MANCHESTER 1824
The University of Manchester

The Application and Integration of Data Analytics Pipelines in Automated Nano-Material Research

A dissertation submitted to The University of Manchester for the degree of

Master of Physics

in the Faculty of Science and Engineering

Year of submission

2023

Student ID

10294857

School of Natural Sciences

Tomasz Neska

Contents

Contents **2**

Abstract **3**

Acknowledgements **4**

1 Introduction **5**
1.1 Motivation 5
1.2 Aims and Objectives 6

2 Introduction to the Existing Architecture **7**
2.1 Introduction 7
2.2 Data Structure 8
2.3 The Application Programming Interface (API) 9
2.4 The User Interface 12

3 Methodology **13**
3.1 Introduction 13
3.2 Computational Time Complexity 13
3.3 The Development Path 14

4 Critical Evaluation **22**
4.1 Goal Success Evaluation 22
4.2 Other Technology 23

5 Future Work **23**

References **25**

Appendices **31**

A Libraries and Technical Details **31**

B Testing and Continuous Development Cycle **31**

C Glossary **31**

D Jupyter Notebook **33**

E Full calculation **34**

Abstract

This report showcases the current architecture and the developments made on the data analytics pipeline designed for analysis of experimental data in nano-material research. The architecture developed in the previous semester was transformed into a functional pipeline with the ability to manage data efficiently, securely and handle images. Additionally, improvements in the user interface offered a more streamlines user experience and hence more accessibility for users with less coding experience. The pipeline includes an Application Programming Interface and a user interface for data manipulation and authentication, utilizing the MongoDB database to store data in a tree data structure. The user interface enables researchers to easily access and manipulate data, and provides security measures such as asymmetric key encryption and JSON Web Tokens generation for authentication. The pipeline was successfully demonstrated on a virtual server and is ready to be utilised as a tool in semiconductor material research. While there are opportunities for improvement, such as the implementation of a visual interface this project presents a suitable architecture for use in high-throughput experiments.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr Patrick Parkinson for the continuous support in my work and research. For his patience, motivation, enthusiasm and immense knowledge.

Besides my advisor, I would like to thank the rest of the research group at the OMS Lab: Dr Stephen Church, Nikesh Patel, Nawal Al-Amairi, Ruqaiya Al-Abri and Dr Hoyeon Choi.

Additionally I would like to thank my friends for the support both technical and emotional: Tristan Broadhead, Euan Lacy and Zobia Hussain.

1 Introduction

1.1 Motivation

1.1.1 Background

In 1989, Tim Berners-Lee wrote the first proposal for the World Wide Web [1], setting off a chain of innovation that allowed for a world defined by information sharing. Fast forward to 2023, and the amount of data surrounding every human increased continuously. Every single aspect of the internet collects personal data to be used as a part of the service or for advertising [2]. This only highlights the importance of data in our lives. Those who can harness this data in its totality and differentiate the signal from the noise [3] have the potential to lead innovation, produce financial wealth and make the world a better, safer [4] and more productive place [5].

Data pipelines serve as a tool for moving data. One side of this pipeline is responsible for data acquisition. This allows for sectioning and filtering while the pipeline itself provides the architecture for moving the information to databases or *data lakes* for processing. Once processed, the data is utilized to derive insights into whatever phenomenon it describes. For example, by using the pipeline along with an automatically updating dashboard. The goal of this project is to produce a tool to handle a scalable amount of data and utilize it fully to calculate the necessary statistical quantities easily and automatically. These methodologies have a range of application from experimental particle physics [6] to pharmaceutical development [7].

One end of the pipeline can be pointed towards the laboratory equipment under minimal supervision to provide the necessary data to the architecture. The other end is pointed towards the researcher who is making decisions regarding in what way the endless landscape of variables is to be changed to probe the unknown.

The best type of experiment in which this approach shines is high-throughput [8], which refers to an experiment that provides large quantities of data at high volume and velocity. Those features of experimental data lend themselves to the analytics pipeline which allows for storage and easy retrieval once analysis is to be conducted.

1.1.2 Nano-materials and Data Pipelines

Consider an experiment in which nano-materials are grown using a set of defined parameters [9]–[11]. A good example is a process of growing GaAs using molecular beam epitaxy (MBE) [12] [13]. This process involves epitaxial growth using one or more molecular beams within a crystalline structure under vacuum conditions. In this process variables like temperature, intensity of the

beams, pulse rate, material purity, substrate width, inert gas composition and many more need to be considered to obtain the desired structure as a result. This has been traditionally done by the development of best practices and hours of experimentation [14]–[16]. However, if a new material needs to be grown or a completely unknown results are expected it is worthwhile to understand the relationships between those variables. This would lead to an understanding which ones lead to the desired product and which ones do not.

Following this growth an experiment is conducted and variables like the optical spectrum and lasing threshold are measured [17]. If the number of parameters for growing such materials is larger than 2-3, the ability to maximize a lasing threshold by manipulating those variables is very difficult if each dataset has to be processed manually. With the use of a data pipeline and analysis code, the growth variables can be encoded as part of *meta-data*, allowing the connection of the experimental equipment with a database. Then a single dashboard has to be written and every time the database containing our experimental data is updated, it just takes a single button press to refresh our findings given new data. This allows the researcher to handle more data and hence get a better picture of how to maximise the desired variable.

This also allows the experimenter to quickly abandon the variables that are not correlated. One of the key benefits is that data is stored in a standardized format that can be easily understood by anyone who wishes to repeat the analysis. This approach makes good practices in data management easier to implement and enables researchers to work efficiently while ensuring that data is stored in a readable way [18] [19].

The further advantage of the system is the ability to integrate data from multiple sources and locations, with data entering the pipeline standardized for consistency and up-to-date analysis, essential for scientific reproducibility [20]. The remaining advantages of this approach are scalability and collaboration [21], which allows for sharing scientific data, leading to new approaches and innovation in describing existing phenomena [5]. In conclusion this technology has application across many stages of nano-material research and the benefits of using the data analytics pipeline are many and the only barrier standing in the way of utilising this technology is making complex code accessible. In this report I will outline the technology developed and the changes made to it to fully allow for integration of this specific pipeline into the research process.

1.2 Aims and Objectives

The goals of the project are the following:

1. Improve the efficiency of the functions used in sending and fetching the data. Demonstrate a quantitative improvement by improved time complexity.
2. Set up the database with existing experimental data and use it to provide meaningful statistical analysis.

3. Improve the existing authentication to ensure the public API calls are secure.
4. Produce functions which allow for scalability by fragmenting datasets. Scalability in this context means the ability of the code to scale with the size of the database and for it to not require a hard limit on the size of data it operates on.
5. Ensure the ability of handling images within the pipeline. This includes fetching, saving and retrieving images.

Expansion of functionality to provide image handling is important to allow for storage of spectral images in the context of nano-materials. Additionally, the scalability is important as a consequence of large image sizes that spectra can become. This means that to handle image efficiently it needs to be done by a scalable architecture.

The improvement in efficiency is presented and explained in more detail further in the report. The goal detailing meaningful statistical analysis can be explained by the usage of this architecture in the process of acquiring the desired calculation results from experimental data that was hosted and delivered by this pipeline. Given this architecture is to be used as a tool for research it is essential that it allows for reliable storage and fetching of data. It is important to mention that goal number 4 cannot be accomplished fully due to technical limitations but it can be expanded to meet those constrictions. For example, any database has a limited size. However, a good database handling tool should be designed as if it would operate on a database of infinite size. Scalability is a design principle and should be used to implement code which fits the technical requirements of the platform it operates on while remaining flexible enough to accommodate a different platform if necessary. This is expanded upon in Section 3.3.4.

This report is going to describe the exiting architecture in details from the data structures used, the included functionality and finally the authentication system. This will be followed by the detailed changes that were made to this architecture this semester and the reason for their inclusion. An ongoing comparison is going to be made to existing technologies and this will be end with the critical evaluation of the project and the description of how each goal was met. Finally the report will summarise the future work that could be carried on this project and the technical improvements that were omitted this semester but should be mentioned.

2 Introduction to the Existing Architecture

2.1 Introduction

The architecture contains the functionality that can be used to store, retrieve and query data. It allows for data handling. This means that data can be inserted into a database and then retrieved

as long as the data inserted follows the formatting required. The functionality then extends to data groups. This allows for linking specified datasets and their paths into a group. These groups can then be retrieved or shared to a different user. To access data a user needs to authenticate that the correct interface is being used and then they are shared the access to only the data in which they are listed as an author. The functionality then extends into being able to insert new data and then assign other users to their data and hence sharing it. These operations can be performed on the level of each *Dataset* but also over the entire *Experiments* and *Projects*. The final functionality of the API comes from the fetching of the data. Once a user is authenticated the data can be fetched and then utilised in analysis.

The architecture is separated into two parts. The first is the the application programming interface (API) which accesses the MongoDB database [22] and allows for storage of data. The second part is the user interface. This is the part which allows for the interaction between the user and the architecture. It contains functions which in turn call the API. The function of the interface is to simplify the functionality of the API and also provide for an intuitive, user-friendly way of allowing for data manipulation.

The best way to understand the function that this architecture performs is to use a data journey. This can be seen in Figure 1. The data starts at the acquisition stage. This data can be collected either by the researcher's previous experiments or it can be automated to be extracted directly from the equipment. This is then converted using a language specific driver into a JavaScript Object Notation (JSON) file. This architecture was designed with files of sizes ≈ 15 MB in mind. This driver converts the data into a desired data structure which can then be inserted through the user interface directly into the API. Given the pre-processing by the driver is done and the data type verification has been done client-side this means that the data can be inserted directly into the database after the user passes the authentication required. Then on a later date the data can be retrieved by the user interface and then given to an interactive dashboard which can update asynchronously to provide insight into a data for a given experiment.

2.2 Data Structure

The smallest building block of any database is a form of a structure. In this case a tree is used along with the *dynamic schema* of a *MongoDB*. The data structure of the system is a hierarchical tree, with the *Project* serving as the root, and the *Experiment* and *Dataset* nodes branching off from it. The *Dataset* represents the lowest level of the tree and is the primary unit of data handling in the system. The MongoDB database implementation represents each *Project* as a separate *Database* on the server, while each *Experiment* is treated as a *collection* within that database. The *Dataset* node is represented as a dictionary object, which is directly stored on the server in a specified location. The system allows for multiple *Experiments* to be stored within a single *Project*, which are grouped together using the *groups* variable. Within each *Experiment*, multiple *Datasets* can be

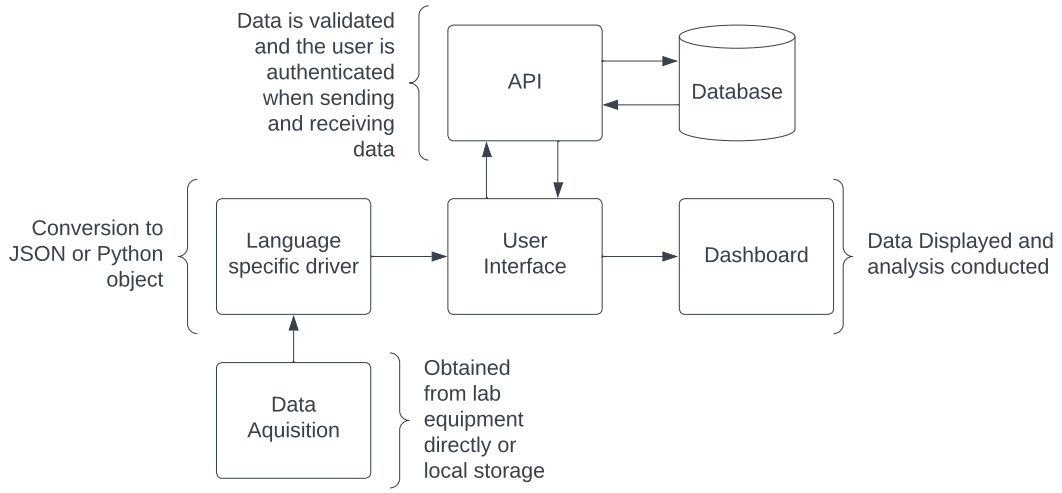

Fig. 1. The description of the journey that data takes within the architecture.

stored, and these are managed using the *datasets* variable. The tree structure of the system can be represented using *Python* classes, with the entire tree being represented as a nested object. The *Pydantic library* [23] is used to create these *objects*, which provides support for variable validation to prevent incorrect data from being assigned. This ensures that the database only contains data that matches the data type that is specified during the initialisation of the respective *objects*. If the data structure is converted into a JSON file using an in-built function, it is represented as a nested dictionary. This structure provides an efficient and flexible means of organizing and managing data within the system. A JSON dictionary is an array of key-value pairs which allows for storage of any data type but in the JSON form the whole list is represented as a single *string*.

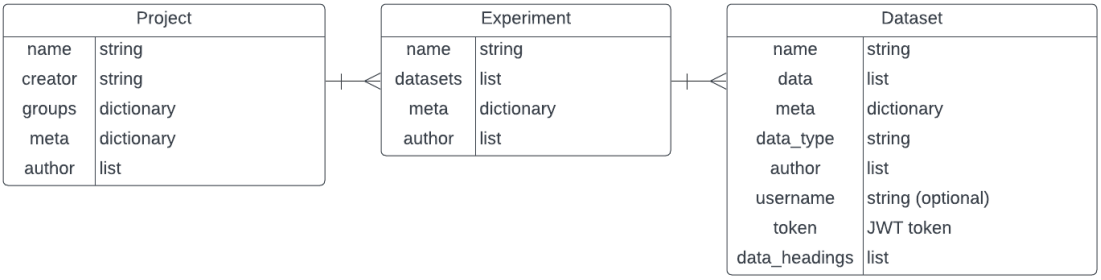

Fig. 2. The data structure utilised by the interface for handling data and also the theoretical model by which the database stores data.

2.3 The Application Programming Interface (API)

2.3.1 The Functionality

The API is a series of functions wrapped using the *FastAPI library* [24]. These functions perform HTTP calls with a server. This can be one or more calls as required to perform a specific action. For

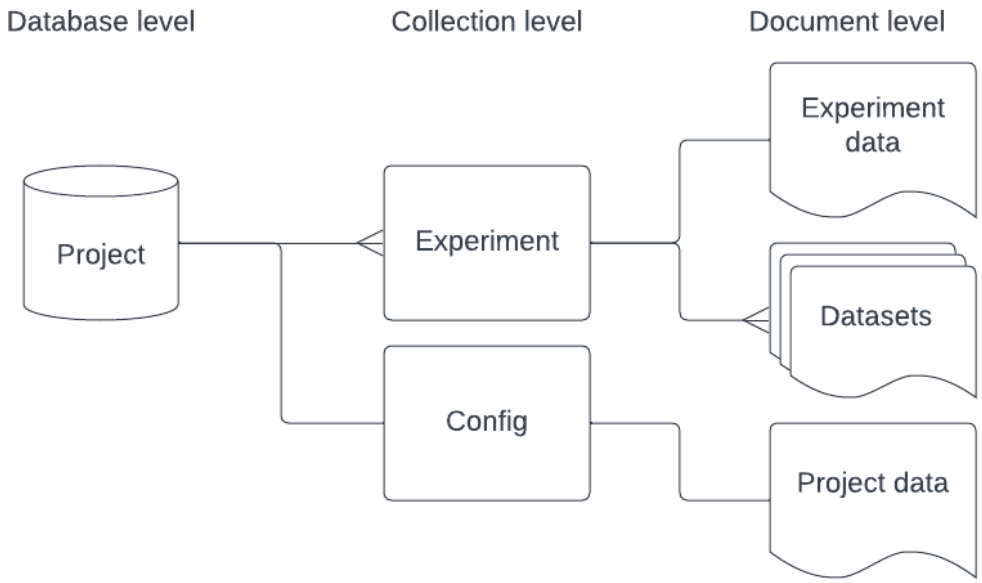

Fig. 3. The schematic view of how MongoDB views data that is within the database in the context of the used data structure.

example, to insert a dataset the user needs to authenticate and then initialise the *Experiment* or *Project* if those do not exist. Then the dataset is checked against the list of datasets that already exist in the specified path and finally the dataset is inserted if it does not already exist.

The user interface then utilises the *request library* [25] to translate those functions into a call which either gets or posts data. This mirrors the HTTP calls of GET and POST. The functionality of the API can be summarised into three categories: data manipulation, initialisation and user management. The data manipulation encompasses the functions which send and retrieve the data from the interface into the database and vice versa. This can be done by the use of the *insert_dataset* function and the fetching of data can be done by the use of *return_dataset*. These interface functions perform calls named *insert_single_dataset* and *return_dataset* respectively.

Secondly, the API allows the *Experiment* and *Project* nodes to be initialised. As seen in Figure 3 the data for the nodes needs to be stored within a *Dataset* on the database. This means that calls need to be made to initialise the Project and Experiment nodes to allows for data storage. This is performed by the API called named: *update_project_data* for the *Project* and the usual *insert_dataset* for the *Experiment*.

The API has been utilised instead of a different approach due to its security, flexibility and ease of integration. The API self contained design allows it to be seamlessly integrated to different architectures written in different programming languages. Additionally, the API handles encryption, user authentication and monitors data custody. Also the API is asynchronous what allows for multiple users accessing it at the same time while other architectures do not offer the same accessibility hence limiting the ability to collaborate on the same data in real time.

2.3.2 Authentication

The last part of the functionality encompasses the user handling and authentication functions. These are mostly handled within the object *User_Auth*. This object performs the necessary encryption, handles checking whether the user's credentials are correct and also updates the user's status to ensure that credentials used to gain access are current and correct.

Authentication is performed in three specific ways. The first way is during the user creation process. Firstly the interface fetches the public key from the API server using the *return_public_key* call. The API is supplied with the username and password which then is used to create a user. The asymmetric key encryption has been implemented this semester. During the user creation the passwords are hashed with randomised *salt*. These hashes are then stored and compared when user wants to authenticate again.

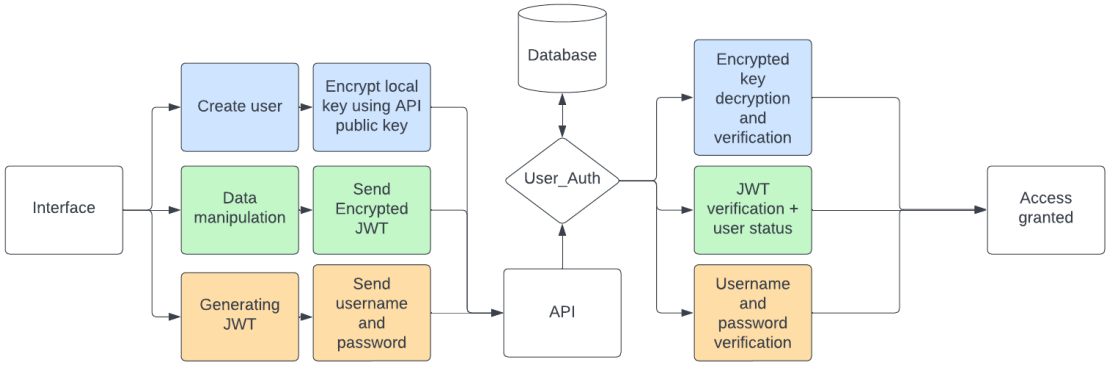

Fig. 4. The diagram showcasing the flow of data in the authentication process. These processes can be split into three categories: open endpoints (blue), data manipulation (green) and token generation (orange). The open endpoints path of authentication includes changes made in this semester.

The remainder of the authentication is done using a *JSON Web token (JWT)* [26]. This token is self contained and contains information regarding the user accessing it. For example, their username and the expiry date of the token. The first type of authentication is the generation of the token using the username and password. This JWT is saved in the user database entry with a time at which the token expires. The second type is the verification of the JWT. This is done by the combination of the username and the token. This token is supplied and verified to be not-expired and matching the username provided.

The *User_Auth* class is responsible for the core of the system's security. This class is initialized whenever authentication is required on the server side, and its functions are used to perform the necessary actions for authentication. These functions include verifying if the user with the given credentials exists, hashing and authenticating the received token.

The authentication process for a user starts with the token generation function. When a user enters their username and encrypted password, the password is encrypted again using the *salt* generated during user creation. If the resulting hash matches the one stored in the database, a token is

generated, and the user's status is updated to "activated". This token expires after 30 minutes, and the user's status is then changed to "deactivated". All other functions use this generated token for authentication. The token is returned and stored within the interface for the duration of the session. Once the token expires, attempting to use the same token for authentication will fail, even if the token is correct.

It is worth noting that the password hashes are compared using a compare digest function that takes the same amount of time, even if some characters match. This is done to prevent time-attacks, where attackers try to decode a password by making multiple calls and timing how long it takes for the server to respond. If the server takes longer, it indicates that the last character is correct, enabling a more efficient brute-force approach, which can reduce the security of the authentication. By employing the compare digest function to compare the hashes, the system's security is enhanced, making it more robust [27]–[29].

2.4 The User Interface

Majority of functionality mirrors the API calls described in Section 2.3. The list below is self-explanatory but it is worthwhile to mention the functionality brought by author query and meta search functions. These two functions along with group management functions allow for transforming the interface from a series of simplistic calls that allow for data to be put in and extracted into a searchable data management software. The function *author_query* allows for a list of datasets fitting the author or a group to be returned as a tree of names. Additionally, *meta_search* allows for searching user-defined meta data. This pushes the functionality beyond the form of the data structure as meta data connections can be used to achieve further complexity. For example, it is possible to use meta data to link datasets and then return them together. This is what has been done in Section 3 to allow for dataset fragmentation to allow for scalability. This opens the conceptual door for the user allowing for implementation of complex data structures like linked lists or graphs.

The user interface functionality includes:

1. Data handling functions - insert/return data
2. Author query - returns a tree of data containing the specified author
3. Check that a node exists - a set of functions which verify that a *Dataset*/*Experiment*/*Project* exists
4. *Experiment* and *Project* initialisation functions
5. User handling functions - these include token generation and user creation
6. Functions that return a list of names of *Projects*/*Experiments*/*Datasets*

7. Author and Group functions - These functions allow for adding an author to a dataset or adding a dataset to a group. Both actions are performed by modifying the meta data
8. Meta data search - This function allows for a return of a tree of objects' names given a meta data dictionary

3 Methodology

3.1 Introduction

The goals listed in Section 1.2 are implemented along the development path this semester. The details regarding how each goal was accomplished are listed in this section. The progress on the project is composed by an introduction of a series of functions utilised to meet a set of objectives.

3.2 Computational Time Complexity

A time complexity is defined by the function that correlates how long an algorithm takes to accomplish a goal given a set of input parameters. Let the time complexity be described using $O(n)$ notation. The value of n represents the number of items that this algorithm operates on. This value represents the quantity on which the algorithm's time complexity scales.

To differentiate the different cases of time complexity different symbols are used. The upper bound time complexity is written as $O(n)$ while lower bound of time complexity is written as $\Omega(n)$. The last symbol used is $\Theta(n)$. This means both O and Ω. If an algorithm has time complexity of $O(n)$ and $\Omega(n)$ then it is $\Theta(n)$ [30]. In this report time complexity is going to be listed using O notation to provide the upper bound for the algorithmic time complexity. Additionally, the complexity tested is estimated by the structure of the code and then calculated using obtained data from testing. It is important to mention that only dominant terms are listed. For example if an algorithm possesses $O(n + m^2)$ time complexity then it is listed as $O(m^2)$ [31]. Additionally, the time complexity measured will always be the average as the measurements are performed over multiple runs using the same set of input parameters.

3.3 The Development Path

3.3.1 Improvements in Efficiency

Goal number 1 was accomplished by the introduction of caching and multi-threading along with stability changes made to the code. The testing of improvements made to the code is conducted by the use of a *send-fetch-cycle*. This function is present within *testing_interface.py* and returns a difference in seconds that it takes to send a *Project* containing a specified number of *Experiments*/*Datasets* to the database and then return it. Given this process contains the core functionality of the architecture and needs to scale appropriately with the size of data being processed it was chosen as the testing case. The function was run across different values (as seen in the plots) and the time complexity was then fitted with the appropriate polynomial. Those values are written out in full in Appendix E. The errors on the values were obtained by running the tests multiple times and using the standard deviation as the representative error for a given value of input parameters.

It is important to recognise that the duration to complete the cycle is not as important as the time complexity of the process. This is due to the ability of a faster machine being able to decrease the run time of constant time complexity algorithms ($O(1)$) by a fixed value. However, when the time complexity is exponential ($O(e^n)$), even a high-end machine will severely slow down for higher values of n. Therefore, it is essential to prioritise efficient algorithm design with best possible time complexity regardless of machine speed. This as a consequence allows for optimal performance on low-end machines. The use of a local MongoDB and a faster internet connection significantly reduces the duration of the send-fetch cycle. This time can vary from 0.23 seconds to as long as 120.00 seconds for the same value of n. Hence, it is important to make changes to algorithm design to address time complexity and not absolute time taken.

The improvements made can be divided into four parts. Firstly, the data insertion algorithm have been implemented with using multi-threading to improve the speed of the process. Due to the nature of the data structure it is possible to populate the tree branches in any order once the root is established. This means that the API calls can be made in any order as long as one layer if finished before another one starts. This is a good place to introduce parallel processing. By inserting experiments and then inserting all datasets at the specified paths in the correct order the compiler can choose the most efficient processes to prioritise hence enabling for faster processing. This theoretically should allow for faster processing in certain cases in which the number of calls is large. However, as seen in Figures 5 and 7 it provided minimal improvements within the architecture.

The second improvement is a change to the connection made. Previously each API call was run using a single call utilising the library get or post function [25]. This required a generation of a port and a certificate to establish the connection. This was simplified by using a session. This session maintains the connection while all actions are performed. This cuts time by not requiring repeated certificate generation and port assignment.

The third improvement was made by streamlining code design. For example, the removal of unnecessary loops or changing data structures used from dynamically generated lists into dictionaries. New data structures and especially data structures specialised in their use case are a key component of efficient code. For example a dynamically generated list has $O(N)$ look-up time while a dictionary maintains a $O(\log N)$ look-up time. This means that to handle large amounts of data which needs to be searchable it is better to use dictionaries than dynamic arrays.

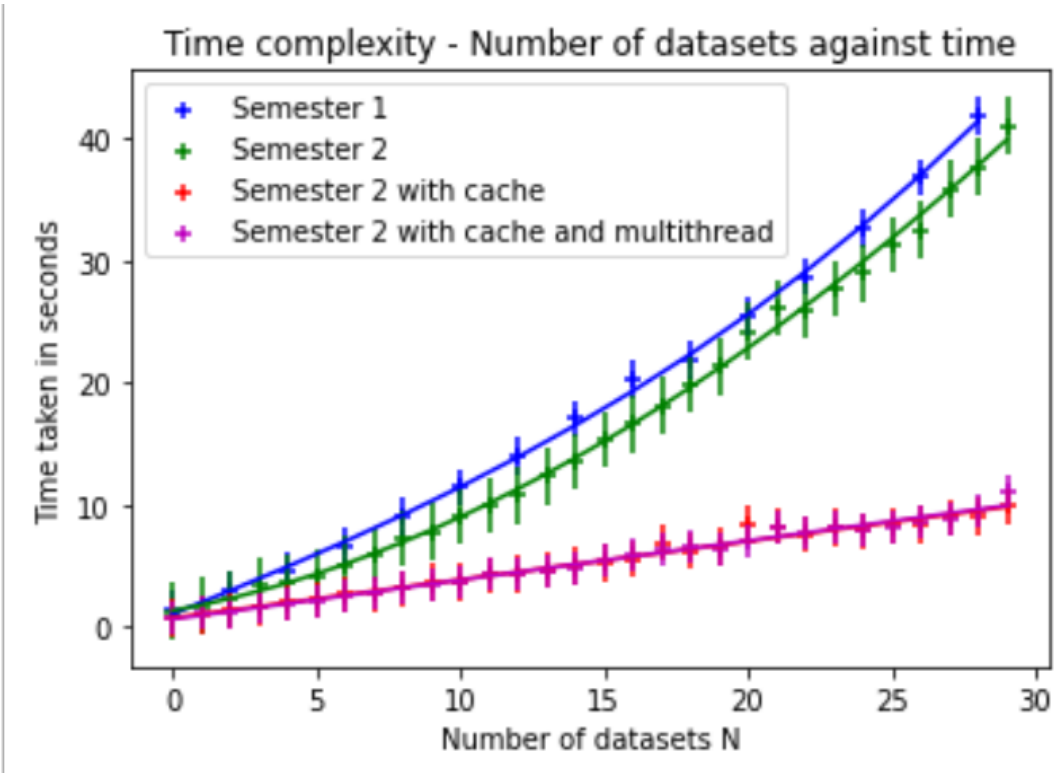

Fig. 5. Time complexity of the *send fetch cycle* using different versions of the architecture. The change in complexity is demonstrated for the number of 64 bit integers being contained within the dataset. The data points for cache and cache + multi-threading are overlapping each other.

The last improvement is *caching*. If a process is performed multiple times to receive the same or similar data it may be advantageous to store the data and update it only if necessary to prevent the same function being run multiple times. This is best demonstrated in the set of functions that receive names of nodes in the data structure. These functions check whether the specified data already exists on the database. This is done to prevent duplicate entries. However, in the previous architecture these functions were called every time a piece of data was to be inserted. This has to be the case due to each call being independent and asynchronous. Hence, to simplify this process and not require repeated calls a cache can be created of the data available to the user. This cache contains a tree of names of the data nodes available to the user. This can then be changed on the client-side if any changes are made. However, it is important to note that the gain in processing speed can bring with itself a flaw caused by data collisions. Suppose two different users using the system want to insert a dataset with the same name. The user whose function executes first will be able to do so while the other call will pass the initial check but fail at the API call level. This exception has been handled but caching can cause the data collision issues to bubble up to the API level and can only be minimised by ensuring that the local cache has a timeout period after which is it

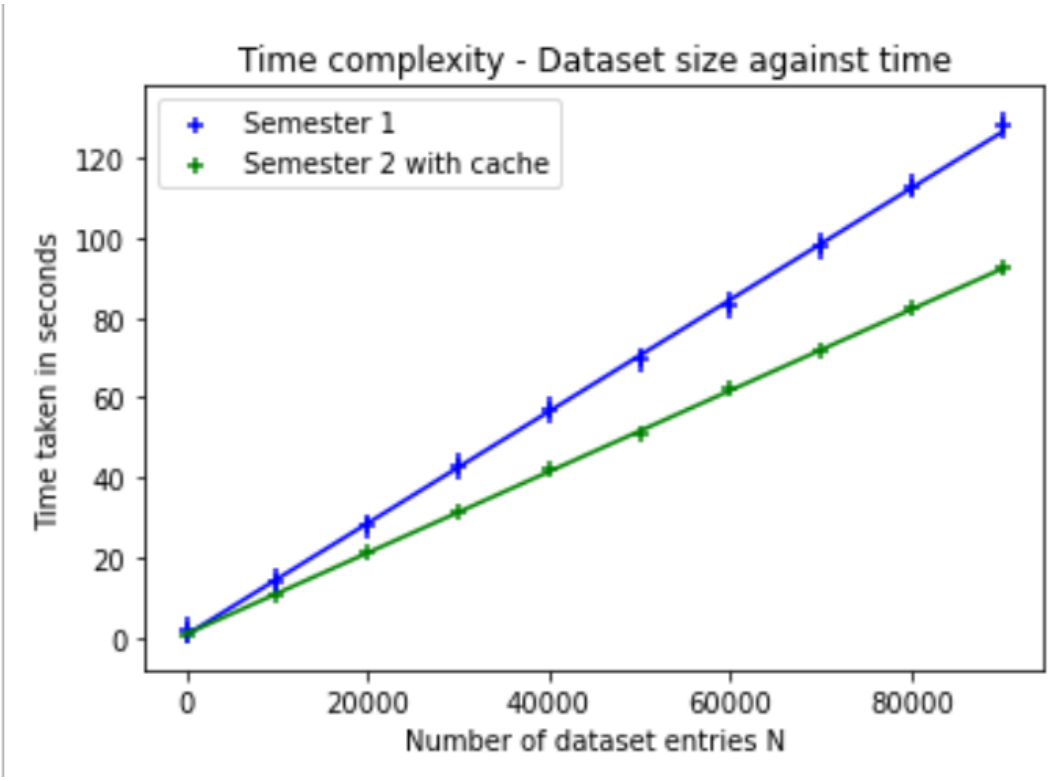

Fig. 6. The time complexity for the *send fetch cycle* using different versions of the architecture. Each dataset contains 1000 entries of 64-bit integers with the number of datasets being variable. The plot for Semester 2 architecture without improvements overlapped Semester 1. The multi-threading improvements show no change comparing to the version utilising only cache. Hence only the caching plot was included.

synchronised with the database. The shorter the period the less chance of data collision but the lower improvement in efficiency. Hence, the period of 1 minute was chosen.

A hierarchical tree was chosen as the data structure to store the names of the data structure. This has been implemented within the file named *data_handle.py*. This tree contains routines that insert a node, clear the tree, populate the tree (within the *constructor*), delete the node and check that a node exists. The reason for a tree data structure implementation is due to the very fast look-up time. The lookup time within this tree follows a $O(\log(n))$ relationship with the tree population having a $O(ped)$ time complexity. For p being the number of projects, e being the number of experiments and d being the number of datasets with $p \leq e \leq d$ and $p << d$ being the average case. This provides an efficient data structure with very fast lookup times what is essential for this use case.

Overall, the changes made within Semester 2 brought with itself a series of bug fixes but the handling of exceptions slows down the code by adding more checks and ensuring that the logic of the algorithms is correct. This can be best demonstrated in Figure 7 where the semester 2 architecture has a constant offset as compared to semester 1. This is due to the added $O(1)$ time complexity exception handling. The semester 1 architecture performs better than the non-feature enabled semester 2 version but this increase in performance brought with itself the increased instability. In general, the implementation of caching changes the complexity of the architecture dramatically

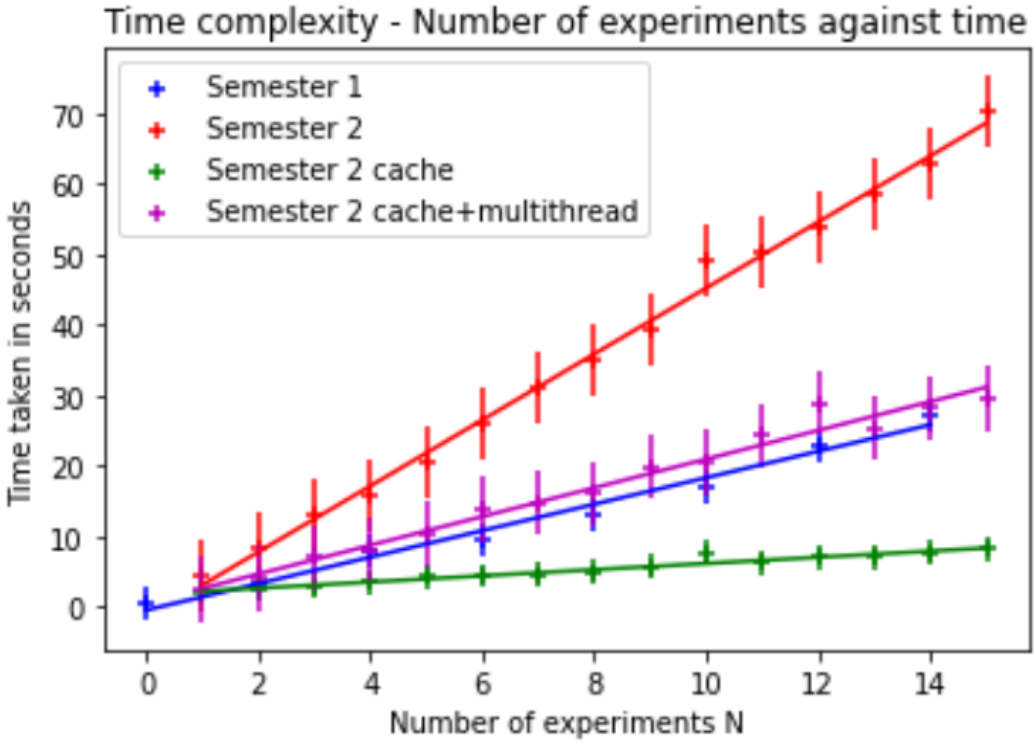

Fig. 7. The time complexity plot for the *send fetch cycle* using different versions of the architecture. The number of experiments is variable while each experiment contains 1 dataset each containing 1000 64-bit integers.

due to the removal of unnecessary API calls. Multi-threading shows performance improvements and especially when a long list of API calls has to be made as seen in Figure 7. However, it is important to mention that in the case of multiple experiment insertion the use of cache alone presents the best unmatched efficiency. Hence, further investigation and improvements would have to be made.

The time complexity of inserting large size datasets remained $O(n)$. However, it is worth noting that the gradient of the scaling has improved hence demonstrating a more efficient process, as seen in Figure 5. The common thread between the tests conducted pushes caching as the best way of improving efficiency. Caching reduces the time complexity from $\approx O(n^3)$ to $O(n)$ in the case of the number of datasets and lowers the gradient in the other two cases. Multi-threading has marginal effects and causes issues with the implementation of stable connections using sessions. This is due to the different threads using different ports to access the API while the caching only method uses the same port.

3.3.2 Experimental Data and Analysis

An important aspect of making the pipeline accessible to the user is by introducing an intuitive interface. The current module named *interface.py* contains the functions described within Section 2.4. These functions perform the actions as necessary by using the interface but they accomplish

the functions by requiring the same details for each call and require a specific data structure as input variables. For example, the insertion of a dataset requires a Python object to be created using a data structure named *Dataset*. To insert any data into the dataset it is required to authenticate every 30 minutes on top of executing the function. The interface (see Section 2.4) does not store any data and only provides enough functionality to allow the user to communicate within the server. By its nature it does not provide the necessary functionality to work as a working repository for data. This is addressed by the introduction of a *simple interface*. This interface allows the user to create data locally on the client side and then once the data is fully compiled it can be synchronised to the database. Additionally, the return functions allow the user to fetch the data and store it locally for future processing. The data is stored within working memory and is only meant to store parts of the project to allow for easier handling. The local storage of data and synchronisation takes advantage of the *caching*.

In conclusion, this approach allows for a more intuitive data manipulation as the functions need to only be provided with variables that are required by the data structure. Additionally, the simple interface is populated with *Exception* calls which interrupt code if the operation fails at any stage while the original interface would not notify the user of the issues but rather only return a HTTP status code of the server or a False *Boolean* value. This provides the user with the ability of monitoring their actions across the client-side to server-side data journey. The previous interface is not inherited by this object but rather initialised as a variable within the *Simple_Interface* class.

The storage of the data is accomplished by the use of a nested list. No specialised data structure has been implemented to allow the user to fetch the data directly from the object within a recognizable form and make changes to it in a programmatic manner. Hence, the architecture is more accessible to the user.

Function name	Function purpose
create_user	This function is used to create a user. Calls the corresponding interface function.
user_authenticate	Allows the user to authenticate and saves the credentials locally.
check_user_authenticated	Returns True if a user has authenticated and False otherwise.
insert_empty_project	Inserts an empty project locally into a data structure.
check_project_exists	Returns True if a project exists locally. Otherwise it returns False.
get_project	Returns the *Project* stored locally.
get_experiment	Returns the *Experiment* stored locally.
pop_project	Deletes the *Project stored locally*.
insert_empty_experiment	Inserts an empty *Experiment* to the local data structure within the specified *Project*.
pop_experiment	Deletes the locally stored *Experiment* from the data structure.
check_experiment_exists	Returns True is an *Experiment* exists locally. Otherwise it returns False.
check_dataset_exists	Returns True is a *Dataset* exists locally within the data structure. Otherwise it returns False.
insert_dataset	Inserts a *Dataset* locally into a data structure.
pop_dataset	Delete the locally saved *Dataset* from the data structure.
get_dataset	Returns the specified *Dataset* from the local structure.
sync_data	Updates the locally stored data from the data structure to the database by adding the parts which do not exist on the database.
return_project	Adds the database stored *Project* to the local data structure. Returns the *Project*.
return_experiment	Adds the database stored *Experiment* to the local data structure. Returns the *Experiment*.
return_dataset	Adds the database stored *Dataset* to the local data structure. Returns the *Dataset*.
clear_local_files	This functions clears the locally saved data

Table 1. Table summarising the new functions added by the *simple interface*.

3.3.3 Authentication improvements

The improvements to the authentication system follow directly from the last semester's critical evaluation. The proposed full implementation of OAuth2 [32] is a secure method however, due to the nature of the API system a more secure method has been developed. The communication required is only one-sided as the interface is only required to perform a single open-end call. This is done to create a user using the *create_user* function. The changes made allow for a full encryption of the *request body* that is sent to the API as a part of the function. This encryption is performed using a public key obtained directly from the API using a new API call named *return_public_key*. This causes the API to generate the public and private key pair. The public key is then sent to the interface for the purpose of this function. Once the request body is fully encrypted and sent then it is decrypted using the private key generated. Due to the length of the data encrypted compared to the key length this provides a very robust form of communication. Additionally, the data sent to the database is fully protected by a man-in-the-middle attack and even if intercepted cannot be used. Additionally, because the keys are generated specifically for the purpose of this call the security is further increased as only the interface can obtain the key. This gives 3 layers of protection. Firstly, the interface and API have a secure tunnel generated using an environment variable key. Secondly, to access the API without valid credentials the communication is fully encrypted and a second secure tunnel is generated for the purpose of authentication. Finally, once the credentials are given the token used for communication needs to be checked against user credentials and is itself encrypted. Furthermore, this token expires after 30 minutes hence needs to be re-generated what is done through the securely added credentials.

The improvements in authentication have been completed and the open back-end function now employs full-encryption. This means that the goal has been accomplished fully. Full asymmetric key encryption provides the best form of security that the open back-end can require. Additionally, this allows for the implementation of fully encrypted data transmission as a possible avenue for future development as the functionality completed for this goals would allow for such an extension.

3.3.4 Scalability and Dataset Fragmentation

Due to the large size of testing images that were used in the image handling functions the development of *Dataset* fragmentation functions was necessary. This was done to allow the large *Datasets* to be send. This fragmentation is required by the restriction of the *MongoDB PyMongo* library requiring a strict limit in the size of each document that it handles. Now the fragmentation is performed automatically if the *Dataset* exceeds this limit. When a dataset is inserted it can either follow the normal path if its size is small or it can become fragmented. Fragmentation is performed by slicing the *data* variable within the *Dataset* object into two datasets. This is performed recursively until the datasets generated all meet the size requirements. The biggest theoretical oversight is the assumption that the *data* variable is going to be the biggest. It is possible that a metadata vari-

able exceeds the size. The size limit is $15MB$ and given that a dictionary of that size would contain 250,000 64-bit characters. This roughly estimates to about 50,000 words in meta data. Hence, it is a justified assumption that the meta data will not exceed the limit. In the unlikely case that it does the API calls are asynchronous and will fail and proceed to the next one causing no disruption to the database.

The *Dataset* that is fragmented is split into a front dataset which acts as the part visible to the user and dataset fragments. Those fragments do not contain an author and therefore, are invisible to the user. These fragments are linked by two meta data variables: *parent_dataset* and *dataset_fragment_id*. These datasets are recalled, sorted by the fragment ID number and then collected into a single *Dataset* and returned by the *return_dataset* function as any other dataset.

The security of the data stored on the database is mostly accomplished by putting only the data the user is authenticated to view within their scope. This is performed by providing only the paths to the documents that the user is listed as an author. Hence, to allow for data fragment collection a new API call had to be written. This function is named *collect_frag_data* and operates on documents available outside of the scope of the user by fetching fragments which do not posses an author. This can potentially lead to a user being able to gain access to datasets of other users. This issue was addressed by making sure that the fragmented data is split into two datasets and the *front dataset* is the only avenue of accessing the fragments. When the user calls a function to collect the dataset fragments the front dataset has to be provided. This name is then passed to the API and the credentials of the user are authenticated against this dataset on the server side. Once the checks are passed the user retrieves the names of the fragments outside of the scope and the datasets are returned within a list. This list is then sorted using a *Quick Sort* and the full dataset is returned. Hence, in summary it is possible to fetch datasets outside of the scope but the knowledge of the variables required makes the endeavour too time consuming and even if successful at least half of the data will still be missing. The amount of time required is made much larger by using generated unique hashes as names for the fragment datasets.

The implementation of scalability brings with itself the idea of an average case versus worst case. In this architecture it is assumed that the usual size of a dataset would be $\approx 15MB$. The implementation of scalability by data fragmentation allows for the expansion of that limit at least double that size without a significant loss in performance. However, the lifting of the cap on the side of the interface does not remove the limiting size of the database used. Overall, the goal has been successfully met and the scalability of data has been implemented and tested. The function responsible for the largest performance impact on data processing is responsible for calculating the size of the object provided. This is due to the the time complexity of the function being $\Theta(n)$ for n being the number of entries within the dataset. This decrease in performance was mitigated by the improvements in efficiency like caching and multi-threading.

3.3.5 Image Handling

Image handling has been implemented by utilising the *Pillow Python library* [33]. This library allows for the conversion of an image file into a *ndarray* as defined by the *Numpy library* [34]. This array is composed of a list of pixels with each pixel being represented by a list of three hexadecimal values representing the three colours Red, Green and Blue (RGB). This array is then treated as a special type of list. The list is converted from the *ndarray* into a regular Python list. This is then wrapped into a *Dataset* and treated like a regular dataset. The only difference is an addition of image type into the meta data. This is then utilised to convert the dataset back into the desired type. A limited list of image types is restricted. These are restricted by the convention of bytes used in their encoding. The allowed data formats are: unsigned 8-bit, 64-bit integers, unsigned 16-bit integers and 32-bit float numbers. The remaining types of encoding are not used often enough to implement. This restriction is an exception during the conversion of an image into a *Dataset*. Image handling is tested in two steps. The first step checks whether the image is converted into a *Dataset* (test 12). The second test checks the insertion of the image into the database and its successful retrieval (test 13). The full list of tests is available in Appendix A. The only thing worth mentioning that it does not preserve header data. This is due to the image not being handled as an image but rather a set of data that can be displayed in a visual form. There is no need to preserve header data as any relevant data should be inputted within the meta data variable. Overall, the goal was met and image handling is a working function of the architecture. It can be argued that the header data should be converted into *meta-data* upon conversion but this was not implemented in this project due to time constraints.

3.3.6 Jupyter Drivers and Notebook

The changes in the *jupyter_driver* file included the addition of H5 file [35] handling functionality. A big aspect of using this architecture relies on the ability of inserting non-python native file formats and converting them into a *JSON file* (test 14). This function unpacks the H5 file into a JSON file and then populates a project with the data contained within the file without any loss of data. This demonstrates the capability of handling large files. Additionally, a MATLAB file is included within the repository and this demonstrates how a driver would be written to allow compatibility across languages. This is important as it allows different researchers with different approaches cooperating on the same data simultaneously.

4 Critical Evaluation

4.1 Goal Success Evaluation

The project was shown to demonstrate the functionality on multiple machines while utilising a virtual machine as the server. This demonstration was conducted on the 02/05/2023 and demonstrated the usage across two different machines. Additionally, the architecture allowed for the insertion of experimental data using the function explored in Section 3.3.6. This demonstrates that the core functionality of the architecture operates as it is supposed to and can provide a utility by its use in data handling with obtained experimental data.

Function	CKAN	Summer 22	Semester 1	Semester 2
Authentication	No	Yes	Yes	Yes
Client-to-server encryption	No	No	No	Yes
Data groups	Yes	No	Yes	Yes
Data insertion and retrieval	Yes	Yes	Yes	Yes
Data sharing	Yes	No	Yes	Yes
Image handling	Yes	No	No	Yes
Scalable	Yes	No	No	Yes
Searching using meta data	Yes	No	Yes	Yes

Table 2. Summary the changes in functionality across project duration as compared to CKAN.

Overall, all goals have been met. The architecture was changed to allow for a more efficient, more secure design and additionally the expansion in the functionality allowed for image handling. The changes in the architecture are explored in Section 3 and the efficiency changes show a provable increase in efficiency. The project was an overall technical success.

The main concern with this architecture remains its accessibility. The addition of a simple interface does not fully remedy the fact that this code base is complicated and does not hide its complexity from the user. The biggest drawback of the architecture can be summarised as its opaqueness to learning how to operate it. This has been only partially addressed by the production of readable documentation using PyDoc [36], the simple interface and the good practices in naming schemes.

Additional problem has been discovered during the implementation of multi-threading as a way of improving a efficiency of the send fetch cycle. The fact that the separation of functions into different threads causes crashes once the number of different threads reaches a high enough number. This usually is addressed by monitoring of the threads and by the use of *Concurrency*. This can implemented using the *Concurrent.futures* module [37]. However, this was not implemented within the project but in its place a more simplistic approach to threads was used. In general, an improvement in efficiency is seen but given this API utilises asynchronous calls it is possible to increase

the efficiency even more dramatically. However, due to time limitations this was not possible but should be seen as a possible future expansion of the architecture.

4.2 Other Technology

CKAN is an open source data management system that has been utilised by many government and private organisations [38]. It is the best direct comparison to the pipeline designed in the duration of this project. It allows for data storage, sharing and possesses nearly identical functionality to this architecture. Hence, it will be a good point of comparison and distinction between the two. CKAN is written mostly in Python and utilises a web-based interface to simplify the access to its data. Additionally, its main purpose is collecting data into manageable and searchable pages. For example, the implementation of CKAN on the Data.gov website allows the query of publicly available data stored by the government of the United States. This allows for a generation of a page which provides the access to a dataset or a summary of the data. This is done by the integration of an API with a website of the client. The important point to make is that the data is publicly available hence there is no authentication system in place. To identify the weaknesses in the architecture it is best to compare it to CKAN. The biggest glaring issue is the difference in accessibility between platforms. CKAN is a web based archive which can be queried the same way a search-engine can while this architecture uses a meta data system which needs to be specified by the user and relies on querying system to be delegated to the user during dataset insertion. Additionally, CKAN allows for data modification while this architecture was designed specifically to not facilitate data alteration.

While CKAN is the best comparison there are many other forms of data management that are currently used within the scientific context. The comparison can be seen in Table 2 .For example, ROOT is a tree data structure used for storage and experimentation on particle physics data obtained from CERN [6]. Another example is GIT. GIT is a software package that is an industry standard for monitoring and managing resources in any coding project [39].

5 Future Work

There are several areas where further development could improve its functionality and usefulness. One potential avenue for future work is the creation of a graphical user interface (GUI) for the pipeline. While the current implementation is intended for use by researchers with a strong coding background and familiarity with APIs, a GUI could make the pipeline more accessible to a broader range of users. A well-designed GUI would enable users to interact with the pipeline more easily and intuitively, which would increase its overall utility.

Another area for potential development is the optimization of the pipeline's performance. While

the current implementation employs multi-threading and caching to improve efficiency, there may be additional ways to enhance its performance. Additionally, the exploration of more advanced multi-threading techniques could be a potential pathway for future development. For instance, incorporating machine learning techniques could potentially lead to faster and more accurate data analysis, further improving the pipeline's efficiency.

In addition to these technical improvements, the pipeline's functionality could also be expanded. For instance, the pipeline could be extended to incorporate additional data sources in different languages or to support additional analysis techniques like the use of machine learning to train AI to recognise patterns in data in real time. By increasing the range of research questions that can be addressed using the pipeline, it could become a more valuable tool for researchers in the semiconductor field. Overall, there are several exciting possibilities for future work based on the current project. By exploring ways to enhance its usability, performance, and functionality, researchers can leverage this tool to gain new insights and advance the field of semiconductor research.

References

[1] M. A. Dennis, *Tim berners-lee*, 2023. [Online]. Available: https://www.britannica.com/biography/Tim-Berners-Lee (cited on p. 5).

[2] S. Braverman, "Global review of data-driven marketing and advertising," *Journal of Direct, Data and Digital Marketing Practice*, vol. 16, pp. 181–183, 3 Jan. 2015, ISSN: 1746-0166. DOI: 10.1057/dddmp.2015.7 (cited on p. 5).

[3] N. Silver, *The Signal and the Noise: Why Most Predictions Fail – but Some Don't*. Penguin Books, Feb. 2015, ISBN: 978-0143125082 (cited on p. 5).

[4] N. Donratanapat, S. Samadi, J. Vidal, and S. S. Tabas, "A national scale big data analytics pipeline to assess the potential impacts of flooding on critical infrastructures and communities," *Environmental Modelling & Software*, vol. 133, p. 104 828, Nov. 2020, ISSN: 13648152. DOI: 10.1016/j.envsoft.2020.104828 (cited on p. 5).

[5] F. Idachaba and M. Rabiei, "Current technologies and the applications of data analytics for crude oil leak detection in surface pipelines," *Journal of Pipeline Science and Engineering*, vol. 1, pp. 436–451, 4 Dec. 2021, ISSN: 26671433. DOI: 10.1016/j.jpse.2021.10.001 (cited on pp. 5, 6).

[6] F. C. R. Rademakers, "Root-project/root: V6.18/02," *Zenodo*, 2019. [Online]. Available: https://zenodo.org/record/3895860 (cited on pp. 5, 23).

[7] S. M. Mennen, C. Alhambra, C. L. Allen, *et al.*, "The evolution of high-throughput experimentation in pharmaceutical development and perspectives on the future," *Organic Process Research & Development*, vol. 23, pp. 1213–1242, 6 2019. DOI: 10.1021/acs.oprd.9b00140 (cited on p. 5).

[8] C. Akin, L. C. Feldman, C. Durand, *et al.*, "High-throughput electrical measurement and microfluidic sorting of semiconductor nanowires," *Lab on a*

Chip, vol. 16, pp. 2126–2134, 11 2016, ISSN: 1473-0197. DOI: 10.1039/C6LC00217J (cited on p. 5).

[9] N. Wang, Y. Cai, and R. Zhang, "Growth of nanowires," *Materials Science and Engineering: R: Reports*, vol. 60, pp. 1–51, 1-6 Mar. 2008, ISSN: 0927796X. DOI: 10.1016/j.mser.2008.01.001 (cited on p. 5).

[10] N. Wang, X. Yuan, X. Zhang, *et al.*, "Shape engineering of inp nanostructures by selective area epitaxy," *ACS Nano*, vol. 13, pp. 7261–7269, 6 Jun. 2019, ISSN: 1936-0851. DOI: 10.1021/acsnano.9b02985 (cited on p. 5).

[11] P. Parkinson, "Physics and applications of semiconductor nanowire lasers," vol. 20, pp. 389–438, Jan. 2021, ISSN: 1876-2778. DOI: 10.1016/B978-0-12-822083-2.00010-1 (cited on p. 5).

[12] E. H. Parker, *The technology and physics of molecular beam epitaxy*. Plenum Press New York, 1985 (cited on p. 5).

[13] A. Cho, "Recent developments in molecular beam epitaxy (mbe).," *J Vac Sci Technol*, vol. 16, no. 2, pp. 275–284, 1979. DOI: 10.1116/1.569926 (cited on p. 5).

[14] H. Tsuchiya, K. Sunaba, T. Suemasu, and F. Hasegawa, *Growth condition dependence of GaN crystal structure on (001)GaAs by hydride vapor-phase epitaxy*. 1998, vol. 189-190, pp. 395–400. DOI: 10.1016/S0022-0248(98)00322-4 (cited on p. 6).

[15] N. Wang, Y. Cai, and R. Zhang, "Growth of nanowires," *Materials Science and Engineering: R: Reports*, vol. 60, pp. 1–51, 1-6 Mar. 2008, ISSN: 0927796X. DOI: 10.1016/j.mser.2008.01.001. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0927796X08000028 (cited on p. 6).

[16] N. Chand, "Mbe growth of high-quality gaas," *Journal of Crystal Growth*, vol. 97, pp. 415–429, 2 1989, ISSN: 0022-0248. DOI: https://doi.org/10.1016/0022-0248(89)90223-6. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022024889902236 (cited on p. 6).

[17] W. W. Wong, Z. Su, N. Wang, C. Jagadish, and H. H. Tan, "Epitaxially grown inp micro-ring lasers," *Nano Letters*, vol. 21, pp. 5681–5688, 13 Jul. 2021, ISSN: 1530-6984. DOI: 10.1021/acs.nanolett.1c01411 (cited on p. 6).

[18] T. Hey, K. M. Tolle, S. Tansley, and A. Hey, *The Fourth Paradigm Data-intensive Scientific Discovery*. Microsoft Research, 2010 (cited on p. 6).

[19] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, *et al.*, "The fair guiding principles for scientific data management and stewardship," *Scientific Data*, vol. 3, p. 160 018, 1 Dec. 2016, ISSN: 2052-4463. DOI: 10.1038/sdata.2016.18 (cited on p. 6).

[20] UKRN, *Uk reproducibility network - webpage*, 2022. [Online]. Available: https://www.ukrn.org/about/ (cited on p. 6).

[21] T. J. Jacobsson, A. Hultqvist, A. García-Fernández, *et al.*, "An open-access database and analysis tool for perovskite solar cells based on the fair data principles," *Nature Energy*, vol. 7, pp. 107–115, 1 Dec. 2021, ISSN: 2058-7546. DOI: 10.1038/s41560-021-00941-3 (cited on p. 6).

[22] D. Mike, *Pymongo documentation*, 2022. [Online]. Available: https://pymongo.readthedocs.io/en/stable/index.html#overview (cited on p. 8).

[23] C. S, *Pydantic documentation*, Sep. 2022. [Online]. Available: https://pypi.org/project/pydantic/ (cited on p. 9).

[24] S. Ramirez, *Fastapi framework*, Open source software, 2023. [Online]. Available: https://github.com/tiangolo/fastapi (cited on p. 9).

[25] R. K, *Requests: A simple, yet elegant http library*, 2022. [Online]. Available: https://github.com/psf/requests (cited on pp. 10, 14).

[26] S. E. Peyrott, *Json web token*, Accessed on 8/12/2022, 2022. [Online]. Available: https://jwt.io/introduction (cited on p. 11).

[27] R. G. Underwood, *Cryptography for secure encryption*. Springer, 2022 (cited on p. 12).

[28] POINTCHEVAL, *Asymmetric cryptography primitives and protocols*. ISTE LTD, 2023 (cited on p. 12).

[29] M. Curtin, *Brute Force: Cracking the Data Encryption Standard*. Copernicus Books, 2010 (cited on p. 12).

[30] G. L. McDowell, *Cracking the Coding Interview*, 6th ed. Palo Alto, CA: CareerCup, Jul. 2015 (cited on p. 13).

[31] S. S. Skiena, *The algorithm design manual the algorithm design manual* (Texts in computer science), 3rd ed. Cham, Switzerland: Springer Nature, Oct. 2020 (cited on p. 13).

[32] E. Pace, M. Woloski, and T. McKinnon, "Auth0 - authentication tool," 2023, Open source software library. [Online]. Available: https://auth0.com/intro-to-iam/what-is-oauth-2 (cited on p. 19).

[33] J. Clark, *Fork of pil - python imaging library*, Open source software, 2023. [Online]. Available: https://pillow.readthedocs.io/en/stable/ (cited on p. 21).

[34] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2 (cited on p. 21).

[35] G. Heber, *Hdf5 1.13.4-1 documentation*, 2022. [Online]. Available: https://docs.hdfgroup.org/hdf5/develop/ (cited on p. 21).

[36] D.-h. Na, *Pydoc documentation*, May 2023. [Online]. Available: https://docs.python.org/3/library/pydoc.html (cited on p. 22).

[37] Python Software Foundation, *Launching parallel tasks*, 2023. [Online]. Available: https://docs.python.org/3/library/concurrent.futures.html (cited on p. 22).

[38] Open Source, "Ckan - the world's leading open source data management system," *CKAN*, 2023, Date accessed: 02/05/2023. [Online]. Available: https://ckan.org/ (cited on p. 23).

[39] S. Chacon and B. Straub, *Pro git*. Apress, 2014, Open Source Software. [Online]. Available: https://github.com/git (cited on p. 23).

[40] G. Currie, "Models as fictions, fictions as models," *The Monist*, vol. 99, pp. 296–310, 3 Jul. 2016, ISSN: 0026-9662. DOI: 10.1093/monist/onw006.

[41] S. Arab, M. Yao, C. Zhou, P. D. Dapkus, and S. B. Cronin, "Doping concentration dependence of the photoluminescence spectra of n-type gaas nanowires," *Applied Physics Letters*, vol. 108, p. 182 106, 18 May 2016, ISSN: 0003-6951. DOI: 10.1063/1.4947504.

[42] P. Yang, R. Yan, and M. Fardy, "Semiconductor nanowire: What's next?" *Nano Letters*, vol. 10, pp. 1529–1536, 5 May 2010, ISSN: 1530-6984. DOI: 10.1021/nl100665r.

[43] T. Neska and P. Parkinson, *Resdata: Python api managing data storage using mongodb database*, 2022. [Online]. Available: https://github.com/SplitSky/resdata.

[44] H. Krekel, B. Oliveira, and R. Pfannschmidt, *Pytest 7.2.0*, Oct. 2022. [Online]. Available: https://pypi.org/project/pytest/ (cited on p. 31).

[45] M. Das and S. K. Ghosh, "Data-driven approaches for meteorological time series prediction: A comparative study of the state-of-the-art computational intelligence techniques," *Pattern Recognition Letters*, vol. 105, pp. 155–164, Apr. 2018, ISSN: 01678655. DOI: 10.1016/j.patrec.2017.08.009.

[46] UCAR/Unidata Program Center, "Integrated data viewer (idv) software," 2022. DOI: http://doi.org/10.5065/D6H70CW6.

Appendices

A Libraries and Technical Details

This architecture uses libraries to allow for its functionality. The libraries and their purpose are listed in Table 3

Library name	Library function
Pydantic	Objects that validate data contained within them. Allows to specify the type of variable to be enforced
Pymongo	Interface used for communication between the API and the database
FastAPI	Tools used to write functions callable by HTTP
requests	Library for making HTTP calls using Python
typing	Library containing data types
hashlib	Library containing hashing and security algorithms. Used in authentication
matplotlib	Data plotting and display
Numpy	Numerical calculations and statistical functions
datetime	Library used for operating on date and time variables
jose	Tools for managing and generating JWT tokens
secrets	Contains functions concerned with data security. Used in Authentication
Threading	Library used to implement multithreading in Python

Table 3. Table summarising the libraries used.

B Testing and Continuous Development Cycle

The testing has been conducting using the Pytest library [44]. All tests have been successful as of May 16, 2024. The tests and their purpose are listed below. The testing functions are within a file named testing_interface.py.

C Glossary

List of words that are defined for the need of this report:

- data pipeline - A set of processes used to automate the movement and transformation of data between a source system and a target repository.
- pull/fetch - To request and receive data from a data repository
- Node - A point in a tree structure where data is contained.
- API - Application Programming Interface - In the context of APIs, the word Application refers to any software with a distinct function. Interface can be thought of as a contract of service between two applications. It refers to a piece of code operating as a handler between two nodes of a data journey.

Test number	Test purpose
0	Check the connection of the API to the database
1	Verify that a user is created
2	Insert a project using previously created user
3	Insert a project. Then return it and compare to the one inserted
4	Authenticate a user and return a JWT token
5	Tree print the data associated with a user after authentication
6	Adding authors and verifies an author was added successfully
7	Tests that the generate_optics_project generated the Ring object successfully
8	Verifies that the search by meta variable works
9	Tests the group feature and adding authors to the group
10	Tests the author query function
11	Testing the add author to group functions on the level of the database, experiment and project
12	Test which creates and inserts a parsed image into a database
13	Sending and retrieval of image datasets testing
14	Population of the database using a h5 file
15	Insert project function using multi-threading
16	Insert project function using multi-threading and cache
17	Simple interface functions testing

Table 4. Table summarising the tests conducted.

Function name	Module name	Function purpose
purge_function	API_server	Clears the database of all data
return_all_dataset_names_group	API_server	Returns the names of the datasets within a group
return_all_experiment_names_group	API_server	Returns the names of the experiments within a group
return_all_project_names_group	API_server	Returns the names of the projects within the group
add_group_to_dataset	API_server	Adds a dataset to a group
meta_search	API_server	Returns the names of datasets containing a meta data variable
return_hash	interface	Shake 256 hashing algorithm
purge_everything	interface	Clears the database of all data
experiment_search_meta	interface	Fetches the datasets matching the metadata variables
add_group_to_dataset	interface	Adds a dataset to a group
add_group_to_experiment	interface	Adds an experiment to a group
add_group_to_experiment_rec	interface	Adds an experiment and all datasets contained within it to a group
add_group_to_project	interface	Adds a project to a group
add_group_to_project_rec	interface	Adds a project and everything contained within it to a group
add_group_to_dataset_rec	interface	Adds a dataset and the things that contain it to a group
get_experiment_names_group	interface	Returns the names of the experiments within a group in a specified path
get_dataset_names_group	interface	Returns the names of the datasets within a group in a specified path
get_project_names_group	interface	Returns the names of the projects within the group
unpack_h5_custom	jupyter_driver	Function used to unpack the experimental data h5 file
save_dataset	jupyter_driver	Function used to save a dataset as a *JSON file*
append_name	jupyter_driver	Function used to append a name of a dataset to a names.txt file. This file is used to manage the datasets saved locally
send_dataset	jupyter_driver	A testing function used to send all datasets saved locally from the names.txt file.
author_query	interface	Returns the list of data nodes containing the author/group specified
tree_print_group	interface	Prints the structure of a group in a readable manner
plot_from_dataset	jupyter_driver	Print the data from a dataset passed in
summarise_dimensions	jupyter_driver	Summarise the mean of the quantities in the datasets passed in
wrap_dataset	interface	given a dataset returns a project and experiment wrapper
generate_dataset_for_list	interface	creates a dataset from the supplied variables
generate_img_from_dataset	interface	converts a dataset into an image file
generate_dataset_for_img	interface	creates a dataset given an image file
collect_frag_data	interface	given a front dataset recombines the data fully
fragment_datasets	interface	splits the dataset using slice_array and returns a list of datasets
slice_array	interface	the function used to splice the list into two parts
check_object_size	interface	function used to check whether the object is within the bounds of size to be sent using the API
convert_array_to_img	interface	a function converting the array of numbers into an image file
convert_img_to_array	interface	converts an image file into an array of numbers

Table 5. The full list of functions added to the modules during this project. Functions contained within the module API_server are public API calls.

- JSON - JavaScript Object Notation - A type of data storage by the use of an often nested dictionary.
- Version control - It's a system of software that helps the developer track changes in code over time. As a developer edits code, the version control takes a snapshot of the files. It then saves the snapshot permanently so it can be recalled later if needed.
- salt - Random input which is used as an additional input to a one-way function that hashes the data.
- data custody - It refers to the process of having the legal right and authentic control over particular set of data elements which are then authorized for storage and use by any particular custodian of that data.
- data redundancy - The practice of keeping data in two or more places within a database or data storage system.
- dynamic schema - A changeable representation of data.
- schema - A representation of a plan or theory in the form of an outline or model.
- open source - Denoting software for which the original source code is made freely available and may be redistributed and modified.
- data journey - The data journey represents the key stages of the data process. The journey is not necessarily linear; it is intended to represent the different steps and activities that could be undertaken to produce meaningful information from data.
- compare digest - A method of comparing two arrays of bits. This function uses an approach designed to prevent timing analysis by avoiding content-based short circuiting behaviour. This makes it appropriate for cryptography.
- ndarray - A data structure defined by the NumPy Python library. It is an object written in C and implemented in Python. It offers various functionality but the important point is that it is very fast and can be processed very efficiently using the in-built functions of the library.
- meta data - Data added to a dataset to provide additional information. It is information about information.
- data lake - A centralized repository designed to store, process, and secure large amounts of structured, semi-structured, and unstructured data.

D Jupyter Notebook

The entire testing notebook used to demonstrate the functionality of the pipeline. Please see Figures 8, 9, 10, 11, 12 and 13.

```
In [6]: ## Sign in and Authentication
username = 'test_user_main'
password = 'some_password'
email = 'email@test_email.com'
full_name = 'Tomasz Neska'
path = 'http://18.200.229.75:8080/'
file_name = "test.json"
# declaration of variables used

In [7]: # initialisation cell
import sys
sys.path.insert(0, "/home/splitsky/Desktop/code_repositories/resdata/")
import interface as ui
api = ui.API_interface(path)
api.check_connection()
#purge call - comment out if necessary -> just for development
#api.purge_everything()

Out[7]: True

In [8]: # make user and populate the database with data
result = api.create_user(username_in=username, password_in=password,email=email,full_name=full_name)
print("User created "  + str(result))

User created True
```

Fig. 8. Cell 1: The definition of the variables. Cell 2: Initialisation cell which declares the objects used. Cell 3: User creation.

```
In [8]: # populate the database
no_of_rings = 5
no_of_experiments = 2

import testing as t
api.generate_token(username, password)
project_name = "project_test_"
experiment_name = "experiment_test"
ds_size = 2000
for i in range(0,4,1):
    t.generate_optics_project(file_name,[no_of_rings, no_of_experiments], project_name+str(i),
                              experiment_name, author_name=username, size_of_dataset=ds_size)
    project = t.load_file_project(file_name)
    api.insert_project(project)

# generate special project for analysis showcase
project_name = "special_project"
experiment_name = "special_experiment_test"
ds_size = 5000

search_value = 15.65
t.generate_optics_project_2(file_name, [1,1], project_name, experiment_name, username, ds_size, search_value)
project = t.load_file_project(file_name)
api.insert_project(project)

print("Database populated")
print("")
api.tree_print()
```

Fig. 9. Cell 4: Populates the database.

E Full calculation

This appendix contains the complete calculation conducted within the report. Those numbers are listed here just for completeness and to provide evidence for claims made within Section 3.3.1. The tables 6, 7 and 8 contain all the calculated values used for fitting functions.

Data fitted	χ^2	a	b
Semester 1	0.809	0.59 $\pm$0.39	0.0014 $\pm$1.40 $\times 10^{-10}$
Semester 2 with cache	1.199	0.857 $\pm$0.029	0.00102 $\pm$1.00 $\times 10^{-11}$
Semester 2 with cache+multi-thread	1.104	0.852 $\pm$0.057	0.00102 $\pm$2.00$\times 10^{-10}$

Table 6. Fitting variables obtained. Equation fitted is $a + bx$. Fitting is a linear regression with errors estimated using multiple runs of testing.

```
In [9]: # Authenticate user cell
        api.generate_token(username, password)

In [10]: # Tree print cell
         api.tree_print()

         The data tree:

In [10]: # Pull individual data set by name and plot
         dataset = api.return_full_dataset("project_test_3","experiment_test 0", "PL spectrum - ring no.0")
         x, y, ey = dataset.data
         ### plot etc...

         import matplotlib.pyplot as plt
         plt.plot(x,y,"r+")

Out[10]: [<matplotlib.lines.Line2D at 0x111daa580>]
```

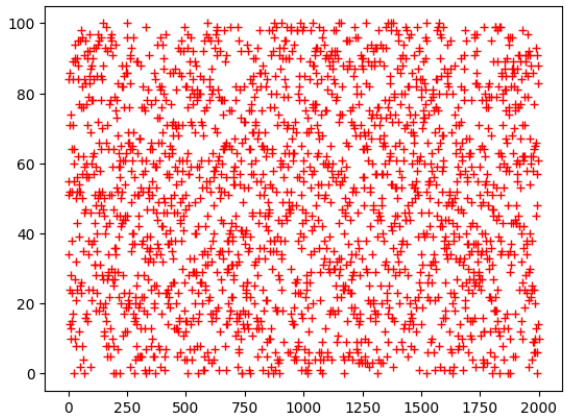

Fig. 10. Cell 5: Token generation. Cell 6: Tree print - data visualisation. Cell 7: Pulls individual dataset and plots the data.

```
In [11]: # Pull single ring -> plot dimensions and all spectra
         import sys
         sys.path.insert(0, "/home/splitsky/Desktop/code_repositories/resdata/")
         import interface as ui
         import jupyter_driver as dr

         api = ui.API_interface(path)
         api.check_connection()
         api.generate_token(username, password)
         api.tree_print()
         ring_id = 0
         experiment_id = "experiment_test 1"
         project_id = "project_test_0"
         datasets = api.experiment_search_meta(meta_search={"ring_id": ring_id}, experiment_id=experiment_id, project_id=project_id)
         for entry in datasets:
             print(entry.name)
         for entry in projects:
             dr.plot_from_dataset(entry, entry.data_type, entry.name)

                 -->Lasing spectrum - ring no.0
         ring_no. 0
         PL spectrum - ring no.0
         TRPL spectrum - ring no.0
         Lasing spectrum - ring no.0
         ring diameter : 0.9637907737377375
         quality : 1
         pitch : 0.18965777228893765
         threshold : 0.6673953433290012
         Printing 2D spectrum with error bars
```

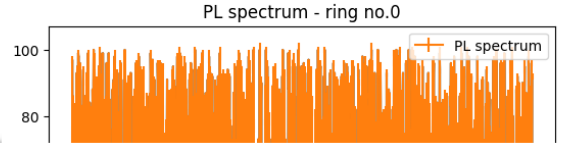

Fig. 11. Cell 8: Pulls single ring and summarises variables and plots spectra.

```
In [12]: # Pull all rings from experiment and summarise dimensions
import jupyter_driver as dr
# pull full experiment
experiment = api.return_full_experiment(project_name="project_test_0" ,experiment_name="experiment_test 0")
# separate dimensions
datasets = experiment.children
desired_data = []
for ds in datasets:
    if ds.data_type == "dimensions":
        desired_data.append(ds)

# summarise the quantities
dr.summarise_dimensions(desired_data)
```

```
Average values for dimensions:
ring diameter : 0.7422914029193357 +/- 0.3187223925063638
quality : 6.4 +/- 3.3823069050575523
pitch : 0.4082606383008508 +/- 0.2555024488839914
threshold : 0.6063285670392031 +/- 0.28511286866429963
```

```
In [13]: # Create a group and pull it together to summarise ring dimensions

group_name = "very_important_things"
group_permission = "write" # or read is recognised
# project_id, author_name, author_permission, group_name
# the function prints true or false depending whether it succeded on adding the group to the project/experiment/dataset
# _rec versions of the function add to groups recursively

print(api.add_group_to_project_rec("project_test_0", username, group_permission, group_name))
print(api.add_group_to_experiment_rec("project_test_1", "experiment_test 0", username, group_permission, group_name))
print(api.add_group_to_dataset_rec(group_permission, username, group_name, "project_test_2", "experiment_test 1", "TRPL spec
```

```
True
True
True
```

Fig. 12. Cell 9: Pulls all ring datasets connected by the same ring id. Cell 10: Creates a group and adds a dataset, an experiment, a project.

```
In [14]: # Pull group together to summarise TRPL spectra from the datasets

# use function which can be used with the group name or author name
structure = api.author_query(group_name)

api.tree_print_group(group_name)
```

```
project_test_0
    -> experiment_test 1
        --> ring_no. 0
        --> PL spectrum - ring no.0
        --> TRPL spectrum - ring no.0
        --> Lasing spectrum - ring no.0
        --> ring_no. 1
        --> PL spectrum - ring no.1
        --> TRPL spectrum - ring no.1
        --> Lasing spectrum - ring no.1
        --> ring_no. 2
        --> PL spectrum - ring no.2
        --> TRPL spectrum - ring no.2
        --> Lasing spectrum - ring no.2
        --> ring_no. 3
        --> PL spectrum - ring no.3
        --> TRPL spectrum - ring no.3
        --> Lasing spectrum - ring no.3
        --> ring_no. 4
        --> PL spectrum - ring no.4
        --> TRPL spectrum - ring no.4
        --> Lasing spectrum - ring no.4
    -> experiment_test 0
        --> ring_no. 0
        --> PL spectrum - ring no.0
        --> TRPL spectrum - ring no.0
        --> Lasing spectrum - ring no.0
        --> ring_no. 1
        --> PL spectrum - ring no.1
        --> TRPL spectrum - ring no.1
        --> Lasing spectrum - ring no.1
        --> ring_no. 2
        --> PL spectrum - ring no.2
        --> TRPL spectrum - ring no.2
        --> Lasing spectrum - ring no.2
        --> ring_no. 3
        --> PL spectrum - ring no.3
        --> TRPL spectrum - ring no.3
        --> Lasing spectrum - ring no.3
        --> ring_no. 4
```

Fig. 13. Cell 11: Pulls the nodes connected within a group.

Data fitted	fit type	χ^2	a	b	c	d
Semester 1	cubic	0.938	1.00 $\pm$0.14	0.948 $\pm$0.014	0.00499 $\pm$0.0001	0.000456 $\pm 5.50 \times 10^{-8}$
Semester 2	cubic	1.875	1.28 $\pm$0.16	0.432 $\pm$0.015	0.0351 $\pm 9.80 \times 10^{-5}$	-0.000138 $\pm 5.00 \times 10^{-8}$
Semester 2 with cache	linear	2.172	0.684 $\pm$0.022	0.315 $\pm 7.70 \times 10^{-5}$		
Semester 2 with cache	linear	1.900	0.584 $\pm$0.016	0.322 $\pm 5.60 \times 10^{-5}$		

Table 7. Fitting variables obtained. Equation fitted is $a + bx + cx^2 + dx^3$. Errors are obtained from the standard deviation of the same test runs for given values of N.

Data fitted	χ^2	a	b
Semester 1	1.571	-0.427 $\pm$0.62	1.88 $\pm$0.0089
Semester 2	1.063	-1.55 $\pm$0.63	4.68 $\pm$0.0076
Semester 2 with cache	0.994	1.8 $\pm$0.066	0.439 $\pm$0.0008
Semester 2 with cache + multi-thread	1.272	0.609 $\pm$0.61	2.04 $\pm$0.0074

Table 8. Fitting variables obtained. Equation fitted is $a + bx$. Fitting is a linear regression with errors estimated using multiple runs of testing.