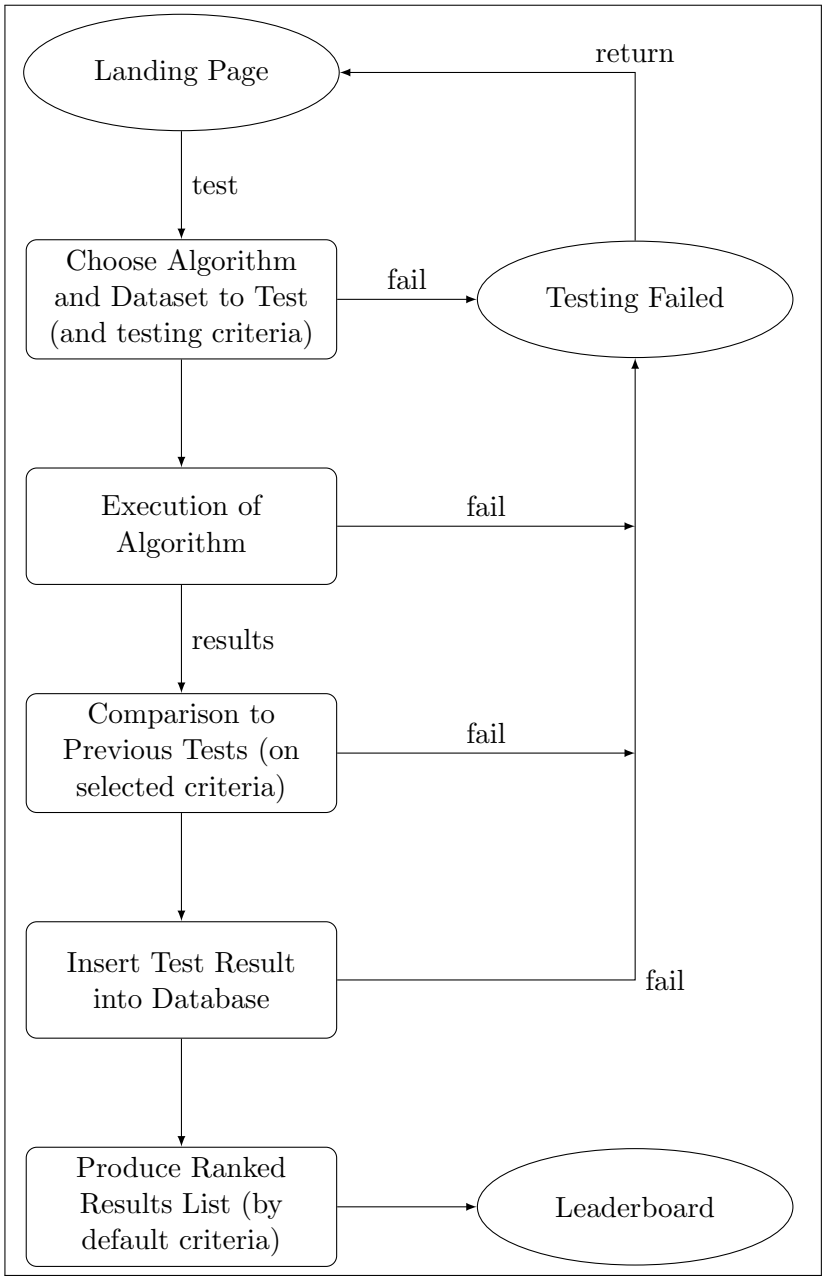

Figure A.test.0: Use Flow Diagram showing use flow for Testers

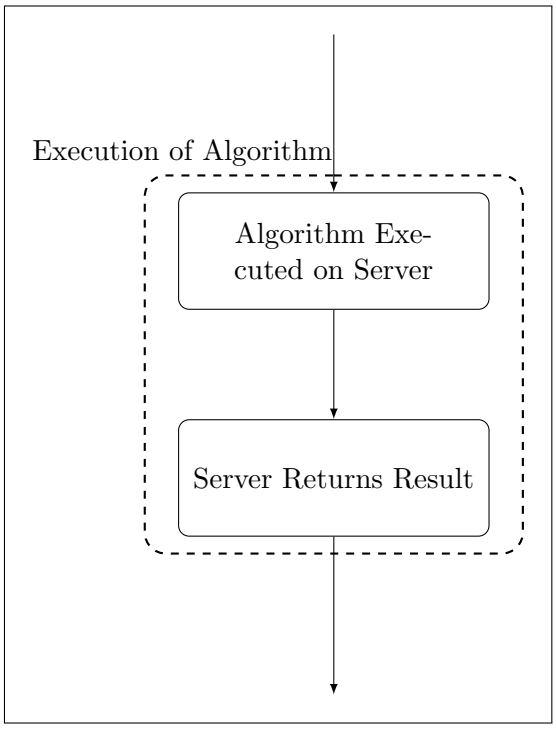

Figure A.test.1: Use Flow Diagram showing use flow for Testers (Local)

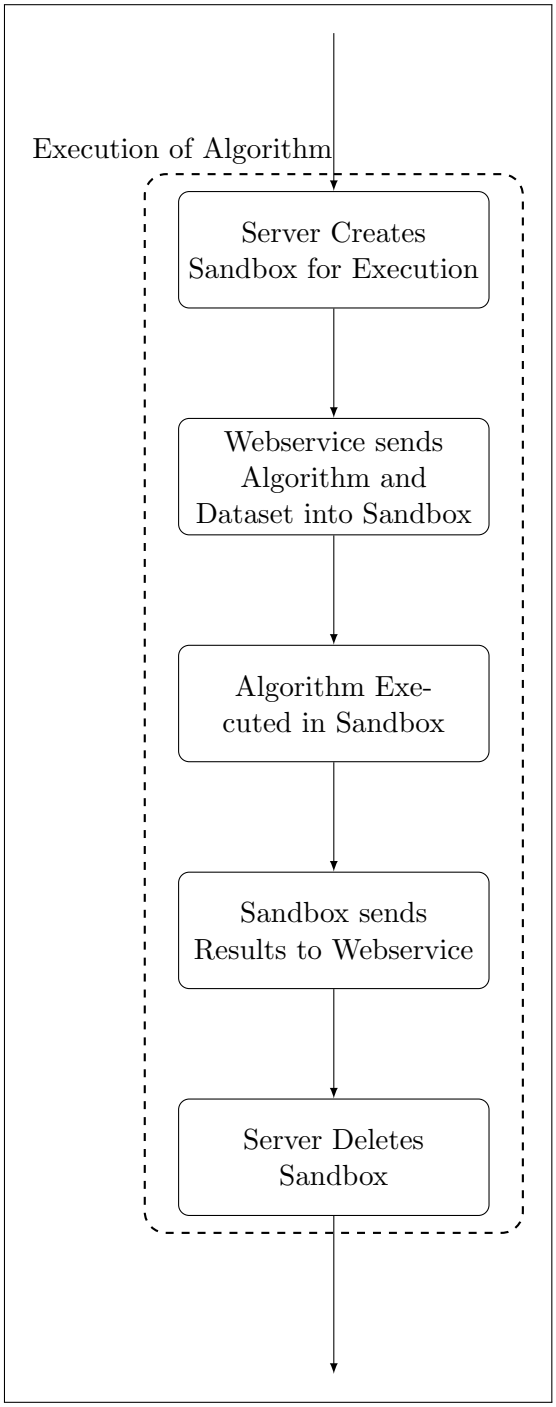

Figure A.test.2: Use Flow Diagram showing use flow for Testers (Sandboxed)

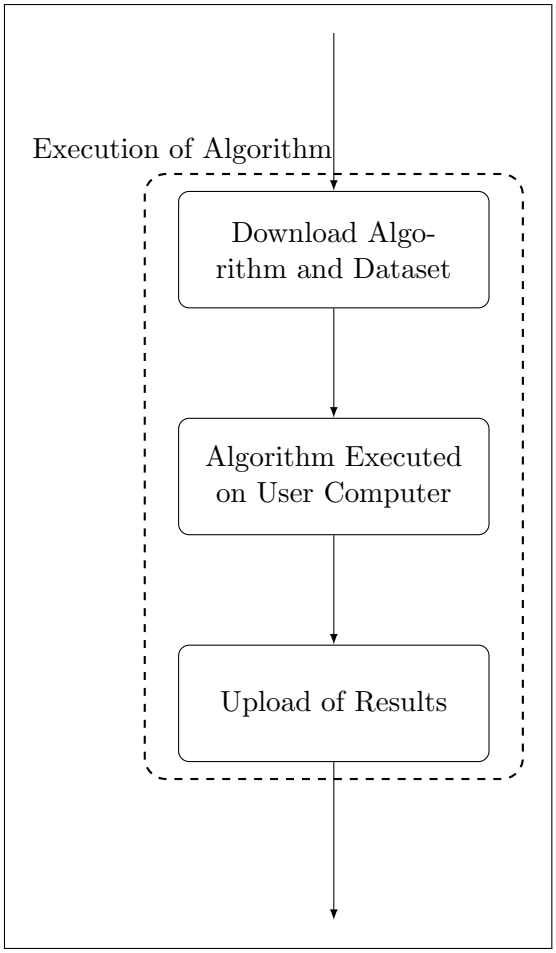

Figure A.test.3: Use Flow Diagram showing use flow for Testers (Client-side)

B.3 Use Flow for the Attitude Estimation Community

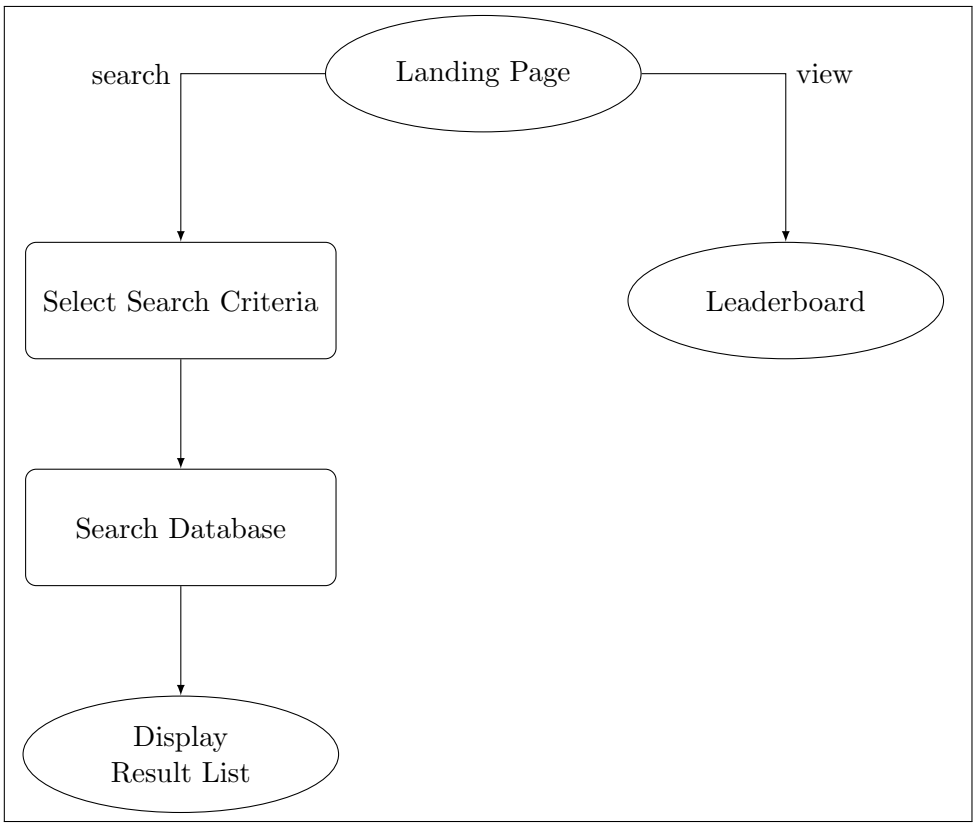

Figure A.comm.1: Use Flow Diagram showing use flow for Community

B.4 Use Flow from Results State

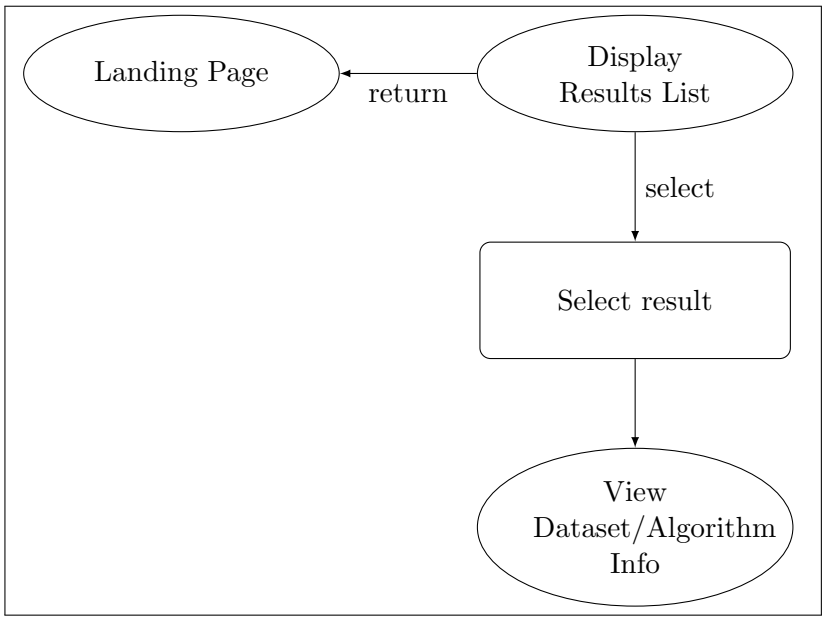

Figure A.resu.1: Use Flow Diagram showing use flow from Results state

B.5 Use Flow from Leaderboard State

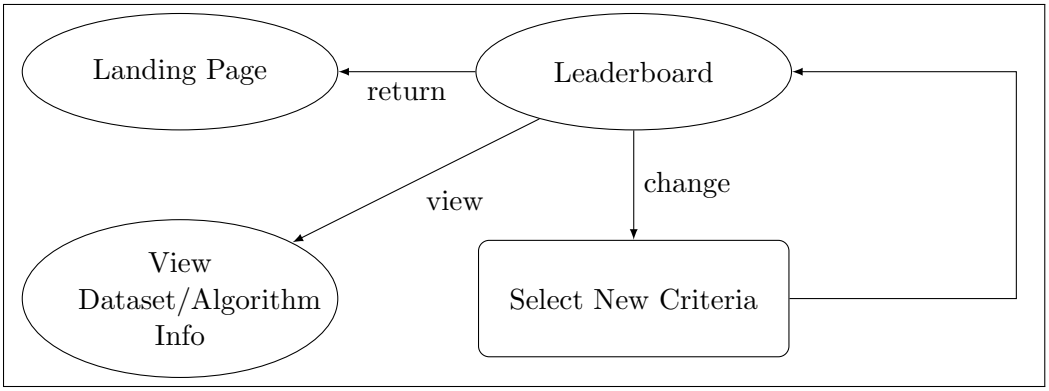

Figure A.lead.1: Use Flow Diagram showing use flow from Leaderboard state

B.6 Use Flow from View State

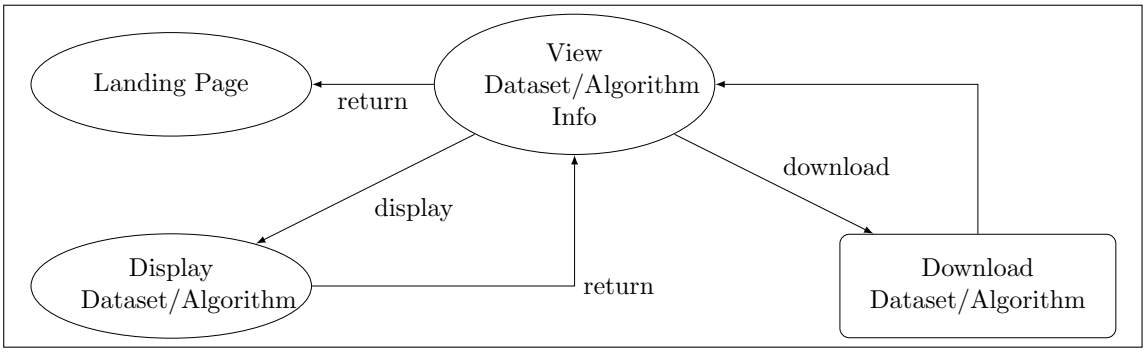

Figure A.view.1: Use Flow Diagram showing use flow from View Info state

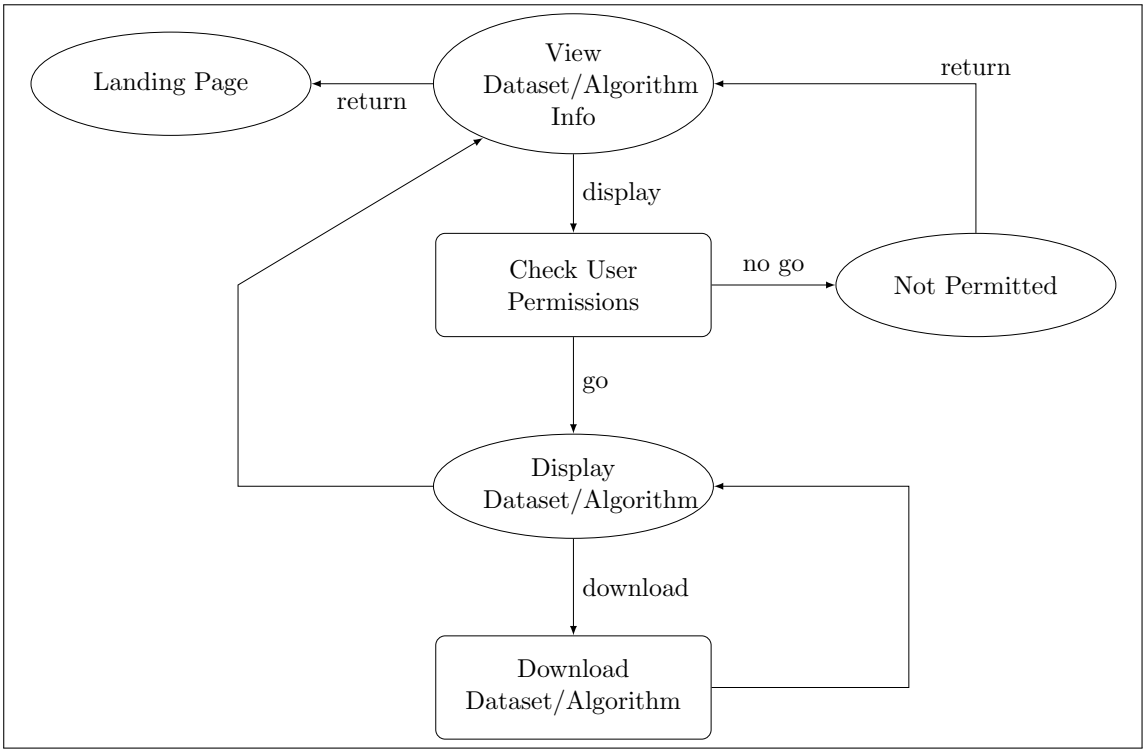

Figure A.view.2: Use Flow Diagram showing use flow from View Info state

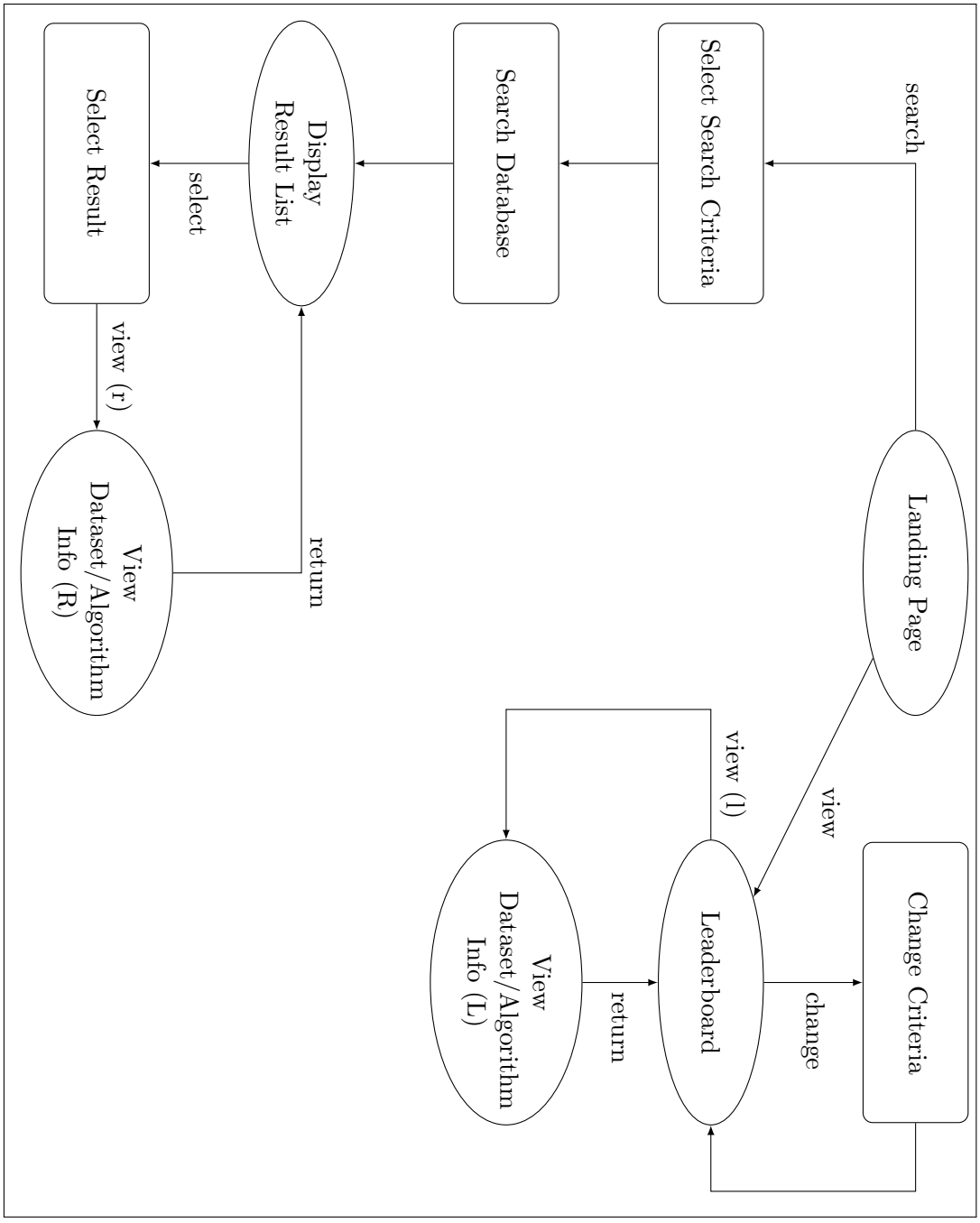

Figure A.alt.2: Use Flow Diagram showing implementation for multiple View Info states

B.7 Composite Structure Diagrams - Implementation 2

The data flow is outlined below. The data contained inside square brackets is the data that is passed, and the arrows show the path that the data takes.

- User Sign-in

  User[id, credentials] → User Authentication

  User Authentication[y/n credentials met] → Webservice → User

- Uploading Datasets and Algorithms

  User[data] → User Authentication

  User Authentication[y/n authorised] → Filter

  User Authentication[data] → Webservice → Filter

  Filter[data] → Database

  Database[confirmation] → Filter → Webservice → User

- Testing Datasets and Algorithms

  A.test.1, A.test.2: User[request] → User Authentication

  User Authentication[y/n authorised] → Filter

  User Authentication[request] → Webservice → Filter

  Filter[request] → Database

  Database[dataset, algorithm, previous results] → Filter → Webservice

  Webservice[result, comparison to previous] → Filter → Database

  Database[ranked list] → Filter → Leaderboard → Webservice → User

  A.test.3: User[request] → User Authentication

  User Authentication[y/n authorised] → Filter

  User Authentication[request] → Webservice → Filter

  Filter[request] → Database

  Database[dataset, algorithm, previous results] → Filter → Webservice

  Webservice[dataset, algorithm] → User

  User[result] → Webservice

  Webservice[result, comparison to previous] → Filter → Database

  Database[ranked list] → Filter → Leaderboard → User

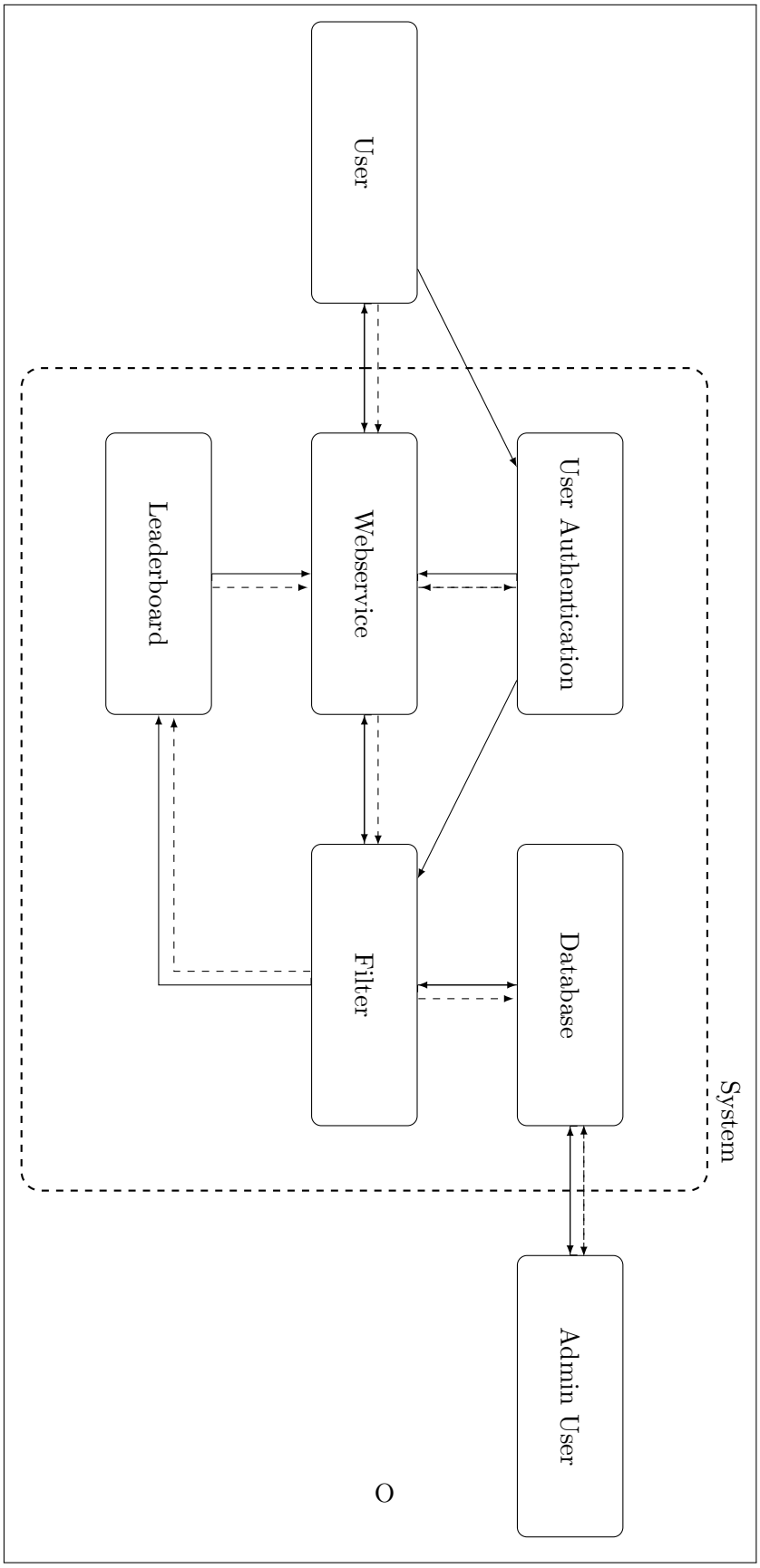

Figure B.2: Composite Structure Diagram Implementation 2

- View Leaderboard

  User[request] → User Authentication

  User Authentication[y/n authorised] → Filter

  User Authentication[request] → Webservice → Filter

  Filter[request] → Database

  Database[ranked list] → Filter → Leaderboard → Webservice → User

- View Dataset or Algorithm

  User[request] → User Authentication

  User Authentication[y/n authorised] → Filter

  User Authentication[request] → Webservice → Filter

  Filter[request] → Database

  Database[dataset, algorithm] → Filter → Webservice → User

C Ranking of Implementations for Tester by Metrics

The metrics relevant to the use flow for testers listed below:

Table C.1: Relevant metrics for ranking

Met No.	Need No.	Metric	Score	Units
13	8	Suppliers can approve access to data	4	Binary
14	9	Location for execution of code (algorithms)	4	List
20	1-3	Average time to complete requests	5	s
21	1-3	Worst case time to complete requests	5	s
22	13	Project deadline is met	4	Binary

are metrics 13, 13, 20, 21, and 22:

The possible implementations were given a score according to how well the metric was met:

5 – metric is met absolutely/exceeded
3 – metric is mostly met
1 – metric is partially met
0 – metric is not met

The score was then multiplied by the weighting, and the total for each implementation summed.

Table C.2: Ranking of Implementations for Tester by Metrics

Metric	Score	Implementation 1		Implementation 2		Implementation 3	
		Score	Weighted Score	Score	Weighted Score	Score	Weighted Score
14	4	3	12	5	20	3	12
15	4	0	0	5	20	3	12
21	5	5	25	3	15	3	15
22	5	5	25	3	15	1	5
23	4	3	12	1	4	5	20
Total			74		74		64

D API

D.1 Webservice

Actions calling the webservice:

- Upload dataset
- Upload algorithm
- Control uploaded data
- Test data

D.1.1 Upload Dataset

The data flow for the 'upload data' action is described below:

User[data] → Webservice → Database
Database[confirmation] → Webservice → User

The functions associated with this action are shown below.

def upload_dataset():

"'The function has no inputs. The user initialises the function when they want to take the 'upload dataset' action.

The variable *file_type* is set to "dataset".

The user is moved to the 'Choose File' state. "'

def select_file():

"'The function has no inputs.

A file browsing window is opened on the user's machine. The user browses the file system and selects the file that they wish to upload. The pathway to the file is set to the variable *client_pathway_to_file*. The user enters a string for the variable *dataset_name*.

The user is moved to the 'Upload File' state. "'

def upload_file(*client_pathway_to_file*, *file_type*):

"'User chooses the dataset or algorithm to upload, and finds the pathway to the file, which is variable *client_pathway_to_file*. User must also state the type of the file as a string, which is the *file_type* variable. Acceptable inputs for *file_type* are 'dataset' and 'algorithm'.

The function first reads the string *file_type*. Depending on this parameter, the function will either return an error, or move the file to the relevant upload directory on the server.

The function returns the variable *temp_pathway_to_file*, which will direct to either the dataset upload directory or the algorithm upload directory, depending on the parameter *file_type*. "'

def parse_dataset(*temp_pathway_to_file*):

"'The variable *temp_pathway_to_file* gives the location of the dataset on the server.

The dataset is run through the dataset parser. The variables *dataset_description*, *time_series_length*, *num_time_series*, *time_series_semantics*, and *time_series_types* are retrieved from the parser.

If any of these variables are missing from the dataset, the variable *error_message* is set to "Error: <missing_variable/s> not found in dataset", and the user is moved to the 'Error (Land)' state. "'

def move_file_to_directory(*file_type*, *temp_pathway_to_file*):

"'The string *file_type* describes the type of the file. Acceptable values are "dataset" and "algorithm". The string *temp_pathway_to_file* is the pathway to the file in the upload directory on the server.

The function reads the variable *file_type*, and moves the file from its current location, recorded in *temp_pathway_to_file*, to a location in either the dataset directory or the algorithm directory.

The new location of the file is recorded in the variable *pathway_to_file*. "'

def enter_dataset_into_database(*dataset_name*, *dataset_description*, *time_series_length*, *num_time_series*, *time_series_semantics*, *time_series_types*, *pathway_to_file*):

"'The inputs are all variables which become fields in the database record. Along with the input variables, the user record is recorded as *user*.

A new record is created in the Dataset table of the database, using the information from the inputs. For the user permissions fields, the default is for the supplier only. The string *confirmation_message* is set to "Dataset entered into database successfully".

The user is moved to the 'Confirmation (Land)' state. "'

D.1.2 Upload Algorithm

The data flow for the 'upload algorithm' action is described below:

User[data] → Webservice → Database
Database[confirmation] → Webservice → User

The functions associated with this action are shown below.

def upload_algorithm():

"'The function has no inputs. The user initialises the function when they want to take the 'upload algorithm' action.

The variable *file_type* is set to "algorithm".

The user is moved to the 'Choose File' state. "'

def parse_algorithm(*temp_pathway_to_file*):

"'The variable *temp_pathway_to_file* gives the location of the algorithm in the upload directory of the server.

The algorithm is run through the algorithm parser. The variables *required_semantics*, *required_shapes*, *algorithm_description*, *algorithm_purpose*, *algorithm_methods*, and *algorithm_imports* are retrieved from the parser.

"'

def function name(*inputs*):

"'define the inputs

describe the function process

describe the outputs "'

D.1.3 Control Uploaded Data

The data flow for the 'control uploaded data' action is described below:

User[permitted users list] → Webservice → User Authentication → Database

D.1.4 Test Data

The data flow for the 'test data' action is described below:

User[request] → Webservice → Database
Database[dataset, algorithm, previous results] → Webservice
Webservice[result, comparison to previous] → Database
Database[ranked list] → Leaderboard → Webservice → User

D.2 Leaderboard

Actions calling the Leaderboard module:

- View leaderboard
- Select result from list
- Change sort criteria

D.2.1 View Leaderboard

The data flow for the 'view leaderboard' action is described below:

User[request] → Webservice → Database
Database[ranked list] → Leaderboard → Webservice → User

The functions associated with this action are shown below.

def function name(*inputs*):

"'define the inputs

describe the function process

describe the outputs "'

def search_for_matching_algorithms(*required_semantic_types*, *required_syntactic_types*):

"' The variable *required_semantic_types* is a list of semantic types that are required for the algorithm. The variable *required_syntactic_types* is a list of syntactic types that are required.

These lists are ordered such that each element in the *required_semantic_types* list has the syntax given by the corresponding element in the *required_syntactic_types*. These can be empty lists.

The database is searched for algorithms that have entries in the *required_semantics* and *required_syntax* fields matching all of those in the *required_semantic_types* and *required_syntactic_types* lists.

The variable *matched_algorithms* is defined as a list of the records in the Algorithm table of the database that match the *required_semantic_types* and *required_syntactic_types* lists. "'

def sort_results(*matched_algorithms*, *sort_criterion*):

"' The variable *matched_algorithms* is a list of records the the Algorithm table of the database. The variable *sort_criterion* is a string corresponding to the desired ranking scheme for the algorithms. This should match either "execution_time" or "accuracy".

The function sorts the records designated by the list *matched_algorithms* by the field determined by the variable *sort_criterion*.

The ranked list is set to the variable *ranked_matched_algorithms*. "'