

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Electron Devices

Design of a Non-Relational Data Storage System for Edge Devices in Flow Chemistry Instruments

Individual Project Laboratory

Name: Md Rakin Sarder
Supervisor: Dr. Ferenc Ender
External Supervisor: Zoltan Nagy, SpinSplit LLC

Budapest, 2020

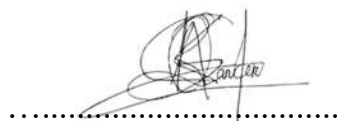


Project task description

Various hardware modules are incorporated in SpinSplit's flow chemistry ecosystem such as pumps, thermostats, valves, electrospinner units. Each of such devices are controlled by edge computing units, which are implementing the physical level communication with the actual hardware actuators and sensors. Once a task is executed at the end-point hardware, its operation should be always supervised at the edge level through the available sensor devices.

The goal of this individual project is to design a container-based data storage system for the supervision subsystem of the edge computing units of SpinSplit LLC, Hungary. The storage system is a non-relational database by requirement, which can deal with heterogeneous data. The project consisted of:

- Reviewing literatures related to non-relational databases and perform comparative analysis of some existing noSQL database systems.
- Choosing a suitable noSQL database system and justifying the choice by conducting performance analysis and suitability testing for real-time operation of the Edge Supervisor.
- Define the system requirements and build the block diagram of the integrated Edge supervisor database to the existing edge computing system.
- Designing algorithm for the database operations based on the system and operational data collected from the flow chemistry instruments and edge computing units.
- Implementing the designed algorithms for the database in python in an OOP style and compile a new python library called "eSupDB".
- Perform post-implementation performance analysis of the written code and the database.
- Participate in the deployment process of the database in the SpinSplit testing environment.



Md Rakin Sarder

Table of contents

1	Company Background.....	3
2	Project overview	3
3	Literature review	6
3.1	Real-time data processing	7
3.1.1	Heterogeneous Data in flow chemistry.....	7
3.1.2	Suitability of non-relational database in the Pharma 4.0 context	8
3.2	Edge Computing.....	9
3.3	Edge-level noSQL data storages in IoT distributed database system.....	11
3.3.1	Low resource and power consideration	11
3.3.2	Raspberry Pi Compute	12
3.4	Comparison of different noSQL storages in the context of IoT	13
4	Summary of the theoretical background	15
4.1	MongoDB Architecture	15
4.2	CRUD in MongoDB.....	16
4.3	Choosing MongoDB for the Edge Supervision project.....	16
5	Design considerations and design tools / Materials and methods.....	18
5.1	eSupDB Storage	19
5.2	eSupDB data insertion.....	22
5.3	eSupDB data query and command update	23
6	Implementation and results	25
6.1	Initial environment setup.....	25
6.2	Core library file	26
6.2.1	Class definition	26
6.2.2	Function implementations.....	27
6.3	Test framework setup	33
6.4	Setup configuration and library compilation.....	33
6.5	Results and discussions	34
7	Conclusions and future plans	38
8	Bibliography.....	39
9	List of Figures	41

1 Company Background

SpinSplit LLC is a Pharma 4.0 based company located in Budapest. They specialize on designing flow chemistry-based instruments for research and manufacturing purposes. One of the core foci for the company is to research and develop interconnected modular flow systems, which can provide higher degree of flexibility over the traditional batch chemistry systems. This approach also allows clients to increase their production reliability and speed while at the same time helps in the economic scaling process from laboratory to the manufacturing[1].

The core products of SpinSplit are “spFlow” flow chemistry and “eSpin” nanofiber technology systems, combined with their proprietary software “SpinStudio Flow”. Sitting at the top-level of SpinSplit’s ecosystem, “SpinStudio Flow” combines spFlow instruments to an integrated flow chemistry system. SpinSplit’s mission is to utilize their vast experience in producing high quality and advanced equipment and systems to fulfill the needs of laboratory-based research.

2 Project overview

SpinStudio integrates with various hardware modules such as pumps, thermostats, valves, electrospinner units through “edge computing units” of spFlow, which is a salient feature of the hardware ecosystem. The integration of the edge computing units of the system with SpinStudio and their communication is key to run and manage all the flow chemistry related operations. While SpinStudio itself is managing multiple edge computing units at a time, currently it is also handling the lower-level modules to perform different operations. The current stage of development of their ecosystem is to further modularize the handling of the low-level RMP (Reactor Main Panel) modules (Figure 1) by assigning the task fully to edge computing units, which are, by resource, capable of managing and monitoring the RMPs like SpinStudio as well.

This development of SpinStudio’s ecosystem will help the system:

- To optimize at the highest level (Level of SpinStudio)
- Utilize previously unused resources of the edge computing units for control and supervision tasks.
- Use less resources to run multi-channel communication between the hardware subsystems and the software subsystems.

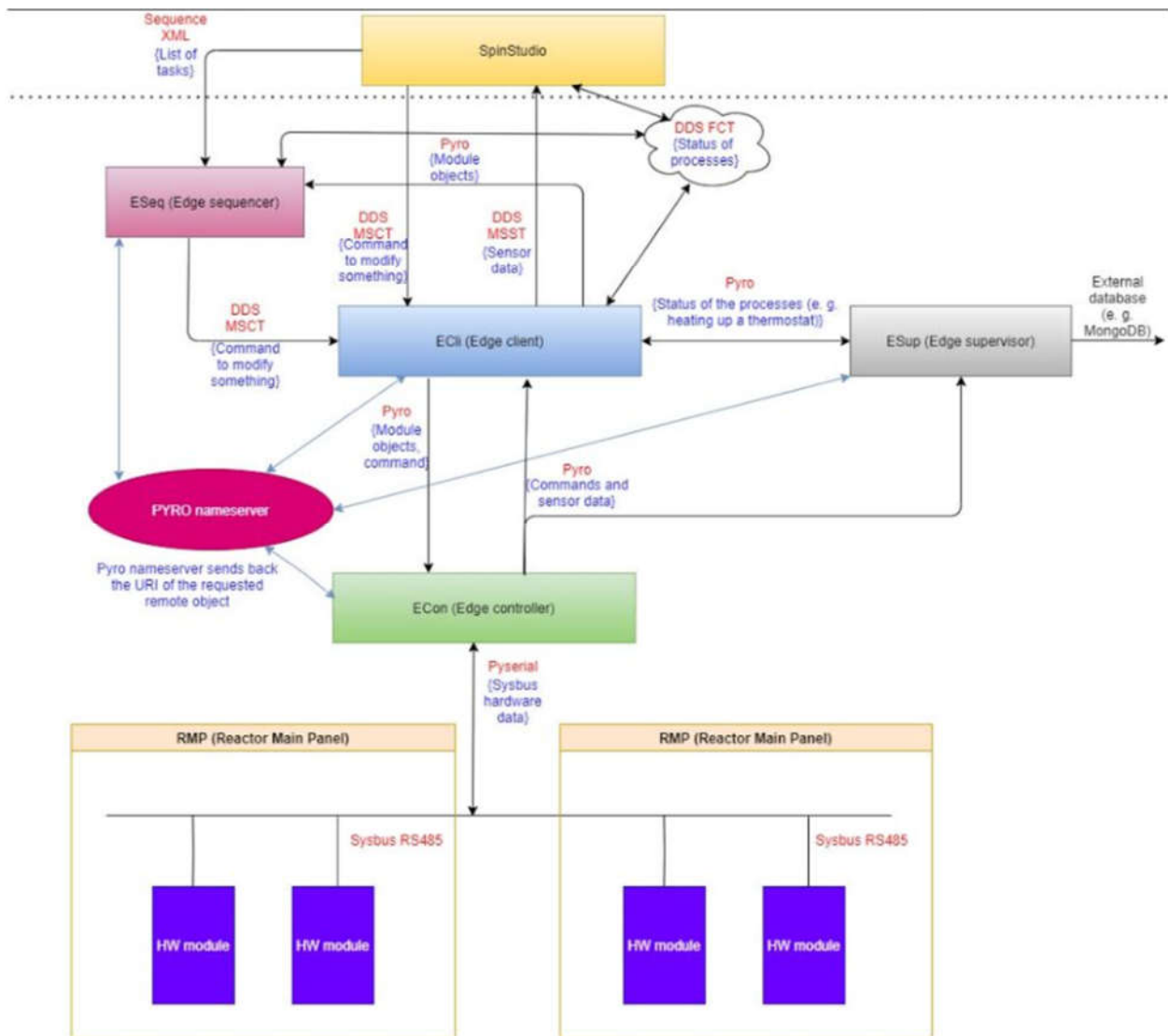


Figure 1 Edge Architecture of the spFlow ecosystem of SpinSplit[2]

“Edge Supervisor” module of the “Edge Client” (the edge computing units) is implementing the physical level communication with the actual hardware actuators and sensors on a supervision level. The four major design goals of the Edge supervisor to achieve success are:

- Store the data collected from different end-point modules.
- Act as a backup data storage for individual edge clients.
- Monitor the end-point modules once a task is initiated from the edge supervisor.
- Analyze the processes of the modules based on the collected real-time data and control the states of the end-point modules.

This project deals with the first two design goals of Edge Supervisor, the data accumulation and transaction. Figure 2 shows the goal of this project in a descriptive manner. As we can see that the data is essentially stored in the database in a non-relational way in the noSQL storage container. This

noSQL based operation allows the system to handle large volume on data without any previous knowledge about its structure. Also, since these local storage containers are isolated from the top-level hierarchy, they are essentially backing up the data inside the edge supervisor module, thus creating a multi-tiered database model with redundancy for all the subsystems.

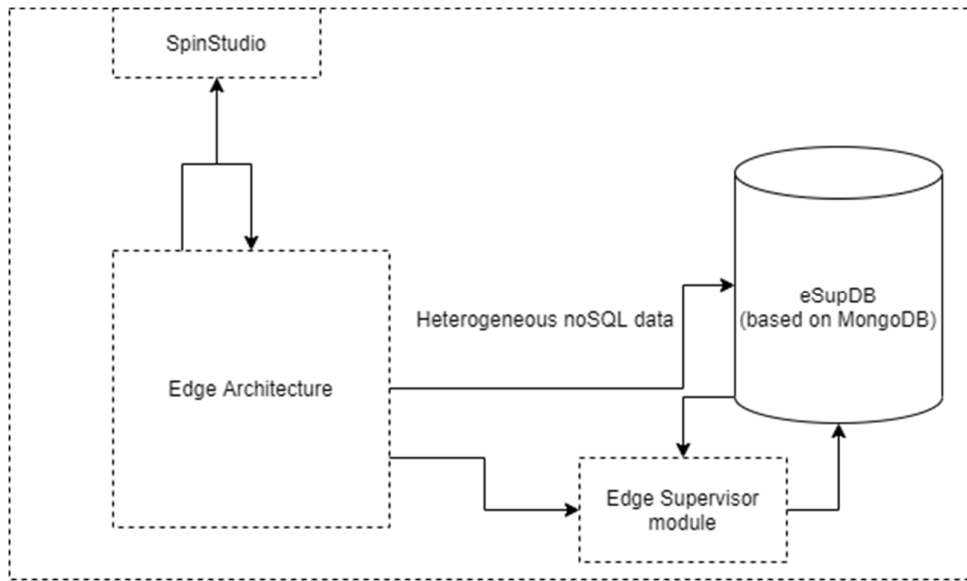


Figure 2 Non-relational data flow of the proposed database system. The database is local only to its edge host

The organization of this paper is as follows: Section 2 describes and discusses related works. Section 3 provides a short background review on NoSQL databases. Section 4 presents a qualitative comparison among the three selected databases. The methodology of the performance experiment is provided on Section 5, while Section 6 presents the results for the quantitative analysis. Section 7 critically discusses the obtained results, while Section 8 concludes the paper presenting directions for future work.

3 Literature review

Industry 4.0 is one of the most trending subjects among the scientists and manufacturers today because of the availability and advancement of smart technologies in the recent years. It consists of CyberPhysical Systems (CPS), Internet of Things (IoT) and cloud computing. The ultimate outcome of this creation are today's "Smart Factories"[3].

The adoption of Industry 4.0 (Pharma 4.0) into the Pharmaceutical industry has resulted significant progress in this industry. Because the industry had been facing some major challenges over the course of time[3]. The concepts of Pharma 4.0 have promising potential to solve these issues, and have already started to prove its potential in this sector[4]. Some of the major contributions of Pharma 4.0 in the pharmaceutical industry are (Figure 3):

- Stabilized and accelerated process development.
- Increase quality of processes via integration of different smart systems and analytics.
- Optimization of the maintenance of components and machineries
- Increased the compliancy rates of the manufactured products

Analytics enabled data processing and management is significant to optimize process modeling, estimate and predict measurement parameters and contribute to various decision making in the laboratories and factories.

IoT implementations depend on a framework which is distinguished by its heterogeneous technology stack in multi-layer networks. Also, IoT networks in Pharma 4.0 provide a variety of equipment, ranging from low-cost small devices with only little processing power to central cloud servers with plenty of resources. Millions of devices with integrated computing systems are expected to be connected to the Internet, producing a vast amount of data[5].



Figure 3 Concepts of Pharma 4.0 [5]

3.1 Real-time data processing

In flow chemistry, as the usage of Pharma 4.0 technologies are increasing, the amount of generated data is also increasing. This ongoing increase of produced real-time data is also being influenced by the following factors[6]:

- Continuous usage of connected devices, hardware drivers and components.
- Advancement of data storage and mining methods.
- Advancement in the computational power among single chip devices, embedded systems, and microprocessors.
- Robustness and reliability in data processing and storing mechanism in the edge.

As the number of influencing parameters for data production increases, the complexity and rigidity of data structures decreases[7].

3.1.1 Heterogeneous Data in flow chemistry

Data storages and processors must deal with high volume of non-uniform unstructured data, because of the variation of different sensors, actuators, and components, and due to the variation of flow process ecosystems. In other words, flow chemistry processes deal with heterogeneous data, with inherent ambiguity, variability, redundancy and accuracy[8].

Flow chemistry processes driven by IoT technology produces heterogeneous data, and suitable data management processes are used to handle them. An example of heterogeneous data generation can be traced to the spFlow instruments of SpinSplit[9]. At SpinSplit, a collection of spFlow modules is used to perform different flow chemistry and microfluidic experiments (Figure 4).

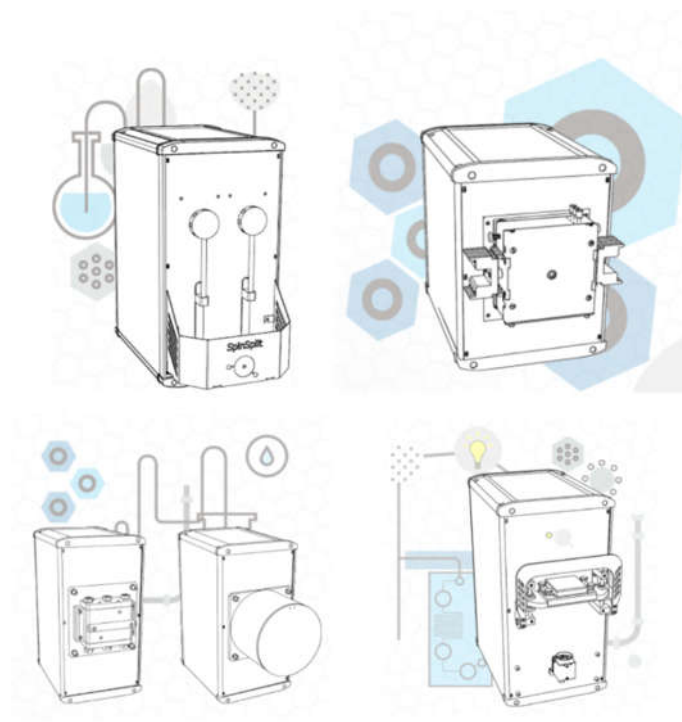


Figure 4 spFlow modules of SpinSplit [9]

These different modules have different internal and process parameters, which produces real-time sensor, control, and indicator data. However, the structure of collected and processed data for one instrument may not be the same for another instrument. This varies from operation to operation of the flow chemistry processes.

Another observation for heterogeneous data in this context is that since we are dealing with real-time data, there might be several missing values within the data set accumulated from different components at a certain time. For instance, in the context of sensors at any certain point of time they might become faulty or may accidentally skip in generating data. Also, an intrusion event may occur, which might compromise the data integrity[10]. This phenomenon of missing data in the context of IoT based sensors are discussed in [10-12].

3.1.2 Suitability of non-relational database in the Pharma 4.0 context

Traditionally, databases have been designed in a relational structure. Relational database systems have defined structure, rigid schema and stores data in tables (Figure 5)[13]. They are usually operated using SQL (Structured Query Language). Relational Database Management Systems, or RDBMS have gone through decades of development because of its longer existence in the computing sector. Some of the most popular RDBMS in the market are MySQL, Oracle, PostgreSQL, MariaDB, MS SQL Server.

The major advantages of RDBMS in the context of IoT in Pharma 4.0 are as follows[14]:

- RDBMS provides more stability and rigidity if the production or experimental data set is uniformed (data variation defined by ranges) and structured.
- Provides much better data integrity and isolation in case of non-distributed and low-volume data.[15]
- Data security is much higher than its counterpart because of years of research and development.

Non-relational databases on the other hand does not follow the tabular relations like relational databases. They are coined by the term “noSQL” (Not-only-SQL). The storage and retrieval are different than relational database, as non-relational databases do not follow any defined methodology to do that. Instead, the data classification method varies from noSQL-to-noSQL databases existing in today’s world (Figure 5). Some of the noSQL data classification methods existing today are[15]:

- Documents (such as Apache Cassandra)
- Key-value based (such as Berkeley DB)
- Wide column (such as Mongo DB)
- Graph (such as Arango DB)

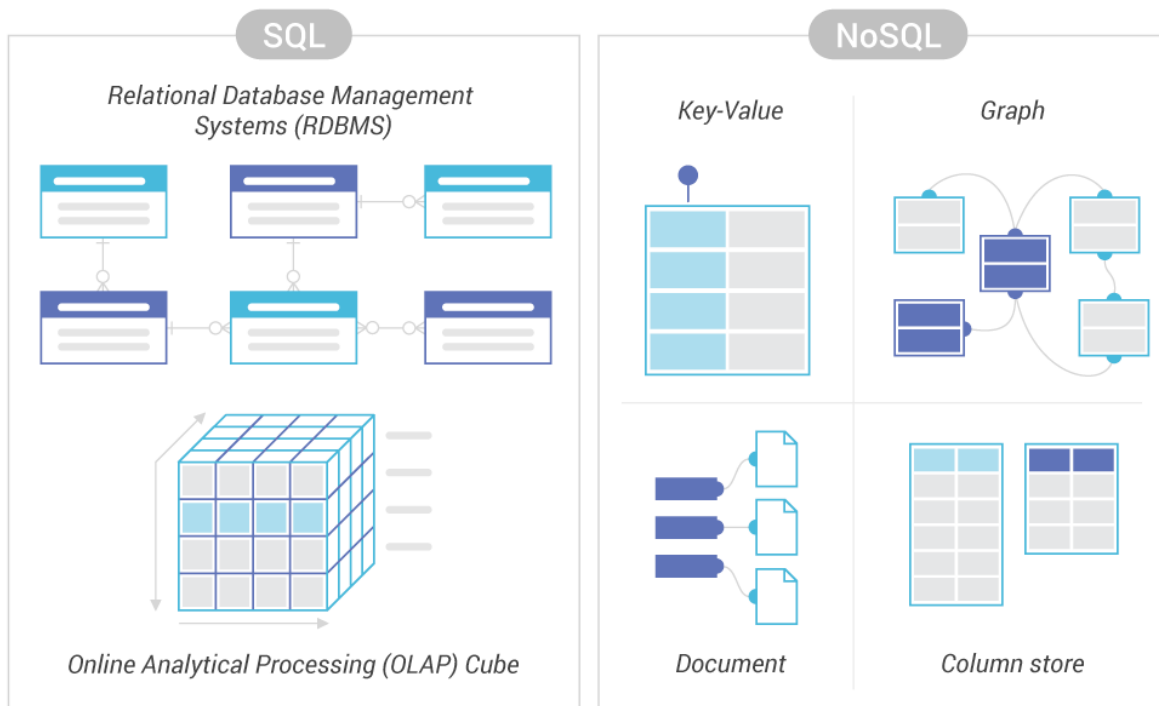


Figure 5 Architecture of RDBMS and noSQL[16]

The major advantages of non-relational databases in the context of Pharma 4.0 are:

- Provides much better scalability and response time when dealing with real-time data accumulated from multiple components in a distributed setup.
- Can deal with heterogeneous data with ease because prior knowledge about data structure is not necessary.
- Provides wide variety of approaches to the user when dealing with analyzing high-volume of data.

IoT and Pharma 4.0 applications can be designed using RDBMS, however as the data volume increases, the time and performance complexities of operations increase. RDBMS has also limited scalability feature for this context. They will eventually hit with constraint walls before they can scale in a project. NoSQL systems can perform much better in this aspect, not just for the initial part of a project, but for its lifecycle. Moreover, the advent of different modular technologies, data analytical and modelling algorithms has paved the path for its usage in IoT context[17].

3.2 Edge Computing

Edge computing is a decentralized and distributed computer architecture where data local to the system is processed by the local system. The salient decentralization feature allows edge computer to essentially operate remotely and empower IoT technologies.

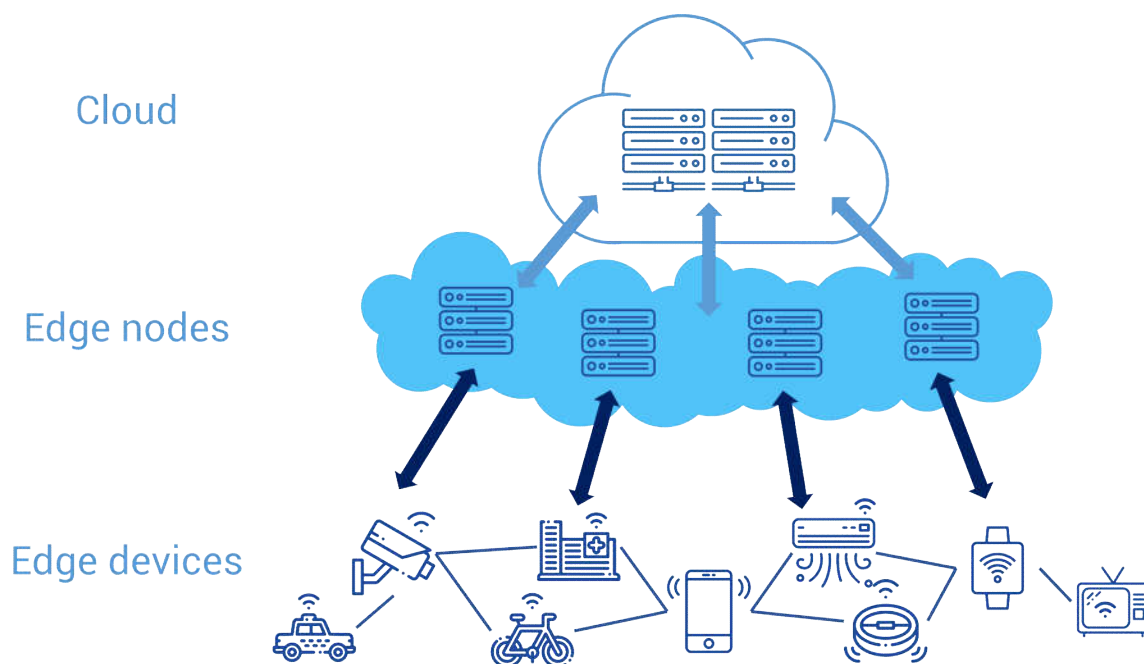


Figure 6 Basic Edge Computing Architecture [18]

A typical edge computing architecture consists of (Figure 6):

- Edge devices: It consists of all the connected devices, appliances, modules to the IoT system. The decision implementations and data collection processes are conducted at this level.
- Edge nodes: The middleware computing systems, that maintains the integration of all the connected devices. This is the main layer of edge computing, as the edge computer are distributed at this layer.
- Cloud/Master Node: The top-most level of the architecture. This node integrates all the edge computing units in a master-slave or producer-consumer approach (at SpinSplit this is SpinStudio).

Different communication protocols are applied to perform communication in this distributed and decentralized structure of edge computing [19]. Edge computing technologies are being used in IoT based flow chemistry applications. Due to the integration of Pharma 4.0 in experimental and industrial flow chemistry processes new systems are implementing edge computing architecture inside their ecosystems, and they are validated by significant research currently ongoing[20, 21].

The major advantages of edge computing over cloud or centralized computing are:

- Huge network bandwidth optimization.
- Storage local to only the edge computing units.
- Have more independence regarding the low-level operations.
- Less resource requirement by the server/central system.
- Data is accumulated, processed, and analyzed locally inside the edge computing units, thus allowing the central node to focus on managing only the edge nodes instead of directly managing the edge devices themselves. This gives more flexibility and optimization to the central/cloud node.

3.3 Edge-level noSQL data storages in IoT distributed database system

Now that the significance of noSQL based database systems in the context of Pharma 4.0 (more specifically, for IoT applications) have been established, the major challenge comes regarding the placement of database into the system. Essentially, for systems like flow chemistry, which consists of a multiple number of edge devices integrated to a high-level system (such as SpinStudio), a single database system comprising the top-level system and its subsystems can be a choice. However, another choice can be to create a distributed database system, where the databases are located at the edge clients as well as the topmost level (master nodes)[22]. A visual representation of the concept is shown in Figure 7.

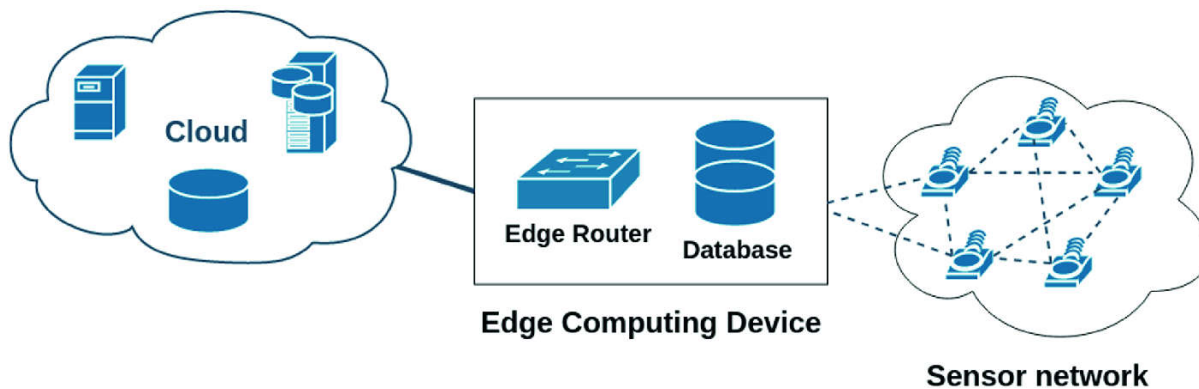


Figure 7 Edge level database placement in a distributed system[23]

NoSQL databases suitable for IoT based flow chemistry systems can be implemented on different cloud-based storage systems[24]. However, if the middleware subsystems are residing isolated from wide-area network (do not have direct access to internet), this implementation will not work[25]. In the context of flow chemistry systems which might have several layers of middleware, providing internet access to the edge systems may not be feasible at all. Again, the edge devices can have complete independence from the top-level system for some specific operations, and only communicate (receive commands/send back results) via a master-slave or a publisher-subscriber approach[26]. Therefore, to implement a distributed database system design among the edge systems, a local database infrastructure design is required. In other words, each middleware nodes need to have local database systems, which will be isolated from the top-level system, and will essentially serve the operations of that particular edge device (Figure 7) [27].

3.3.1 Low resource and power consideration

When constructing a database system for an edge computing system, it is essential to consider the computational resources and capacity of that edge device[28]. Single-board computers are used as the edge computing nodes in an edge architecture. These units have limited resources, and they are configured for different specific tasks. Therefore, when choosing any database system, either relational or non-relational, pre-construction analysis must be done regarding the suitability of it in that edge computing system. Table 1 shows the specification ranges of currently prevalent single-board computers in the market, which are used by different researchers and industries for edge architectural design. While designing the table, focus was given only on the widely used computers.

Table 1 Technical specification ranges of popular edge computers in the market[29-32]

Specification	Capacity
Cores	1-4
Clock speed	200 Mhz-4x2.1Ghz
Architecture	32/64 bit
RAM	32MB – 8GB
Power input	4.5 to 20 Vdc
Power consumption	0.5W – 4.0 W

3.3.2 Raspberry Pi Compute

Inside the edge architecture of spFlow and SpinStudio, Raspberry Pi compute modules are used. It is a System on a Module (SoM) chip that contains a processor, memory, and an integrated power circuit. They are widely used in the production level to design custom systems with the capability to control the form factor of the device, unlike the conventional Raspberry Pi boards[33]. Table 2 shows the technical specifications of Raspberry Pi compute. The designed edge supervisor was based on this compute module.

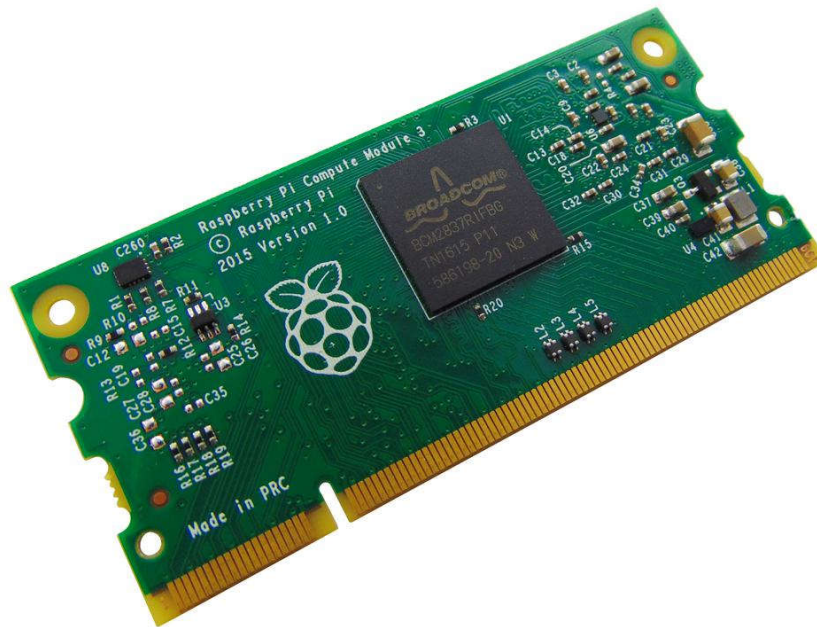


Figure 8 Raspberry pi compute module [33]

Table 2 Raspberry Pi compute module specification[34]

Specification	Capacity
Cores	2
Clock speed	1.2Ghz
Architecture	Cortex-A53 64 bit
RAM	1GB
Storage	8-32 GB flash

3.4 Comparison of different noSQL storages in the context of IoT

When choosing a suitable database system based on the application, the data management requirements must be known[35]. In the context of edge computing in systems such as flow chemistry, the requirement analysis is essential for the system's internal performance, as well as the lifecycle of the solution. According to [35], the following key points (the points relevant to this project) was analyzed:

- Heterogeneous property of data
- Semantic interoperability for meaning data exchange
- Scalability of the database
- Fast and real time processing of temporal data
- Database security
- Data aggregation to provide fused information

5 noSQL database solutions suitable for IoT based edge computing devices have been chosen for the analysis. The selection was also done based on the capacity of these database solutions to satisfy the requirements mentioned above. The details of the feature analysis are mentioned in Table 3. In the table, we can see that the solutions have been further analyzed in the context of the current "Edge supervision" project. All these solutions are either open source, or commercially licensable. Python compatibility was also checked since the programming was done in Python and Raspberry Pi has extensive support for it.

Table 3 Comparison of 5 noSQL databases suitable for edge computing in IoT

Name	Requirements/dependencies (if any)	ACID (Atomicity, Consistency, Isolation, Durability)	Existing python library	Debian compatibility	Near “realtime” data operation (Data storing and transactional speed)	Database source type (open or paid)	Miscellaneous features
CodernityDB	Python (no 3 rd party dependency)	High ACID compliance	Yes (built for python)	Yes	100 000 insert and more than 100 000 get operations per second	Open source (Apache 2.0)	- Schema less - Sharding functionality
Unqlite	- C (it’s based on C) - 256MB free RAM	High ACID compliance	Yes (but it is a wrapper for the C library, Cython required)	Yes	<u>In mem:</u> store: 901.9 ms, search: 2586 ms <u>In Local disk:</u> store: 15223.3 ms, search: 114.2 ms	Open source (2-Clause BSD license)	Zero configuration, cross platform, self-contained
TinyDB	Python (no 3 rd party dependency) - Extremely lightweight	Poor ACID compliance	Yes (built using python)	Yes	Near real-time data operation (BUT ONLY CAN HANDLE SMALL DATA SETS)	Open source	Zero configuration, cross platform, self-contained
MongoDB	-gnupg (a Debian library)	Average ACID compliance	Yes	Yes	Not near real-time operation (speed & performance like RBMS)	GNU AGPLv3	-
UpscaleDB	C/C++	High ACID compliance	Yes	Yes	Real-time data operation	Apache 2.0 Open source	- Vert fast I/O, integrated compression, high scalability and high-level encryption

4 Summary of the theoretical background

Mongo DB was chosen as the database system for the edge supervision project. Mongo DB is a cross platform database system based on document-collection approach. It creates documents inside collections with optional schemas.

4.1 MongoDB Architecture

A visual representation of the architecture inside a mongodb database is given in figure 9. At its core, MongoDB functions on BSON (Binary JSON), which is an extended version of JSON. It stores data as BSON objects, or in mongodb term, as “Documents”. In simple terms, the document model of MongoDB maps the BSON objects to the application code. MongoDB has the following component types:

- **Database:** The top-most level of the mongodb tree. It contains one or more collections that holds the documents.
- **Collections:** A collection in a mongodb database is a group of mongodb documents. This feature has similarity to a RDBMS structure, however there is no structural constraints and is completely schema-less.
- **Documents:** Documents are the records in a collection. They are stored in key-value pairs as lists or arrays, or sometimes as nested documents (multi-level documents).
- **Field:** Fields are the key-value pairs in a document. They can be of any data type. A document may have zero or more fields. Fields can be corresponded to columns in a RDBMS table.

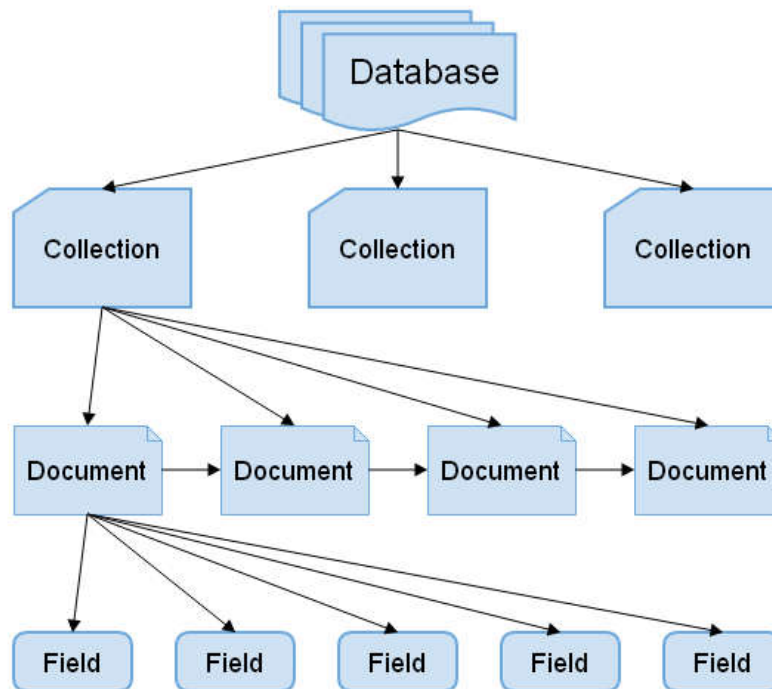


Figure 9 Architecture of a mongodb database [36]

4.2 CRUD in MongoDB

CRUD operations (Create, Read, Update, Delete) in mongodb deals with documents. CRUD operations of a persistent data storage system provide developers a framework for constructing a complete data model for a particular application[37]. The details of CRUD operations in mongodb are given below:

- **Create operation:** We use create operation to add or insert a new document in a collection. If the collection does not exist, mongodb will automatically create the instructed collection. The following methods are used. Write operations in mongodb are atomic in nature, i.e., they target a single collection.

```
db.collection.insertOne()  
db.collection.insertMany()
```

- **Read operation:** We use read operation to retrieve a document in a collection based on a query. The query operation takes “filters” and “limits” for the returned result. The following primary methods are available.

```
db.collection.find()  
db.collection.findOne()
```

- **Update operation:** We use update operation to retrieve one or more document in a collection based on a query and update them. Like create, Update operations in mongodb are atomic in nature, i.e., they target a single collection. The query operation takes “filters” to search for records to be updated, similar to read. The following primary methods are available:

```
db.collection.updateOne()  
db.collection.updateMany()  
db.collection.replaceOne()
```

- **Delete operation:** We use delete operation to delete one or more document in a collection based on a query (in case of selective deletion). Like create, Delete operations in mongodb are atomic in nature, i.e., they target a single collection. The query operation takes “filters” to search for records to be deleted, similar to read. The following primary methods are available:

```
db.collection.deleteOne()  
db.collection.deleteMany()
```

4.3 Choosing MongoDB for the Edge Supervision project

The primary reason for choosing mongo DB over the other noSQL solutions in the market is because of its active development lifecycle by its developers. Some of the noSQL database systems which has been studied for this project do not provide active support or development lifecycle. On top of that, apart from Mongo DB the database systems did not perform satisfactorily during the initial configuration and setup stage in the Debian buster environment. Also, some solutions, such as UpscaleDB does not have active python support, and has gcc and g++ dependencies to compile and make the driver files inside the debian system. Mongo DB driver on the other hand did not require any additional dependencies to compile.

Considering some other factors, the following are the complete list of reasons for choosing Mongo DB:

- Active development lifecycle and support by the developers.
- Horizontal scaling
- Native Sharding
- Lightweight and easily traversable
- Easy to configure and setup in Debian buster.
- Mongo DB supports “on-premises infrastructure”, which was an essential requirement for the Edge architecture of SpinStudio and spFlow.
- Flexible document-based modelling with indexing
- Ad-hoc query implementation
- Real-time data aggregation

5 Design considerations tools and methods

Table 4 represents the technical requirements with the specifications of the proposed solution.

Table 4 Project requirements and the corresponding specification

Requirement	Specification
Edge device	Raspberry Pi Compute
System OS	Raspberry Pi Debian Buster
Linux Kernel	4.19
Database system	Mongo DB Community
Programming language	Python 3.8
Python library dependencies	Pymongo, Mongo-queue-service, datetime, json

The database system design of eSupDB consists of two primary sections (Figure 10):

- Storage:** Stores the records of SpinStudio “sessions” and the corresponding tasks of them. Storage sections essentially stores the real-time logged time series data by the “Edge client” (refer to Figure 1) and organize them as “Tasks” under different “Sessions”. Each “Task” has one or more active/finished “commands”. Detailed discussion regarding tasks will be discussed in the following section.
- Command event queue:** Command event queue is a priority queue-type data structure that adds new “commands” passed to the database in a FIFO format. The priority queue essentially creates a “job list” for the Edge Supervisor to look at the active “commands” in operation at a certain time. This approach helps in significantly reducing the time and resources that would have been required to search for active tasks through the whole storage system.

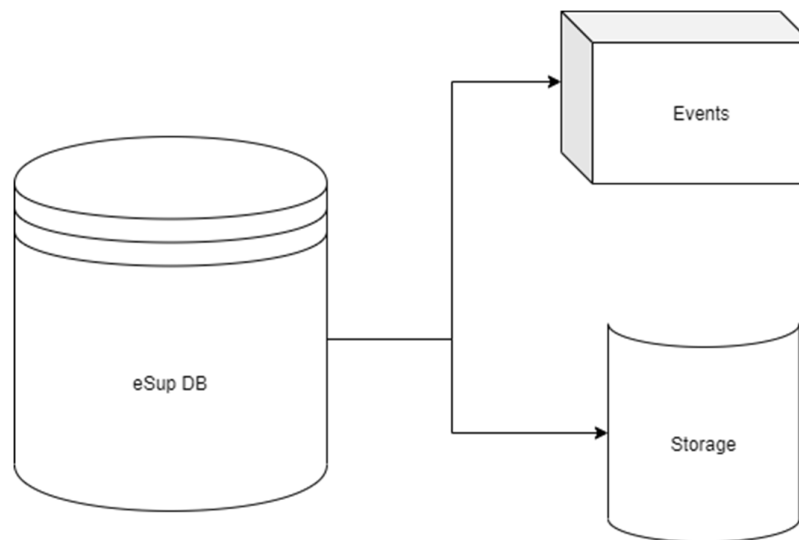


Figure 10 Top-level breakdown of eSupDB

5.1 eSupDB Storage

As mentioned earlier, storage section of eSupDB stores the records of various flow chemistry operations that are being handled by an edge node in the SpinStudio architecture. The breakdown of storage in eSupDB is shown in Figure 11. As we can see, the top-level nodes of storage are “Sessions”. Each session will have at least one or more “Tasks”. Similarly, each “Task” will have at least one “Commands” associated with it.

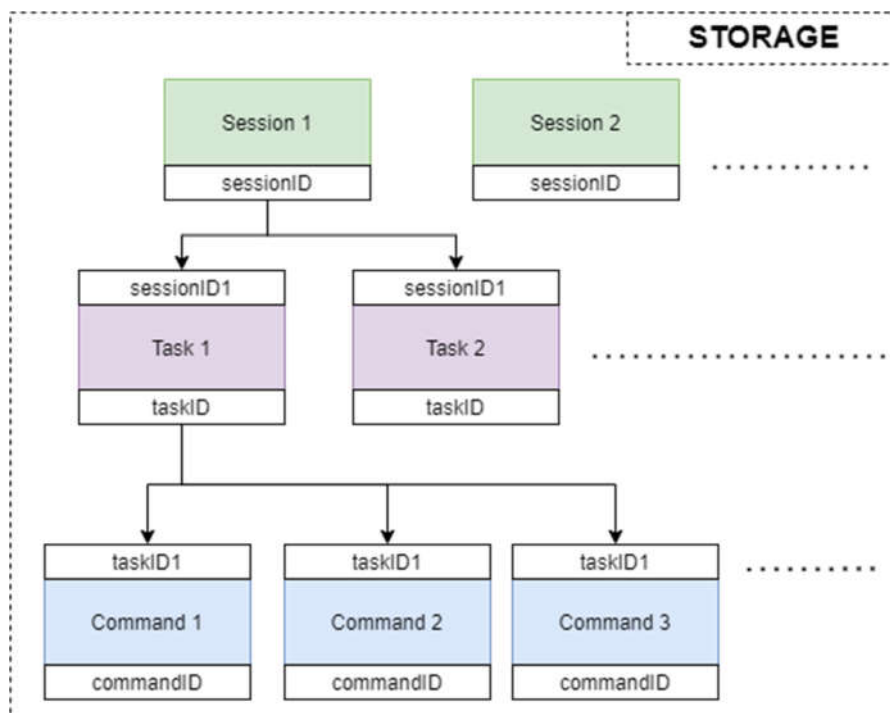


Figure 11 Storage Breakdown of eSupDB

- **Sessions:** The “Sessions” collection stores the sessions of SpinStudio which was initiated and executed by the user. It records the session data associated to the host edge node. A session may have one or more tasks associated with it. Each session is uniquely identified by its “sessionID” which was created in SpinStudio. The default traversing process for this collection is In-order traversal (top to bottom, in this case from a session to its tasks). The fields of a session document type are shown in Figure 12.

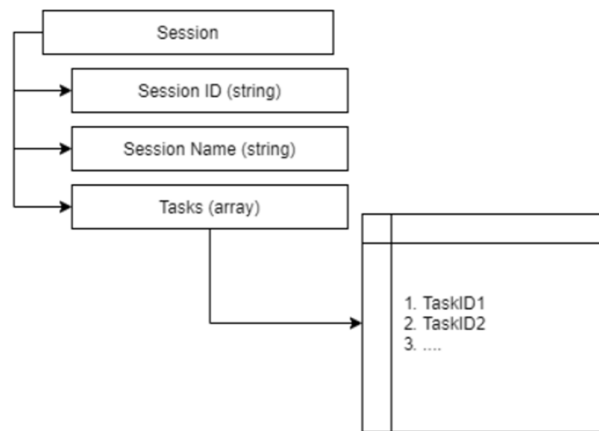


Figure 12 Fields of a session document

- Tasks:** The “Tasks” collection groups the commands produced by the user for a particular session. “Tasks” basically creates a hierarchy to “Commands”, because it groups a certain set of commands together for a running session. A task is uniquely identified by its taskID generated by the eSupDB and the parent sessionID. The reason to store the sessionID is to traverse a Task both In-order traversal (top to bottom, in this case from a task to its commands) and Post-order traversal (bottom to top, in this case from a task to its parent session). The fields of task document type are shown in Figure 13.

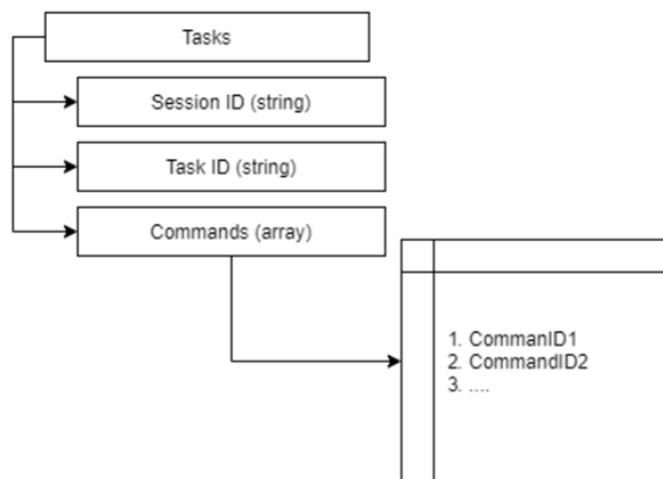


Figure 13 Fields of a task document

- Commands:** The “Commands” collection stores the commands produced by the user for a particular task. “Commands” are the bottom most level of eSupDB. A command is uniquely identified by its commandID generated by the eSupDB and the parent taskID. The reason to store the taskID is to traverse a command via Post-order traversal (bottom to top, in this case from a command to its parent task). Figure 14 shows the fields of commands document type. As we can see that commands includes complete details, including the control values, target

values and raw indicator data collected from different modules. The indicator data is the raw time series data of the database, and it is used by the supervisor to monitor modules and command progresses.

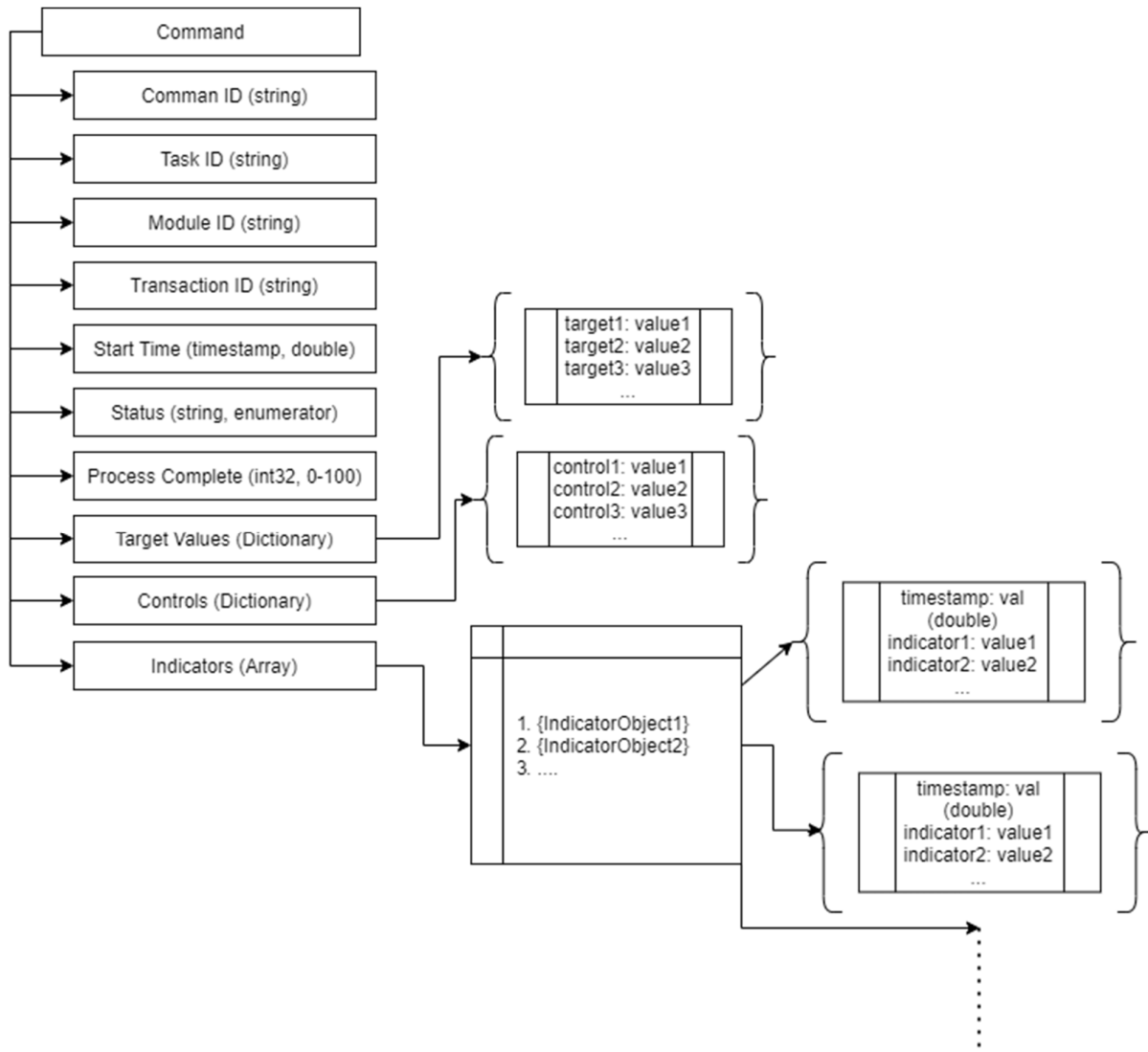


Figure 14 Fields of a command document

5.2 eSupDB data insertion

Session creation: Figure 15 show the process diagram of how the data is flown from the edge client to the eSupDB in case of a new session creation. Note that edge client is inserting into the database. During the creation of a session, no new events are created.

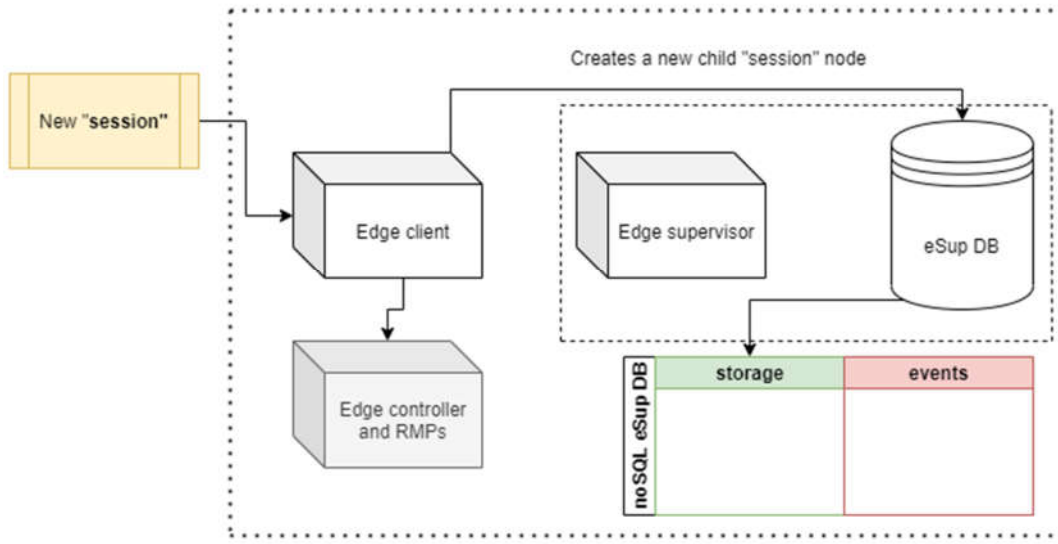


Figure 15 Process diagram of a new session-type document creation

Task and collection creation: Figure 16 show the process diagram of a new task for a session is created and inserted into eSupDB. The edge client posts the task and the command documents to the DB, the DB stores it into the corresponding collections and publishes an event to the event queue. Note that edge client is inserting into the database. Each command is created for a module, with a label "Running", which depicts as the active status of that command. In case of new commands for a task with taskID, new commands are inserted into the commands collection, while adding an event to the event queue at the same time.

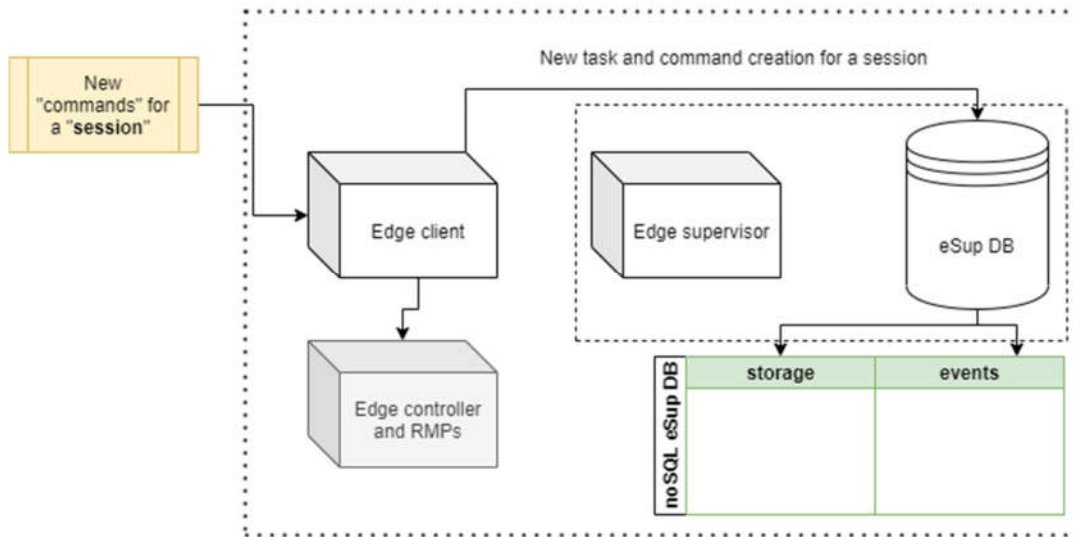


Figure 16 Process diagram of a new task-type and command-type document creation, and new command insertion for a particular task

New indicator record data insertion: Figure 17 show the process diagram of how commands are updated with the addition of new indicator records. Indicator data values of a command are recorded at a particular timestamp. The "indicator" data essentially acts as the logged raw time series data of eSupDB. During this operation, no new events are created in the event queue, as no new commands are being created by this operation.

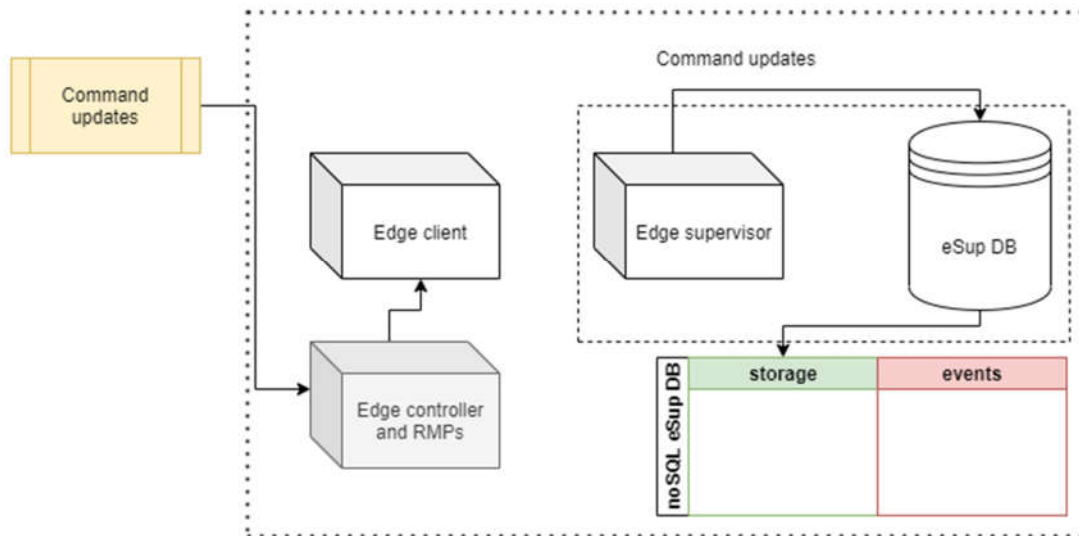


Figure 17 New indicator data updates for a command

5.3 eSupDB data query and command update

Active command "events" lookup: Figure 18 shows the process diagram of how the edge supervisor checks for active events in eSupDB. As already mentioned, command events are published in a separate collection in a queue format. Initial priorities are given to the command events published

first. However, users can also insert a “priority” event at a certain time if its priority exceeds the priority of the existing events.

A user collects an active command event by using “pop” operation to the queue. The payload of the pop operation contains the taskID and commandID associated with that command. Using these values, user gets the full event details from the storage. This mechanism helps the user and the system to spend less time and computational resource to search through the entire database.

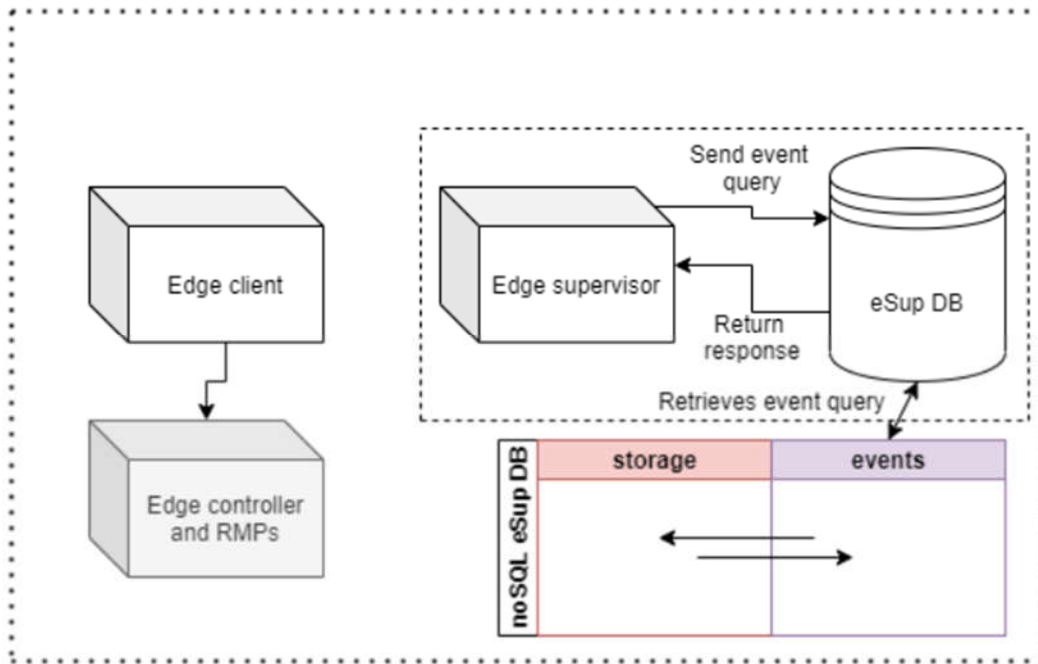


Figure 18 Process diagram to look up for active commands and retrieve its details

Updating command progress: Figure 19 shows the process diagram of how the edge supervisor updates a command in eSupDB. The edge supervisor monitors the progresses of different commands at the module level. Based on that it updates the progresses of commands in the database. During the update process, the user will have the taskID and commandID which was initially collected from the command event queue.

Once the user sends an instruction to update a command, eSupDB checks for the existence of the commandID in its collection and updates it. If the process completion value reaches 100%, eSupDB automatically labels the command as “Finished”, marking it as completed.

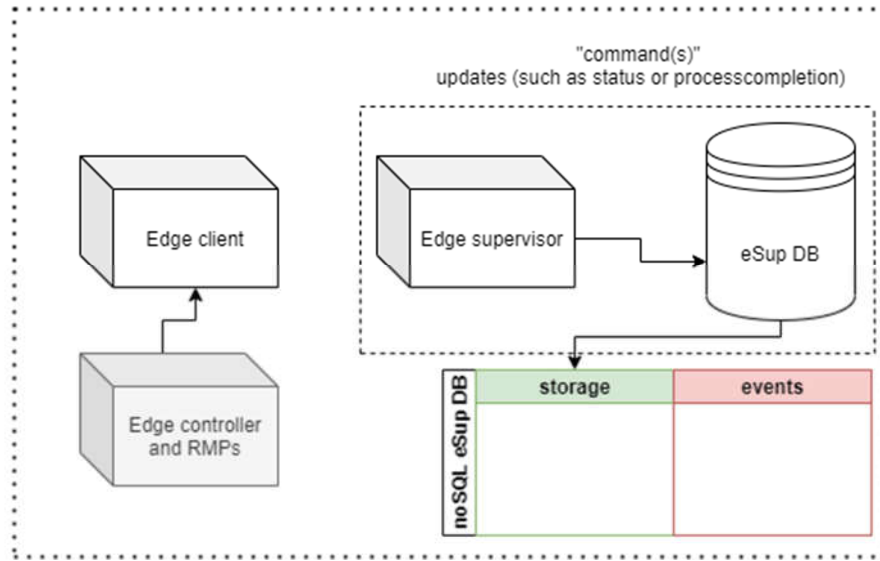


Figure 19 Process diagram to update commands

6 Implementation and results

The eSupDb was implemented in Python. In the previous section the discussion regarding the requirements of the database was mentioned. Several methods of implementation were initially considered for the development of the library, such as: API, cloud deployment, python library and as a simple python class file. However, the database was designed as a python library, which possesses several benefits for the lifecycle and SpinStudio ecosystem. The benefits are:

- Scalability
- Portability
- Easy deployment
- Bug tracking and fixing

6.1 Initial environment setup

Before the development process of the eSupDB python library, mongo driver for Linux OS was installed. A python environment was initially setup by creating a virtual environment. Creation of the virtual environment was important to encapsulate the project with only the required resources, thus the library will not cause any dependency issue in the future. After the virtual environment was activated, the required python libraries were installed for the project:

- wheel
- setuptools
- twine
- pytest
- pytest-runner

The library project folder was organized in the following way (Figure 20). Note that the “build “and “dist” folders were not initially created, they were created during the packaging stage of the library.

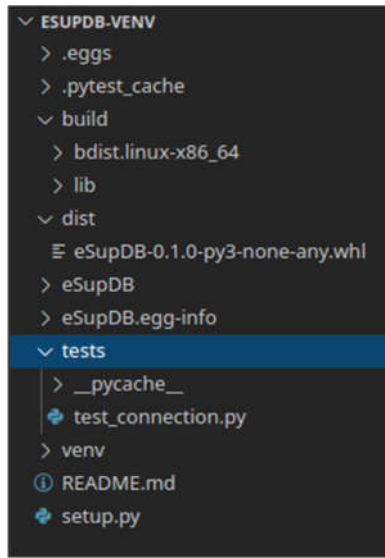


Figure 20 Environment setup and python file structure

6.2 Core library file

The core library source file was created in the “eSupDB” folder, the folder consists of two files, “__init__.py” and “connector.py” (Figure 21). “__init__.py” is an empty file that is required to mark the directory as a python package directory. The main code of eSupDB resides inside the connector.py file.

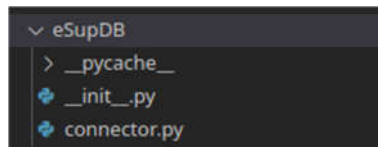


Figure 21 Project core folder

6.2.1 Class definition

The connector.py file consists of the following class declaration corresponding to eSupDB. There are two private variables for the class: client (A MongoClient instance) and eventQueue (A Mongo Queue instance). The default localhost configuration was used for the development (host: localhost, port: 27017).

```
class eSupDBClient:
    def __init__(self):
        self.__client=MongoClient()
        self.__eventQueue =
Queue(MongoClient().eSupDB.commandEvents,
consumer_id="edge_supervisor", timeout=300, max_attempts=3)
```

The class contains the following public methods:

```
def postData(self, *args)
def getEvent(self)
def getEventDetails(self, **kwargs)
def updateCommand(self, **kwargs)
```

The class contains the following private methods:

```
def __updateSessionCollection(self, **kwargs)
def __updateTasksCollection(self, **kwargs)
def __updateCommandsCollection(self, **kwargs)
def __postNewEvent(self, **kwargs)
```

6.2.2 Function implementations

updateSessionCollection: The updateSessionCollection function creates new session documents in the “sessions” collection. It has the following parameters:

- sessionID
- sessionName
- taskID

The function at first checks whether the passed sessionID exists in the database or not. If not, then it creates a new python list like with the provided data and inserts the data:

```
t=sessions.find_one(filter={"_id":kwargs['sessionID']})
if(t==None):
    data={
        '_id':kwargs['sessionID'],
        'sessionName':kwargs['sessionName'],
        'tasks':[kwargs['taskID']]
    }
    x=sessions.insert_one(data)
```

If the session exists in the database, then the taskID passed is pushed into the ‘tasks’ array of that session. The “addToSet” operator was used to skip duplicate insertion into the array.

```
sessions.update_one({'_id':kwargs['sessionID']}, {'$addToSet':{'tasks':kwargs['taskID']}})
```

updateTaskCollection: The updateTaskCollection function creates new task documents in the

“tasks” collection. It has the following parameters:

- sessionID
- commandID
- taskID

The function at first checks whether the passed taskID exists in the database or not. If not, then it creates a new python list like with the provided data and inserts the data:

```
t=tasks.find_one(filter={'_id':kwargs['taskID']})
if(t==None):
    data={
        '_id':kwargs['taskID'],
        'sessionID':kwargs['sessionID'],
        'commands':[kwargs['commandID']]
    }
    x=tasks.insert_one(data)
```

If the task exists in the database, then the commandID passed is pushed into the ‘commands’ array of that task. The “addToSet” operator was used to skip duplicate insertion into the array.

```
tasks.update_one({'_id':kwargs['taskID']},{'$addToSet':{'commands':kwargs['commandID']}})
```

updateCommandCollection: The updateCommandCollection function creates new command documents in the “commands” collection. It has the following parameters:

- command_priority (optional)
- commandID
- taskID
- commandData (a dictionary object)
 - moduleID
 - startTime
 - transactionID
 - target_values (a dictionary object)
 - controls (a dictionary object)
 - indicators (a dictionary object, optional)
 - processComplete (optional)
 - status (optional)

The function at first checks whether the passed commandID exists in the database or not. If not, then it creates a new python list like with the provided data. Additionally, it checks for the optional parameters mentioned above. If they exist, then the data list with the key value is appended, else a default value is assigned to them. This approach essentially validates the heterogeneous property of the database.

```
data['startTime']=cmd_data.pop('startTime', None)
```

```

data['moduleID']=kwargs['moduleID']
data['transactionID']=kwargs['transactionID']
data['target_values']=cmd_data.pop('target_values', None)
data['controls']=cmd_data.pop('controls', None)
#OPTIONAL FIELDS
if('status' in cmd_data):
    data['status']=cmd_data.pop('status', None)
data['startTime']=cmd_data.pop('startTime', None)
data['moduleID']=kwargs['moduleID']
data['transactionID']=kwargs['transactionID']
data['target_values']=cmd_data.pop('target_values', None)
data['controls']=cmd_data.pop('controls', None)
#OPTIONAL FIELDS
if('status' in cmd_data):
    data['status']=cmd_data.pop('status', None)
else:
    data['status']='RUNNING'
if('processComplete' in cmd_data):
    data['processComplete']=cmd_data.pop('processComplete', None)
else:
    data['processComplete']='0'
if('indicators' in cmd_data): #timestamp from main func
    data['indicators']=[cmd_data.pop('indicators', None)]
else:
    data['indicators']=[]

x=commands.insert_one(data)

data['startTime']=cmd_data.pop('startTime', None)
data['moduleID']=kwargs['moduleID']
data['transactionID']=kwargs['transactionID']
data['target_values']=cmd_data.pop('target_values', None)
data['controls']=cmd_data.pop('controls', None)
#OPTIONAL FIELDS
if('status' in cmd_data):
    data['status']=cmd_data.pop('status', None)
else:
    data['status']='RUNNING'
if('processComplete' in cmd_data):
    data['processComplete']=cmd_data.pop('processComplete', None)
else:
    data['processComplete']='0'
if('indicators' in cmd_data): #timestamp from main func
    data['indicators']=[cmd_data.pop('indicators', None)]
else:
    data['indicators']=[]

x=commands.insert_one(data)

```

After this operation, an event is posted based on the existence of `command_priority` parameter. If the parameter was passed, then a priority value will be passed to the event queue. Else, no priority will be assigned to the queue.

```

if not(kwargs['command_priority']==''):

task_event=self.__postNewEvent(db=kwargs['db'],taskID=kwargs['task
ID'],commandID=x.inserted_id,command_priority=kwargs['command_prio
rity'])

    else:

```

```
task_event=self.__postNewEvent(db=kwargs['db'],taskID=kwargs['taskID'],commandID=x.inserted_id,command_priority=0)
```

If the command exists in the database, then several situations are considered:

Indicator data passed: This means that the user has requested to insert a new indicator data with a timestamp to be pushed into the indicators array. This is a feature for the Edge client.

Process completion and status data passed: This means that the user is also requesting to update the process completion data and status of that command in the database. This is a feature for the Edge supervisor.

postNewEvent: The postNewEvent function creates new event documents in the “commandEvents” collection (the queue of eSupDB). It has the following parameters:

- taskID (payload of the queue)
- commandID (payload of the queue)
- command_priority (priority queue parameter)

If the passed priority value is greater than 0, then the priority is assigned to the queue input, otherwise no priority is assigned, just the payload is passed.

```
if(command_priority>0):  
  
self.__eventQueue.put({'taskID':taskID,'commandID':commandID},priority=command_priority)  
else:  
self.__eventQueue.put({'taskID':taskID,'commandID':commandID})
```

postData: The postData is a public function (user-side function) that allows the user to post new data into eSupDB. It has the following required parameters:

- sessionID
- sessionName
- transactionID
- moduleID

It also consists the secondary set of parameters. Some of these parameters have been configured as optional by design to perform different task-specific operations.

- startTime (optional, if user does not provide then this function will generate it)
- target_values (optional, only required during the first command creation)
- controls (optional, only required during the first command creation)
- status (optional, because the library can update the status based on the process completion data)
- processComplete (optional, because during the indicator data insertion part by the edge client, the processComplete data is not required. It will only be required when the Edge supervisor is actively monitoring or supervising a module).

The postData function performs several internal operations to successfully execute an insertion

process. The algorithmic steps for this function are given below:

- Creates new taskID and commandID. In the case of first-time creation of a command, the user does not pass the taskID and commandID. ESupDB generates the taskID and commandID for a particular task and command. Also, in case of an existing task but a new command for it, the user will only pass the taskID associated to it. The system will generate the commandID for that command. This approach was implemented in the following code:

```
if ('taskID' in kwargs): #that means user wants to update
    taskID=kwargs['taskID']

commandID=taskID+'__cmd_'+str(commands.count_documents({})+1)
if ('commandID' in kwargs): #that means user wants to update
    commandID=kwargs['commandID']
```

- Insert/update the session document, followed by the insertion/update process of a task document

```
curSession=self.__updateSessionCollection(db=db,sessionID=sessionID,
sessionName=sessionName,taskID=taskID)

curTask=self.__updateTasksCollection(db=db,sessionID=sessionID,taskID=taskID,commandID=commandID)
```

- Insert the command document to the commands collection. However, several things need to be considered during the command insertion:
 - Create a new python dict which will consist of the aggregated command data.
 - Check whether the user has passed the startTime or not. If not, retrieve the current time stamp.
 - Check for the optional parameters.
 - Create a new timestamp for the indicator data and append to the indicator dict.

The final command document is then inserted via the following code:

```
curCommand=self.__updateCommandsCollection(db=db,moduleID=kwargs['moduleID'],commandID=commandID,taskID=taskID,transactionID=kwargs['transactionID'],cmd_data=cmd_data,command_priority=command_priority)
```

At the end of postData function, the taskID and commandID is returned to the user as its response.

getEvent: The getEvent is a public function which can be used by the user to retrieve the topmost element in the commandEvents queue. It will return taskID and commandID of the popped top element. Once the retrieval is done, the job is marked as completed. If no event exists in the queue, then it will return a null command to the user.

```
try:
    job=self.__eventQueue.next()
    payload=job.payload
```



```

        job.complete()

    return payload
except Exception as e:
    print('No command in queue')
    return 'command'

```

getEventDetails: The getEventDetails is a public function which can be used by the user to retrieve the detailed data of a command. The user needs to pass the commandID as the function parameter. The returned JSON value with their retrieved mapped keys is given below:

```

dataJson={
    'commandID':kwargs['commandID'],
    'sessionID':curSession['_id'],
    'sessionName':curSession['sessionName'],
    'moduleID':curCommand['moduleID'],
    'taskID':curTask['_id'],
    'transactionID':curCommand['transactionID'],
    'status':curCommand['status'],
    'processComplete':curCommand['processComplete'],
    'controls':curCommand['controls'],
    'target_values':curCommand['target_values'],
    'startTime':curCommand['startTime'],
    'indicators':curCommand['indicators']
}
# print(dataJson)
return dataJson

```

If the commandID does not exist in the database, then an error response is returned to the user.

updateCommand: The updateCommand public function updates the progress of a command in the database. It takes the commandID and processComplete data as its parameters. The function makes the following algorithmic approach:

- Checks whether the command exists or not.
- Tries to update the command progress, if any error occurs, then an exception is raised, and a failed response is returned to the user.
- If the process completion reaches 100, then the function automatically updates the status of the command from “Running” to “Finished”.

```

try:
    commands.update_one({'_id':curCommand['_id']},{'$set':{'processComplete':kwargs['processComplete']}})
    # return 'Update successful'
except Exception as e:
    print(e)
    return 'Failed to update command'

####
if(int(kwargs['processComplete']>=100)):
    try:

```

```

commands.update_one({'_id':curCommand['_id']},{'$set':{'status':'FINISHED'}})
    return 'Update completed. Status Finished'
except Exception as e:
    #print(e)
    return 'Failed to update status during status completion update'
return 'Update successful'

```

6.3 Test framework setup

The test framework for the project was setup using pytest. The Pytest framework makes it easy to write small yet complex tests for applications and library applications [38].

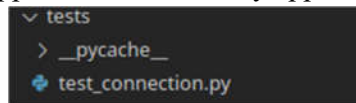


Figure 22 Test framework folder for the eSupDB python library

The testing of the eSupDBClient was done based on the following considerations:

- Assertion of the user accessible function
- Assertion of the returned responses
- Analyze the execution time

The following methods were tested:

- postData
- getEvent
- getEventDetails
- updateCommand

6.4 Setup configuration and library compilation

The setup configuration of the library was done using setuptools. Using setuptools, a new setup.py file was created, with the following configuration and an alias for pytest.

```

from setuptools import find_packages, setup
setup(
    name='eSupDB',
    packages=find_packages(include=['eSupDB']),
    version='0.1.0',
    description='Edge Supervisor DB, SpinSplit',
    author='Md Rakin Sarder',
    license='MIT',
    install_requires=['pymongo', 'mongo-queue-service'],
    setup_requires=['pytest-runner'],
    tests_require=['pytest'],
    test_suite='tests',
)

```

```
[aliases]
test=pytest
```

The setup.py was then executed to generate all the tests stored in the “tests” folder using the following command:

```
python setup.py test --addopts "-vv --durations=0"
```

Finally, the library was built using the following command, which generated “build” and “dist” folders. The “dist” folder contained the packaged library file.

```
python setup.py bdist_wheel
```

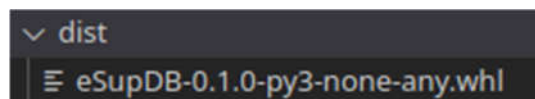
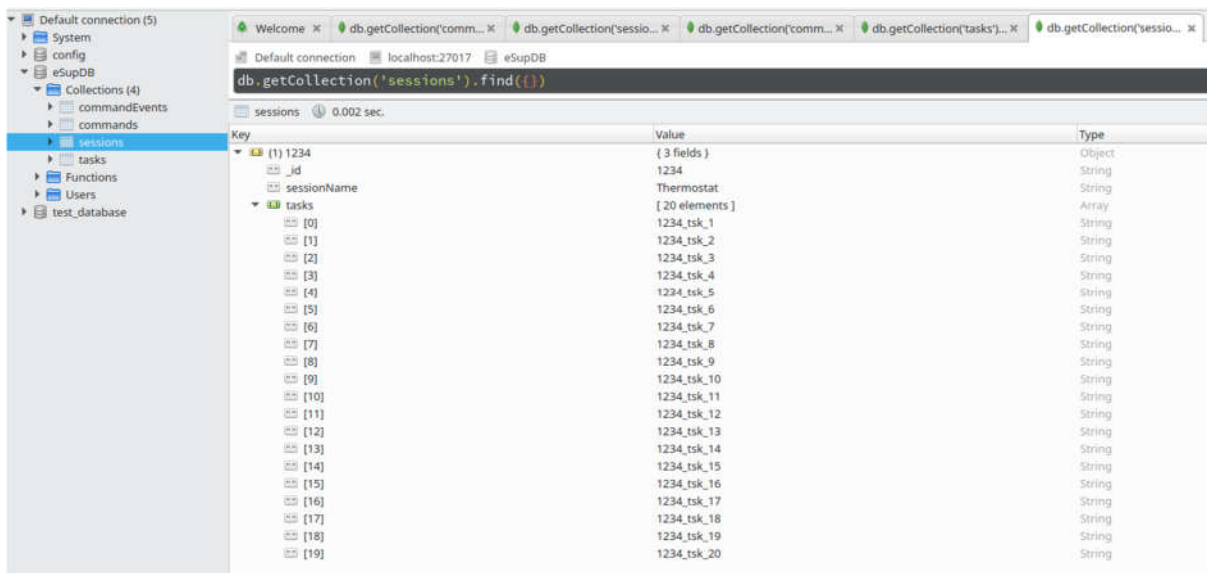


Figure 23 eSupDB packaged library

6.5 Test Results and discussions

6.5.1 Test responses the Insertion and Query of eSupDB

Test_insert: To test the insertion operation of eSupDB Robo3T was used. Robo 3T (formerly Robomongo) is an open-source MongoDB workbench for MongoDB 4.0 with a GUI to view and edit the database and an embedded mongo shell to perform query operations with other automation features. After the posting of session, task, and command documents, robo3T was used to verify the insertion. Figure 24-27 shows the structure of the database after several test insertion operations. We can see that one session document object was created (figure 24), which contains the “tasks” array that contains the task ids of all the tasks of that session.



Key	Value	Type
(1) 1234	{ 3 fields }	Object
_id	1234	String
sessionName	Thermostat	String
tasks	[20 elements]	Array
[0]	1234_tsk_1	String
[1]	1234_tsk_2	String
[2]	1234_tsk_3	String
[3]	1234_tsk_4	String
[4]	1234_tsk_5	String
[5]	1234_tsk_6	String
[6]	1234_tsk_7	String
[7]	1234_tsk_8	String
[8]	1234_tsk_9	String
[9]	1234_tsk_10	String
[10]	1234_tsk_11	String
[11]	1234_tsk_12	String
[12]	1234_tsk_13	String
[13]	1234_tsk_14	String
[14]	1234_tsk_15	String
[15]	1234_tsk_16	String
[16]	1234_tsk_17	String
[17]	1234_tsk_18	String
[18]	1234_tsk_19	String
[19]	1234_tsk_20	String

Figure 24 Structure of the sessions collection in eSupDB visualized by robo3T

In figure 25, we can see 20 new tasks were created for session with sessionID “1234”. Each of the documents contains a unique id (format: sessionID_tsk_XXX), the parent sessionID and the commands associated to that task (array type).

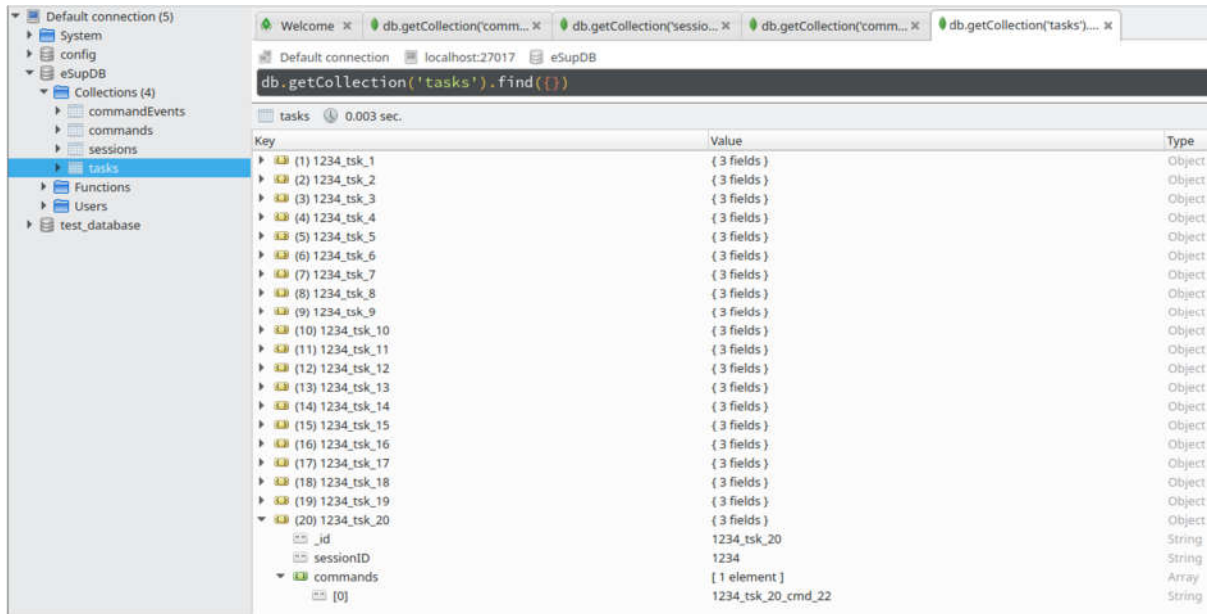


Figure 25 Structure of the tasks collections after several insertion operations

In figure 26, we can see all the commands that were created during the test operation. Each command document was created under only one task, which can be identified by the taskID field inside a command document. It also contains its own unique commandID, moduleID and other fields which were discussed in section 5.1.

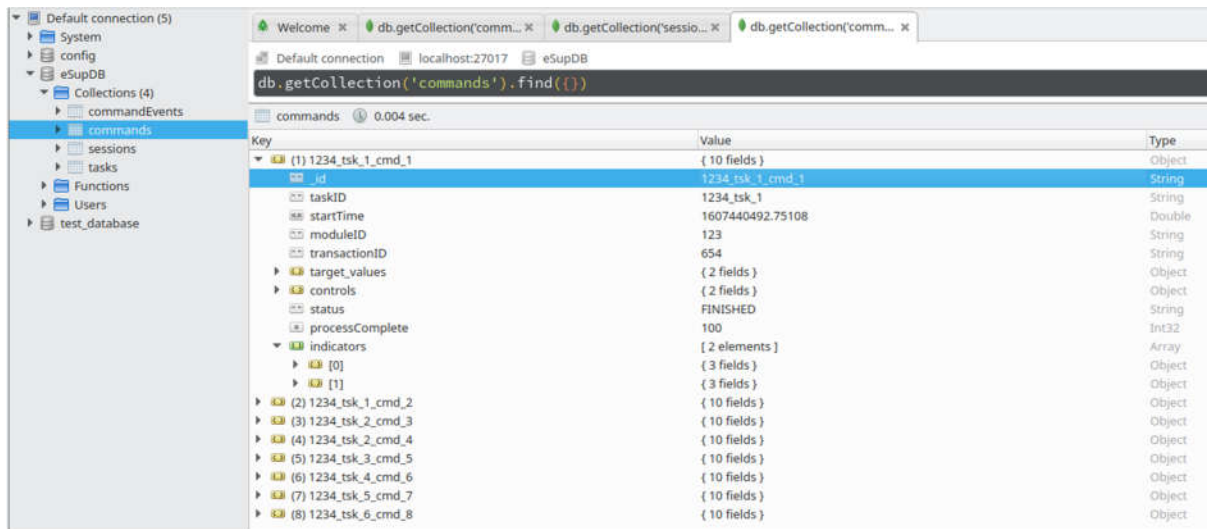


Figure 26 Structure of the eSupDB commands collection after several test insertion operations

Lastly, in figure 27, we can see the commandEvents queue of eSupDB. The queue, which was generated using mongo-queue service, are stored as documents, which contains a priority value of int32 type, and a payload dict which contains the taskID and commandID associated to that event document.

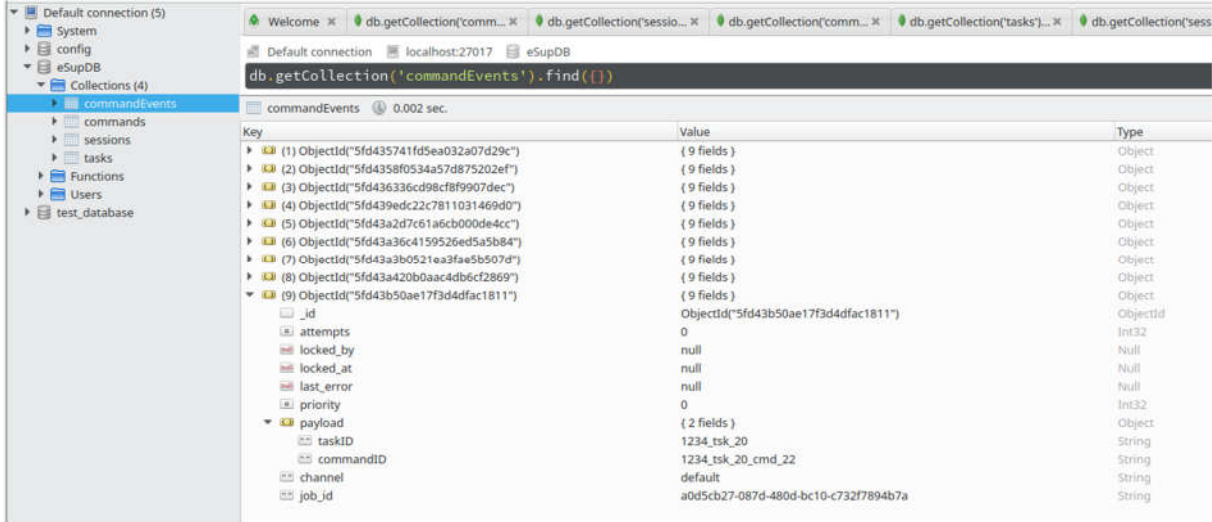


Figure 27 Structure of the commandEvent queue. Each event contains the taskID and commandID as the payloads

Test_get_event_details: To test the query operation of the library, we tested the get_event_details method. Figure 28 shows the returned response from the get_event_details operation. We can see that the returned response has a JSON format, consisting of key-value pairs of the requested data. We can also see the time series “indicator” field is an array type, which has subdocuments as its elements.

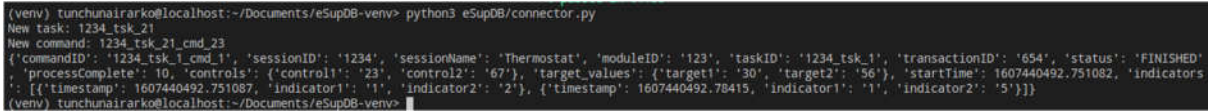


Figure 28 Returned response from the get_event_details operation

6.5.2 Analysis of the eSupDB library performance

Pytest analysis: The console test result during the pytest operation is given in Figure 29. As we can see from the console results that our timing analysis resulted successful pass operations for all the four functions tested. We can see that the assertion of the function access and responses did not return any error, which was marked by the “PASSED” keyword.

In the duration analysis part, we can see some slack times during the call operations of some functions. The “test_insert”, “test_get_event_details” and “test_update_command” functions call had the most significant call time (0.02s). This may be because of the payloads and the internal operations of the corresponding functions under testing. However, these slacks are not high enough to cause any significant impact to the end-user operations, which is proved from the final “PASSED” test result.

```

tests/test_connection.py::test_insert PASSED [ 25%]
tests/test_connection.py::test_get_event PASSED [ 50%]
tests/test_connection.py::test_get_event_details PASSED [ 75%]
tests/test_connection.py::test_update_command PASSED [100%]

===== slowest durations =====
0.02s call     tests/test_connection.py::test_insert
0.02s call     tests/test_connection.py::test_get_event_details
0.02s call     tests/test_connection.py::test_update_command
0.01s call     tests/test_connection.py::test_get_event
0.00s teardown tests/test_connection.py::test_update_command
0.00s setup    tests/test_connection.py::test_get_event
0.00s setup    tests/test_connection.py::test_get_event_details
0.00s teardown tests/test_connection.py::test_insert
0.00s teardown tests/test_connection.py::test_get_event
0.00s setup    tests/test_connection.py::test_update_command
0.00s setup    tests/test_connection.py::test_insert
0.00s teardown tests/test_connection.py::test_get_event_details
===== 4 passed in 0.13s =====

```

Figure 29 Unit test execution result of eSupDB

The summary of the result is given in Table 5.

Table 5 Unit test result summary

Test component	Result	Summary
User access assertion	Passed	All functions are accessible with the passed parameters.
Response assertion	Passed	The returned responses of the function showed expected output.
Function call time	0.01-0.02 seconds	The slowest call time value was not significant to affect the overall performance
Function setup time	0.00 seconds	No delays observed
Function teardown time	0.00 seconds	No delays observed
Overall execution time	0.13 seconds	The test result passed the test criterion successfully

7 Conclusions and future plans

A scalable and easily deployable noSQL database system for SpinSplit's edge architecture was designed in this project. Through this project, the two main goals of Edge supervision system were achieved. The designed database system was constructed considering the lifecycle and scalability of the internal edge architecture.

Also, the eSupDB library code was written considering the heterogeneity of the data. Taking arbitrary parameters as inputs, the DB can be scaled both vertically and horizontally without any prior knowledge about the structure of the data. This will allow future users to focus only on the usable application designs for the database, while they will need minimal efforts into implementing data models.

The future plan for this project may include an interface to visualize the noSQL database. The tree structure of different collections can be visualized, also the time series indicator data can be visualized using different visualization approaches. Another future plan that can be considered for this project is the optimization of the library code, in order to improve the timing performance. Also, other approaches for the noSQL functionalities of the database can be tested and implemented, to analyze their performance.

8 Bibliography

- [1] "Consortium - CPS4EU," ed, 2020.
- [2] S. LLC, "edge Architecture documentation," ed, 2020.
- [3] V. Steinwandter, D. Borchert, and C. Herwig, "Data science tools and applications on the way to Pharma 4.0," *Drug Discovery Today*, vol. 24, no. 9, pp. 1795-1805, 2019, doi: 10.1016/j.drudis.2019.06.005.
- [4] G. A. Van Norman, "Drugs, Devices, and the FDA: Part 1: An Overview of Approval Processes for Drugs," *JACC: Basic to Translational Science*, vol. 1, no. 3, pp. 170-179, 2016/04/01/ 2016, doi: <https://doi.org/10.1016/j.jacbts.2016.03.002>.
- [5] P. World, "Pharma 4.0: Industry 4.0 Applied to Pharmaceutical Manufacturing - Pharmaceutical Processing World," ed, 2019.
- [6] D. A. Pereira, W. Ourique de Moraes, and E. Pignaton de Freitas, "NoSQL real-time database performance comparison," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 33, no. 2, pp. 144-156, 2018/03/04 2018, doi: 10.1080/17445760.2017.1307367.
- [7] K. Adnan and R. Akbar, "Limitations of information extraction methods and techniques for heterogeneous unstructured big data," *International Journal of Engineering Business Management*, vol. 11, p. 184797901989077, 2019, doi: 10.1177/1847979019890771.
- [8] K. F. Jensen, "Flow chemistry—microreaction technology comes of age," *AIChE Journal*, vol. 63, no. 3, pp. 858-869, 2017.
- [9] SpinSplit, "SpinSplit spFlow Modular Instruments - SpinSplit," ed, 2020.
- [10] R. Zuech, T. M. Khoshgoftaar, and R. Wald, "Intrusion detection and big heterogeneous data: a survey," *Journal of Big Data*, vol. 2, no. 1, p. 3, 2015.
- [11] L. Li, B. Du, Y. Wang, L. Qin, and H. Tan, "Estimation of missing values in heterogeneous traffic data: Application of multimodal deep learning model," *Knowledge-Based Systems*, p. 105592, 2020.
- [12] Z. Liu, W. Zhang, S. Lin, and T. Q. Quek, "Heterogeneous sensor data fusion by deep multimodal encoding," *IEEE Journal of Selected Topics in Signal Processing*, vol. 11, no. 3, pp. 479-491, 2017.
- [13] E. F. Codd, "Relational database: a practical foundation for productivity," in *Readings in Artificial Intelligence and Databases*: Elsevier, 1989, pp. 60-68.
- [14] S. Rautmare and D. M. Bhalerao, "MySQL and NoSQL database comparison for IoT application," 2016: IEEE, doi: 10.1109/icaca.2016.7887957. [Online]. Available: <https://dx.doi.org/10.1109/icaca.2016.7887957>
- [15] V. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri, "Sqlvm: Performance isolation in multi-tenant relational database-as-a-service," 2013.
- [16] A. Turner, "ScyllaDB | NoSQL vs SQL," (in English), *ScyllaDB*, 2020/10/27/ 2020. [Online]. Available: <https://www.scylladb.com/resources/nosql-vs-sql>.
- [17] M. Asay, "Why You Need NoSQL For The Internet Of Things - ReadWrite," ed, 2014.
- [18] A. Cloud, "What Is Edge Computing? Advantages of Edge Computing - Alibaba Cloud Knowledge Base," ed, 2020.
- [19] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637-646, 2016.
- [20] I. Sittón-Candanedo, R. S. Alonso, J. M. Corchado, S. Rodríguez-González, and R. Casado-Vara, "A review of edge computing reference architectures and a new global edge proposal," *Future Generation Computer Systems*, vol. 99, pp. 278-294, 2019.
- [21] B. H. Song, B. Lee, K. T. Kim, and H. Y. Youn, "Enhanced query processing using weighted predicate tree in edge computing environment," in *2017 IEEE Conference on Standards for Communications and Networking (CSCN)*, 2017: IEEE, pp. 48-53.
- [22] P. Paethong, M. Sato, and M. Namiki, "Low-power distributed NoSQL database for IoT middleware," in *2016 Fifth ICT international student project conference (ICT-ISPC)*, 2016:

- IEEE, pp. 158-161.
- [23] P. Grzesik and D. Mrozek, "Comparative Analysis of Time Series Databases in the Context of Edge Computing for Low Power Sensor Networks," in *Computational Science – ICCS 2020*, Cham, V. V. Krzhizhanovskaya *et al.*, Eds., 2020// 2020: Springer International Publishing, pp. 371-383.
- [24] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645-1660, 2013/09/01/ 2013, doi: <https://doi.org/10.1016/j.future.2013.01.010>.
- [25] R. Cruz Huacarpuma, R. T. de Sousa Junior, M. T. De Holanda, R. de Oliveira Albuquerque, L. J. García Villalba, and T.-H. Kim, "Distributed data service for data management in internet of things middleware," *Sensors*, vol. 17, no. 5, p. 977, 2017.
- [26] P. Sethi and S. R. Sarangi, "Internet of things: architectures, protocols, and applications," *Journal of Electrical and Computer Engineering*, vol. 2017, 2017.
- [27] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for internet of things: a survey," *IEEE Internet of things journal*, vol. 3, no. 1, pp. 70-95, 2015.
- [28] B. Ahmed, B. Seghir, M. Al-Osta, and G. Abdelouahed, "Container Based Resource Management for Data Processing on IoT Gateways," *Procedia Computer Science*, vol. 155, pp. 234-241, 2019, doi: 10.1016/j.procs.2019.08.034.
- [29] T. R. P. Foundation, "Raspberry Pi 4 Model B specifications – Raspberry Pi," ed, 2020.
- [30] P. Galkin, L. Golovkina, and I. Klyuchnyk, "Analysis of Single-Board Computers for IoT and IIoT Solutions in Embedded Control Systems," 2018: IEEE, doi: 10.1109/infocommst.2018.8632069. [Online]. Available: <https://dx.doi.org/10.1109/infocommst.2018.8632069>
- [31] S. J. Johnston *et al.*, "Commodity single board computer clusters and their applications," *Future Generation Computer Systems*, vol. 89, pp. 201-212, 2018/12/01/ 2018, doi: <https://doi.org/10.1016/j.future.2018.06.048>.
- [32] TS-7260, "TS-7260," ed, 2020.
- [33] Element14, "Development Board, Raspberry Pi Compute Module 3 Lite, DDR2-SODIMM Mechanically Compatible SoM," ed, 2020.
- [34] T. R. P. Foundation, "Buy a Compute Module 3+ – Raspberry Pi," ed, 2020.
- [35] S. Amghar, S. Cherdal, and S. Mouline, "Which NoSQL database for IoT Applications?," 2018: IEEE, doi: 10.1109/mownet.2018.8428922. [Online]. Available: <https://dx.doi.org/10.1109/mownet.2018.8428922>
- [36] S. Interactive, "Getting to know MongoDB - Blog - Screen Interactive web studio (Kharkov, Ukraine)," ed, 2020.
- [37] C. Academy, "What is CRUD? | Codecademy," ed, 2020.
- [38] M. Müller, "Professional Testing with pytest and tox," ed, 2020.

9 List of Figures

Figure 1 Edge Architecture of the spFlow ecosystem of SpinSplit[2].....	4
Figure 2 Non-relational data flow of the proposed database system. The database is local only to its edge host	5
Figure 3 Concepts of Pharma 4.0 [5]	6
Figure 4 spFlow modules of SpinSplit [9].....	7
Figure 5 Architecture of RDBMS and noSQL[16]	9
Figure 6 Basic Edge Computing Architecture [18].....	10
Figure 7 Edge level database placement in a distributed system[23]	11
Figure 8 Raspberry pi compute module [33]	12
Figure 9 Architecture of a mongodb database [36].....	15
<i>Figure 10 Top-level breakdown of eSupDB</i>	<i>18</i>
Figure 11 Storage Breakdown of eSupDB.....	19
Figure 12 Fields of a session document	20
Figure 13 Fields of a task document	20
Figure 14 Fields of a command document.....	21
Figure 15 Process diagram of a new session-type document creation.....	22
Figure 16 Process diagram of a new task-type and command-type document creation, and new command insertion for a particular task.....	23
Figure 17 New indicator data updates for a command	23
Figure 18 Process diagram to look up for active commands and retrieve its details	24
Figure 19 Process diagram to update commands.....	25
Figure 20 Environment setup and python file structure.....	26
Figure 21 Project core folder	26
Figure 22 Test framework folder for the eSupDB python library.....	33
Figure 23 eSupDB packaged library	34
Figure 24 Structure of the sessions collection in eSupDB visualized by robo3T.....	34
Figure 25 Structure of the tasks collections after several insertion operations.....	35
Figure 26 Structure of the eSupDB commands collection after several test insertion operations.....	35
Figure 27 Structure of the commandEvent queue. Each event contains the taskID and commandID as the payloads	36
Figure 28 Returned response from the get_event_details operation.....	36
Figure 29 Unit test execution result of eSupDB	37