Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Analytics Dashboard for Duchenne Muscular Dystrophy Clinical Trials

Supervisor:
Dr. Aldo Faisal

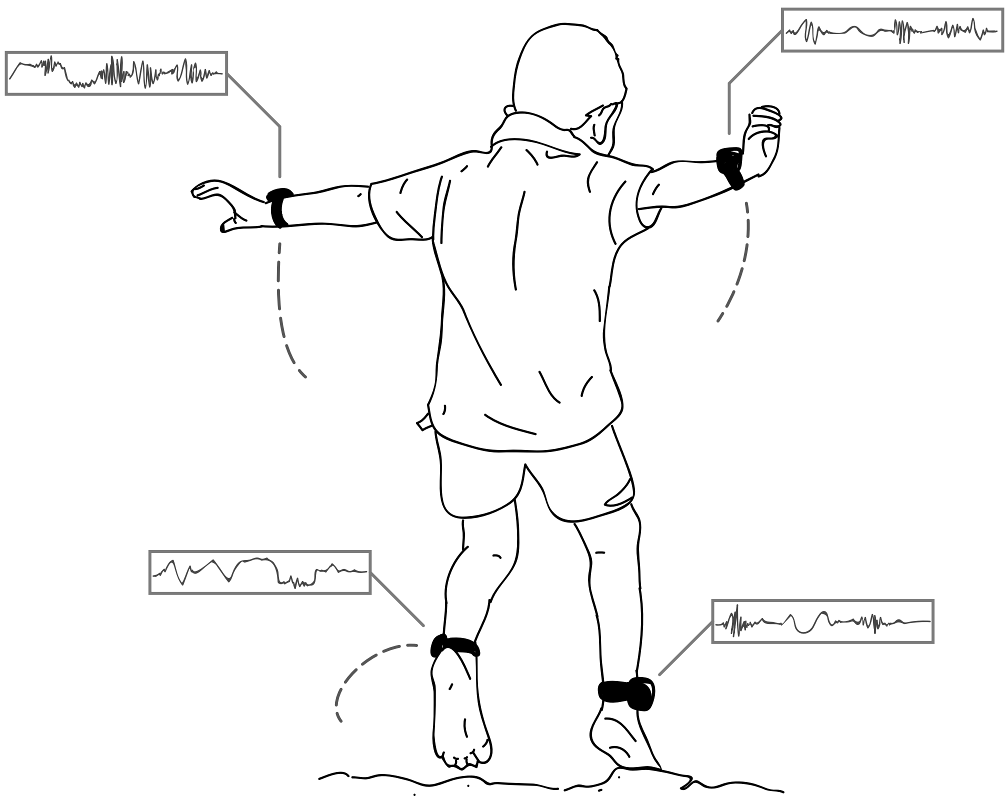
Author:
Tom Bellingham

Second Marker:
Dr. Ruth Misener

June 17, 2019

Abstract

The goal of the *KineDMD* study led by Dr. Aldo Faisal is to develop new motion-based digital biomarkers for quantifying the progression of a degenerative disease which affects mobility, Duchenne muscular dystrophy. As part of the study, a large volume of high-frequency motion sensor data is being collected from smart watches worn by young boys throughout their daily lives who are participating in an ongoing clinical trial. To remedy the lack of transparency in the data collection process, one of the products of this project is an interactive visual dashboard which provides clinicians and researchers with high-level day-to-day information about the data collection process. To progress towards the ultimate aim of extracting biomarkers from the data, the other deliverable of this project is an analytics engine with a programmatic interface which makes it easy to define, execute, and visualise analytics computed over the large and growing data set and to do so in an efficient way with limited computational resources.



Acknowledgements

Thank you, Dr. Aldo Faisal, for your suggestions, feedback, and encouragement throughout this project. Thank you for steering me towards software engineering rather than machine learning, for taking time to chat at a difficult moment in the year, and thank you for your jovial no-nonsense charisma that brightened the daily toil. It has been a delight to hear your name whispered in the corridors, with due gravitas, and to witness the slight wrinkles of apprehension appear on the foreheads of colleagues who await your penetrating insight with an appropriate tinge of fear. Nothing helps a fellow student prepare for a presentation better than the phrase "Aldo's coming!"

Thank you, Dr. Balasundaram 'Bala' Kadirvelu, for your inspiring perseverance in the midst of chaos; you exude calm and determination even when deadlines loom large, and you never complain even when you haven't had enough sleep (or coffee, or both). One week your kids were ill and you were up every night preparing for a conference and you would *still* check up on me and answer any questions I had. You are kind and compassionate, generous with your snacks and with your time, and I will think back on the our time working together with gladness. You have been a great ally and I hope your children will grow up to share many of your characteristics.

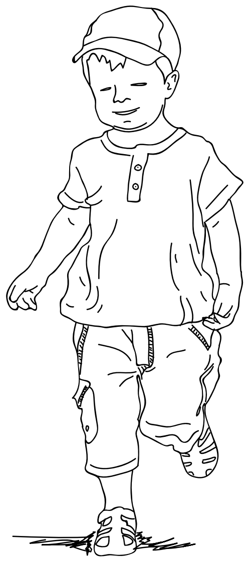
Thank you, flatmates, for making life at home a pleasure instead of a pain: Sam Lowe for your delicious meals, Arthur Niculae for banter about how hard life at Imperial is, and Samuel Tang for adding a musical twist to our midst.

Thank you, Dad, for helping me to stay sane by your wit and wisdom and for your many years living an honourable and inspiring life in the face of myriad challenges. Thank you Mom, Amy, and Josh for your reminders that there is much more to reality than whatever is bothering me on the day.

I applaud the boys and families participating in the clinical trials for their selfless willingness to endure the inconvenience of wearing devices for hundreds of hours to support the *KineDMD* research. They deal daily with burdens that I struggle to imagine, and this project would not have been possible without their noble commitment.

Finally, I am thankful to the creator of the universe. In case they are reading this and didn't yet know, I am immensely grateful for the opportunity to be alive in this world: to strive, to fail, to grow, to hurt, and to love. There is just enough joy available in rare moments of beauty to offset the struggles which so often characterises this brief life. Thank you.

Contents



1 Introduction

The holistic context of this project is that there are many diseases and disorders which impair bodily movements and which have a profound and negative impact on the experience of human beings. As such, there is an energetic global quest to develop cures and treatments for these conditions. To assess the effectiveness of a potential cure it is imperative to have precise and objective feedback about how the severity of a condition changes over time in different people, but this kind of clear feedback is hard to acquire when outward symptoms are subtle or only noticeable over long periods of time. Developing outcome measures to quantify changes in mobility is thus uniquely challenging: how the movements of an affected person differ from an otherwise healthy subject can be nuanced, they can be unique to individuals, and they can be easily obfuscated by factors unrelated to the severity of the disease in question. So, given the widespread need for good clinical measures and the difficulties in developing them, there is ample reason to search for novel techniques which outperform current clinical measures.

In a more specific context, Duchenne muscular dystrophy (DMD) is one such disease and it is the subject of the *KineDMD* study lead by Principle Investigator Dr. Aldo Faisal [1][2]. DMD is a degenerative neuromuscular condition caused by a genetic abnormality prevalent in young boys which leads to impaired mobility over time and a low life expectancy of 19-26 years [3]. The goal of the *KineDMD* study is to discover and develop new motion-based digital biomarkers which can be used to quantify the progression of the disease; those biomarkers will serve as outcome measures which improve upon the current clinical tests used to assess the severity of the condition, thus providing precise and personalised objective feedback which is available more quickly, frequently, and conveniently than existing physical examinations.

As part of the *KineDMD* project a previous Master's student, Shuai Zeng, developed and deployed a patient monitoring system which collects data from smart watches worn by participating subjects throughout their daily lives. Participants are each encouraged to take part in the study by wearing four Apple Watches regularly for 12 months; all of the data collected from them is stored on a single server at Imperial College London. This high-frequency multi-dimensional time series motion data will eventually be analysed using A.I. and machine learning techniques to extract the desired biomarkers. However, there are a number of challenges to address before reaching that stage: challenges in correctly collecting and pre-processing valid data, in making the data conveniently and efficiently accessible for analysis, and in storing and distributing the data to be co-located with sufficiently powerful resources for computationally-intensive analyses. The existing system as it was delivered did not include a way for clinicians or researchers to access any high level or decision-critical information about the data collection process, nor was there a computational strategy in place for how best to use the terabytes of data already stored on the server.

1.1 Goals

The key objectives of this project were twofold: firstly, to develop a data processing framework to facilitate the computation of relevant analytics on a large centralised data set on a single machine and, secondly, to produce an interactive web-based dashboard interface for use by clinicians and researchers both to monitor the live data collection process and to provide them with insights which could influence strategic decision-making.

The need for a bespoke analytics pipeline emerged because there is a lot of data to be processed in a timely fashion on a single machine; cloud computing resources and commercial analytics platforms were not financially viable given the upfront investment already made on a central server. There are myriad metrics and analytics which the research team may be interested in evaluating on the collected data, so the analytics pipeline serves as an instrument for running calculations which can guide further analysis and investigation.

The desire for a dashboard was rooted in a practical need, given the lack of transparency into the data collection process. Little was known about how many hours of valid data had been collected since the system was activated months ago, whether patients were or were not recording data on any given day, or whether there were faulty devices in use which needed to be replaced or manually restarted. This kind of information could only be queried or inferred by SSH'ing into the server and inspecting logs, using a remote access web application to view files and records on the laptops installed in participants' homes, or directly contacting the parents of those participating in the trial. The dashboard interface can be thought of as a microscope with which to inspect collected data and as a user-friendly tool for identifying anomalies which may require interaction with the trial participants.

The final system should make it easier for relevant clinicians and non-experts to monitor the ongoing data collection process through an intuitive interface and make it relatively straightforward for researchers to define, execute, and visualise different analytical computations despite the constraint of running on a single machine and working on a large data set. Together these tools will help support the *KineDMD* project and, by extension, the wider international effort to improve the lives of those affected by Duchenne muscular dystrophy.

1.2 Challenges

Apart from personal lack of familiarity with web technology, database performance optimisation, and designing scalable and user-friendly data analytics systems, the principle challenges to overcome were to do with the limited computational resources of the central server and the poor performance of queries to the MongoDB database already in use. It was also a challenge to develop a programmatic interface for specifying analytics computations which was both versatile and relatively straightforward for users to understand.

With only one central server to use, it was important to consider what optimisations were possible for each part of the system: what database to use, how it should be configured, how queries are constructed and sequenced, how data is stored in main memory, and how data is operated on.

1.3 Contributions

The first key contribution is a bespoke analytics pipeline with a programmatic interface which allows daily and all-time analytics to be defined hierarchically in terms of different data sources and other analytics. In this case, an *analytic* is a computation or function which produces an output that describes or summarises the input data. For example, the standard deviation of acceleration could be an analytic, or a list of time intervals of data that have been marked valid. The main component of the pipeline is a computation scheduler which loads configurations for different analytics from YAML files and sequences their order of execution to minimise I/O usage (one of the main performance bottlenecks), minimise main memory usage to support space-intensive operations on large batches of data, and to satisfy the dependency prerequisites of analytics which are defined in terms of other analytics. This makes it relatively easy to define and execute a metric or function which operates on each day of a user's data with minimal additional scripting or supervision.

The second key contribution is the web dashboard which serves as a visual aid for inspecting analytics results produced by the aforementioned pipeline. The dashboard also provides high-level summary information and alerts relevant to clinicians and non-experts supervising the ongoing data collection. This allows relevant team members to see which trial participants are using their smart watches on which days and at what times. It also allows them to identify potential hardware faults when data from one or more devices is missing, to identify mistakes that patients may be making in their use of the equipment, and to see or infer many other details relevant to supervising the clinical trials.

2 Background

2.1 Duchenne Muscular Dystrophy

At present, clinical estimates of the progression of DMD in individuals is quantified by the performance of patients in a variety of physical activities, such as the North Star Ambulatory Assessment (NSAA), which scores patients on a functional scale by tallying their performance in 17 common activities (e.g. walking or rising from the floor) [4], or the 6-Minute Walk Test (6MWT), which measures endurance by how far a subject can walk in 6 minutes [5]. There is a moderate to good correlation between the clinical scores produced by these tests [6], so they are reliable enough to act as ground truth values to train machine learning models, but they could be improved in various ways. Scores based on physical tests can be affected by mood, what was eaten for the last meal, irregular sleep, and many other factors that fluctuate over days or weeks. It is also possible for NSAA scores to be affected by the subjectivity of the observing clinician; they have to decide whether a patient has used any unusual movements to compensate for muscular weakness when performing some activities. The physical examinations are also relatively infrequent - for the current trials with *KineDMD* the checkups are once every 6 months - and inconvenient for parents, children, and clinicians: children can be intimidated by the environment, by the equipment they have to wear, and they may refuse to comply with instructions. There are also relatively few ambulant boys affected by DMD available to participate in lengthy clinical trials. All these shortcomings together justify the search for biomarkers that can provide more objective and insightful feedback more quickly and conveniently.

There is also an active search for other kinds of biomarkers, such as measuring the presence of certain molecules or proteins in muscle biopsies [7] or in blood serum samples [8], or using magnetic resonance imaging of muscle tissue [9]. Blood samples are less invasive than muscle biopsies and can be more reliable than physical tests like the NSAA and 6MWT. Magnetic resonance imaging is less invasive [10] than taking blood samples or muscle biopsies but it can be more expensive, time consuming, and inconvenient, requiring each patient to be in a machine for 1-2 hours [7]. Though many of these are promising avenues for improved outcome measures, and they may be differently applicable at different stages of DMD progression (e.g. depending on whether a patient is ambulant or not), there are still drawbacks in terms of cost, time, and how invasive they are. These shortcomings can all potentially be ameliorated by new digital motion-based biomarkers.

2.2 Medical Wearables and Digital Biomarkers

There are various efforts to use wearable sensors and continuous monitoring of patients to discover biomarkers for conditions that affect mobility. For example, ActiMyo [11] is a watch-like wearable device designed specifically for measuring precise movements of patients with neuromuscular or neurological pathologies, including Duchenne muscular dystrophy and Parkinson's disease. It has a triaxial accelerometer, triaxial gyroscope, and a triaxial magnetometer. Data is transmitted to a server for analysis whenever the device is docked on a docking station. Prototype ActiMyo sensors were used in a study to find potential digital biomarkers for DMD [12]: one ActiMyo sensor was worn on each wrist and 7 patients participated in a modified version of the Minesota dexterity test, which measures a range of upper limb movements. They found that the norm of the angular velocity and the mean of the elevation rate (angular velocity at which forearm is lifted) were fairly reliable and correlated well with the scores obtained in most of the activities from the Minesota test. These results emphasise the potential for using wearable sensors to derive digital biomarkers based on limb movement.

Aparito [13] are a company who specialise in wearable devices for real time patient monitoring. They have developed their own watch-like wearable equipped with a variety of sensors to count steps taken, measure distance moved, heart rate, and other high-level data, however it is not clear from their public information whether the device transmits raw movement data for later analysis or if it only uploads summary analytics computed on the device. Aparito provide a mobile phone application to allow clinical trial participants to log arbitrary extra relevant information like newly observed symptoms, an unusual event or accident which might affect the interpretation of recorded data, or a problem with a device. They also provide a real-time analytics dashboard to their research partners to help medical professionals supervise clinical trials. They have partnered with Duchenne UK to conduct a feasibility study in order to develop a version of the aparito platform tailored for DMD [14].

Wearable motion sensors have been used to help monitor and diagnose other conditions that affect mobility, such as Parkinson's disease. In one study of patients with PD, where their movements at home were recorded continuously by 3 wearable inertial sensors for one week, it was discovered that PD patients could be recognised by detecting and characterising their turns [15]. People with PD would take shorter turns with smaller turn angles and take more steps when turning. Other investigations found that measuring postural sway, anticipatory postural adjustments, and arm swing while walking were good for diagnosing PD and predicting falls [16]. These kinds of discoveries suggest that it is possible to find distinctive features in the way people move that can serve as biomarkers for a condition that affects mobility. It is this confidence that motivates the wider search for new motion-based digital biomarkers for DMD.

2.3 Existing System

The existing data collection system for the *KineDMD* study was deployed by a previous Master's student, Shuai Zeng [17]. Each participating patient is asked to wear 4 Apple Watches, one on each ankle and each wrist, on as many days as they can for 12 months and for as long as possible each day. All the watches record measurements from their accelerometer, gyroscope, and heart rate sensors continuously at 100Hz. This section will be an overview of how the system works, first from a data processing perspective and then from a user perspective.

Data

The movement of data throughout the collection process can be divided into four stages: recording sensor data on the smart watches, transmitting the sensor data to a laptop base station, uploading the data to the central server, then finally decompressing and inserting the data into a database on the server. These stages are illustrated in Figure 2.1.

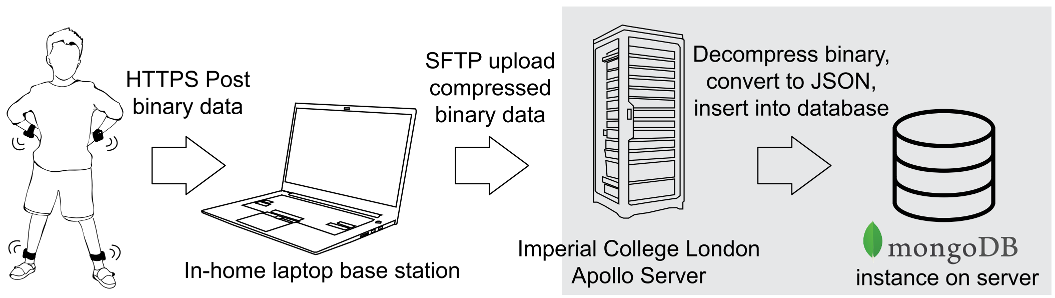


Figure 2.1: Arrows indicate the movement of data. In the final step the data is not moved to another machine but into a database on the same machine. The arrows are annotated with the communication protocol used to transmit the data.

There are two kinds of data collected from users: raw sensor data from the watches and manually logged usage timestamps. The laptops deployed in patients' homes have a user interface with 4 toggle switches (one for each watch) to indicate when the boy is wearing which watches. Whenever one of the buttons is pressed, a record is added to a plain text timestamp file indicating which watch changed status (RH, LH, RL, or LL), what the status was (On or Off) and at what time the change happened, for example `LH On 2019-06-11 08:34:47.708339`. Since the watches are programmed to record data whenever they aren't charging, whether or not they are being worn, then these manually entered timestamps can be used to discard data from devices that doesn't fall within the time intervals they were actually worn.

The other kind of data, sensor readings, consists of a series of data frames which each capture readings at one moment in time. Each raw binary data frame produced by the custom Apple Watch application has 11 fields, occupying a total of 88 bytes, thus 12 hours of 100Hz data from one watch is approximately 360MB (over 4 million samples), so each patient can generate over a gigabyte of data per day. The tick timestamp field, `ts`, is a 64-bit unsigned integer and all other fields are double precision floating point numbers. The other fields include the tri-axial values for the rotational acceleration from the gyroscope, the absolute rotation relative to a reference frame (roll, pitch, and yaw), the accelerometer acceleration values, and the heart rate (See Figure 2.2).

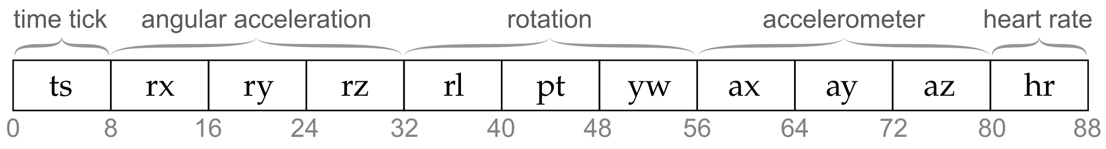
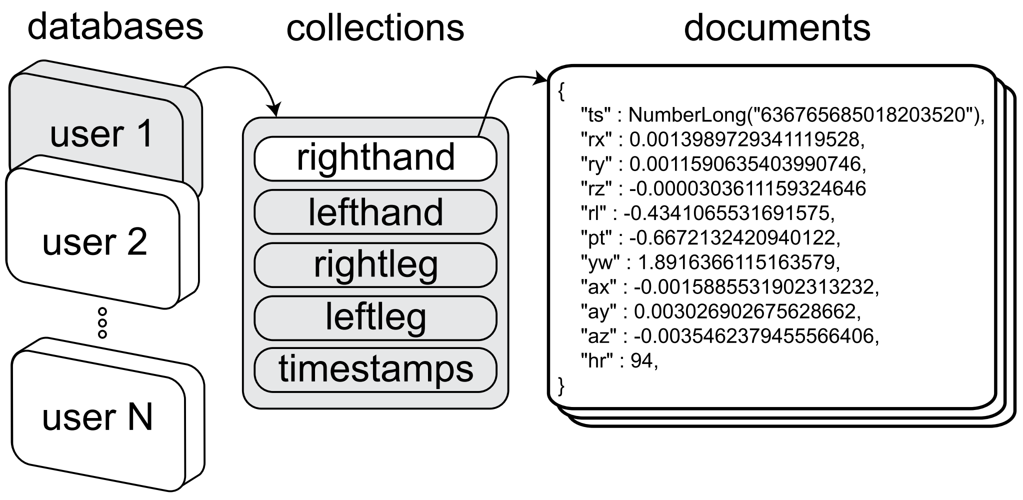


Figure 2.2: Field structure of one raw binary data frame generated from Apple Watch sensor readings. The numbers represent the position of the field, in bits, within the data frame.

The raw binary data from the watches is compressed on the laptop with the lossless LZMA compression algorithm before transmitting to the server, which reduces the file size (and therefore transmission time) by 50%. Given the limited download bandwidth of the central server, the upload process is staggered across the users participating in the trial, so they take turns uploading data in batches. Once received, the server unzips the compressed binary data from each watch, transforms them into JSON files, then inserts them into MongoDB. MongoDB is a schema-less database which consists of multiple *databases* (one for each user), which each have multiple *collections* (one for each watch and for the manually logged timestamps), which each contain many *documents* (individual data frames or timestamp records). In Mongo-speak, an SQL table is a collection, an SQL record is a document, and an SQL column is a field. Since MongoDB is schema-less, individual documents within collections are self-describing: the documents themselves contain all the information necessary to read them, and the documents in a collection don't need to have the same structure. This can provide convenient flexibility, however the data for this clinical trial is highly structured and repetitive and therefore unlikely to change. In Chapter 3 we investigate alternative databases which may afford better space and time-saving properties than MongoDB.

The time series data frames are stored in a particularly redundant format. Every individual document has string identifiers for each of the 11 fields, all of which are stored on disk. As shown in Figure 2.3, the JSON document for each data frame has all of the 2-character string keys in it. MongoDB has a very sophisticated compression scheme so, despite the string keys being duplicated in every document, the total size on disk is only slightly larger than raw binary files. However, this format leads to performance problems when querying data. The performance drawbacks of MongoDB will be discussed in the Optimisation section of the analytics pipeline on page 36.



```
{
  "ts" : NumberLong("636765685018203520"),
  "rx" : 0.0013989729341119528,
  "ry" : 0.0011590635403990746,
  "rz" : -0.0000303611159324646
  "rl" : -0.4341065531691575,
  "pt" : -0.6672132420940122,
  "yw" : 1.8916366115163579,
  "ax" : -0.0015885531902313232,
  "ay" : 0.003026902675628662,
  "az" : -0.0035462379455566406,
  "hr" : 94,
}
```

Figure 2.3: On the left are depicted the individual databases belonging to each user, in the middle are the collections within one of those databases, and on the right is one of the JSON object documents inside one of those collections.

Usage

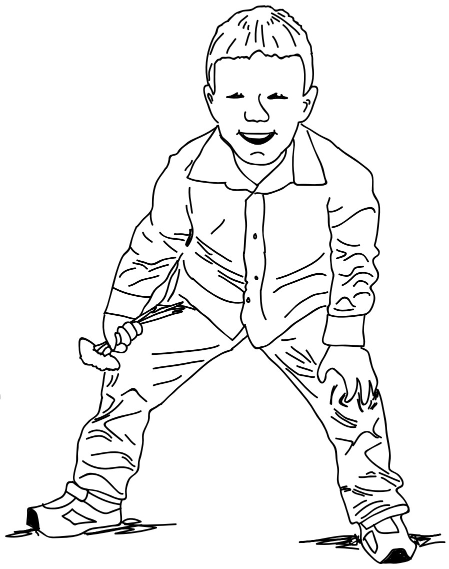
It is important to know when a watch is being worn because the watches will collect data as long as they are not docked for charging, regardless of whether they are being worn or not, but making this distinction is not straightforward. There are three options: ask patients to input manually all the times that they wear the watches, automatically detect when a watch is being worn, or use a mix of both strategies. Trial participants (children and parents) are instructed to record when they put the watches on and when they take them off via a digital interface on their provided laptops, however since they don't take their laptops with them out of the house they cannot record if the watches are taken off while they are out (e.g. during a school sports class). As a backup plan, kids are also told to perform a signature action after putting the watches on and before taking them off, involving raising each limb in a specific order one after the other. It may be possible to use this signal to automatically detect when the watches are worn. There have been some preliminary investigations into using the raw motion data and heart rate data alone to classify when a watch is being worn, but that is a tricky problem. How to address this issue is beyond the scope of this project.

Patients are instructed to put the watches on charge when they have finished wearing them each day; it is whilst the watches are charging that they attempt to upload data to the laptop base station. Watches can collect a few days' worth of data before their tertiary storage is saturated, at which point data collection will halt, so it is possible to use them for some time without reconnecting to the base station.

According to our current calculations, patients wear all 4 watches simultaneously for an average of 7.5 hours per day, however this does not account for all the times when the watches have been recording data without being worn. At the time of writing, there are ≈3.7TB of data already in MongoDB.

Another Master's student has developed a new data collection pipeline that doesn't require a laptop base station to be setup in patients' homes, doesn't require a central server, and allows patients to log when they are using the watches on the devices themselves instead of only on a laptop at home. With that system, the Apple Watches are configured to connect directly to an API endpoint on Amazon Web Services and to upload data directly to a cloud data lake. However, the new setup cannot be used without approval from a clinical authority nor until funding is available. As such, the central server (which has already been approved for storing clinical data) will be used for a long time in the future and needs to be harnessed as fully as possible.

3 Analytics Pipeline



The initial project proposal was to create a web dashboard to display daily, weekly, and all-time aggregate analytics and summary information. Most analytics take minutes to compute, which is too long for a responsive user interface, so the analytics pipeline developed here is designed to pre-compute relevant results to be displayed on the dashboard.

The analytics pipeline is the backbone of the dashboard; it is responsible for accomplishing all daily and historic computations. It loads definitions of analytics from YAML configuration files, it queries for and operates on patient data stored in MongoDB, it can be invoked through the server command line and runs daily as a `cron` job, it stores results in MongoDB which can be displayed on the dashboard (Chapter 4), it automatically detects when certain analytics need to be recalculated (e.g. when new data for a certain day arrives), it orders internal calculations to satisfy dependencies in the defined analytics, and it minimises its I/O usage and main memory footprint. The analytics pipeline is also a standalone tool to execute user-defined computations over the whole patient data set. This chapter is a breakdown of the setup, design, and implementation of the pipeline.

3.1 Project Setup

For version control, the GitLab service hosted by the Department of Computing was chosen over GitHub so that the project is not visible to the public and appropriate access privileges can be configured for supervisors and relevant colleagues already on the Imperial network (free private repositories on GitHub can only have up to 3 collaborators [18]).

Python was chosen as the language of implementation for the analytics pipeline and for the front-end web dashboard because it had been used to write many of the scripts on the server. Code written in Python would be easier for other team members who know the language to maintain after the Master's project is over.

`virtualenv` [19] was used to create a virtual Python environment for isolating module dependencies; this prevents newly installed modules from replacing or interfering with globally installed modules used by other programs on the server.

A local instance of MongoDB was used on a development machine with synthetic test data to facilitate quick feedback and iteration during development. The artificial data set uses the same JSON structure as the documents in the server-side database, but it can be optionally generated with a low sample rate to make experimental computations faster (less documents per day of data). This fake data is first stored on disk as JSON files grouped in folders for each test user before it is inserted into a local MongoDB database. In this way the same test data can be re-used for benchmarking other procedures which don't use MongoDB as an interface to the data (e.g. other databases or file-handling scripts).

`pytest` [20] is used for unit testing, and a shell environment variable is used to toggle local testing mode. The environment variable determines whether `pymongo` [21] (a Python MongoDB driver) connects to the local developer database or the production database on the server, as well as the string format for matching the right user names in the database (e.g. 'sg01' vs. 'test01').

Finally, before working with the data stored in MongoDB on the server, it was necessary to add indexes to each collection of time series data frames for each user. This was because of a hard constraint in MongoDB that large collections need to be indexed before sorted queries can be performed. The indexes were added with `db.collection.createIndex({ts:1})` (indirectly via the equivalent `pymongo` function), where `ts` is the integer tick timestamp of the data frames in the collection.

With a version control system in place, development and testing environments setup, and a production database configured for basic queries, I could start designing the analytics pipeline and investigating potential performance bottlenecks.

3.2 Design

By 'design' we mean high-level decisions about how a system is to be used and implemented, where principles such as simplicity, security, versatility, efficiency and others may be invoked. Design is required when there are *constraints* which can be satisfied by a variety of potential solutions and where it is not possible to conduct a comprehensive empirical investigation into the merits and shortcomings of every alternative.

The basic goal was to build tools which would help those responsible for supervising the lengthy clinical trials, and support data analysts as the project gradually moves into a more focused biomarker discovery phase. The initial request made in the project proposal was for an analytics dashboard, but the request did not include specific performance requirements or many feature requirements, nor even any explicit mention of a bespoke analytics engine. The project was quite open ended and required initiative and investigation to determine what kind of systems or tools would be most appropriate and helpful.

The first major design decision was made here: if I were only trying to create a productive setup for use by others after I was gone, I could have done so by installing and configuring mature and feature-rich self-hosted analytics software, perhaps adding some custom routines and pre-processing steps for our particular data. This may well have had a higher long term utility value for the team, however there are three reasons I opted against it. Firstly, platforms that provide high-level data processing features usually do so by abstracting away the interface from an underlying computational engine and hiding the internal implementation. This means that, in some cases (perhaps ours), the lack of fine-grained control over how data is handled could lead to performance bottlenecks that can't be overcome without modifying the platform source code. Without access to the source code, such a bottleneck would become a permanent drawback of the final system. On the contrary, a custom analytics engine built by one person in a relatively short time may not have the kind of sophisticated optimisations expected of a mature analytics product, but a custom system would at least be open for further modification as required. Secondly, no extra funding was allocated for this project and the range of free or open source analytics software available is comparatively small and not as mature as popular commercial products and services. Finally, I would learn a lot more in designing and developing a bespoke system than in re-using existing products or services. So, to have the flexibility of a ground-up implementation, to avoid spending any money on existing products or services, and to maximise learning, I rejected the possibility of using an existing analytics platform for this project. Now, before I could decide how to start building a custom platform, I needed to understand the constraints of the physical hardware I would be working with.

Constraints

The first constraint to consider is **available CPU time**. The server already spends a portion of every day receiving, unzipping and inserting large binary files into MongoDB. According to Shuai's report [17], the server will be busy all day every day when there are 70 active participants submitting data daily. At the moment, there are less than 20 active patients. If each patient runs some analytical calculations every day, then the maximum capacity of the server will drop correspondingly. Longer analytics computations mean lower total capacity.

The second constraint is **how much data is stored already**; newly defined analytics may be calculated for any existing data, but any calculations on historic data will consume CPU time which may be needed for the essential daily processes. At the time of writing there are ≈5000 hours worth of *valid* data (≈590GB) stored on the server, but nearly 4 terabytes in total from the 20-or-so subjects currently participating (some stopped participating early in the trial). Depending on how many people wear their watches and for how long, between 3 and 15 gigabytes may be added on any given day. However, more participants will eventually be added, so the system should ideally cope with a bigger load. Since there is so much historic data for which to calculate analytics, it is important for the analytics computations to use CPU time sparingly.

The third constraint to consider is **available RAM**; the server has 64GB of main memory. This constraint is relevant to the question of what quantity of data should be the unit of computation for the analytics scheduler, that is, how much data to load into main memory from disk via the database drivers as one computational batch. This constraint is also relevant to how the data loaded from the database should be represented in main memory.

The fourth and final constraint to consider is the **I/O data read rate** when retrieving data from MongoDB with the Python driver, `pymongo`. When calculating simple metrics like mean and standard deviation, almost the entire CPU time (≈1-2 minutes for a day of data) is spent just reading data from MongoDB.

The straightforward meaning of these constraints is that the analytics engine should make it a priority to employ optimisations which minimise I/O (reading from MongoDB), save CPU time, and minimise the size of the internal representation of data in main memory.

Unit of Computation

The analytics pipeline is structured around days of data as a unit of computation: one day of patient data is loaded into main memory, various analytics are computed for that day, then the data is dropped from main memory to make room for the next day. There are a few reasons for this choice: new data arrives in daily batches, this scheme is simple and thus easy to implement correctly and to optimise, one day of data can fit in main memory with enough room to do complex operations on it, it is logical given the request for daily analytics to be displayed on the dashboard, and it creates the possibility for storage optimisations which save space on disk and allow for reading whole days of data quickly as contiguous blocks of memory. This is discussed again in the section on Replacing MongoDB under Implementation (page 36).

So, the decision had been made to build a new analytics pipeline, performance constraints had been identified, and days of data were chosen as the unit around which to structure the rest of the system. With many users, many days of data, and many possible analytics for each day of data, we now present the hierarchical format chosen for specifying analytics.

Hierarchical Analytics Specification

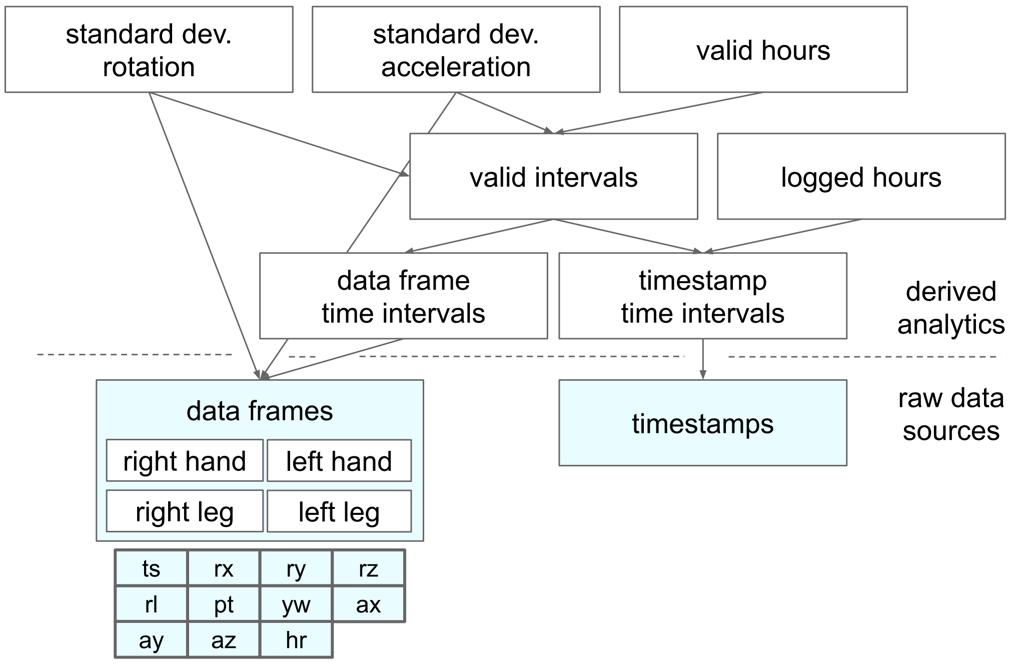


Figure 3.1: Blue boxes denote raw data sources, white boxes denote derived analytics. An arrow from box X to box Y means that analytic X uses Y in its computation so Y must be computed first. The small boxes below the 'data frames' box show the individual data frame fields that can be required by a derived analytic.

The format is called hierarchical because analytics of interest can be specified either in terms of raw data sources or other analytics. The raw data sources are (1) the time series data frames from the Apple Watches and (2) the usage timestamps manually entered by a user with the buttons on their provided laptop. Other analytics are any results computed either from raw sources or other analytics, for example, one analytic may be defined as the ratio between two other analytics which each output a single number.

The reason for including this notion of hierarchy in the specification format is so that new analytics can be defined conveniently in terms of other intermediary results and so that the scheduler can preemptively and automatically compute all the prerequisites for a given analytic. For example, if the ratio of the standard deviation of the x and y components of acceleration is requested, the analytics scheduler will first make sure the standard deviations of the x and y components have been computed, yet without recomputing them if they have already been calculated. This is important because recomputing an intermediary analytic could mean reading raw data frames from MongoDB, which is a very expensive operation.

Consider Figure 3.1: this diagram illustrates some user-defined analytics which require a mix of inputs, either raw data sources (blue) or other derived analytics. For example, ‘valid intervals’ requires both ‘data frame time intervals’ and ‘timestamp time intervals’. In practice this means that, for a given user and a given day, if the ‘standard deviation of rotation’ analytic is requested then the execution scheduler will first check if ‘valid intervals’ has been computed and, if it hasn’t, schedule it to be computed first. In this way the user of the system doesn’t need to think about which computations to perform in what order - as long as it has been defined in the specification that one analytic depends on the other, the scheduler will always make sure prerequisites are computed first. This hierarchical specification format also allows for a variety of performance optimisations, which are discussed in the Optimisations section on page 34.

YAML Configuration Files

YAML **A**in’t **M**arkup **L**anguage is a "human friendly data serialization standard for all programming languages." [22] New daily and all-time analytics can be defined by adding an entry to the corresponding configuration file. For example, the analytic ‘actual_hours’ which is derived from ‘actual_intervals’ is specified like this:

```yaml
actual_hours:
  defined_on: 'other_stats'
  requires: ['actual_intervals']
  path: 'analytics/intervals.py'
  function: 'compute_actual_hours'
```

This specification format will be referred to throughout the explanation of the rest of the pipeline since some of the options will only make sense in context. Here it suffices to show that the format is easy to read and write.

In summary, we now have a high-level view of relevant resource constraints, why the system is structured around days of data, and the idea of hierarchical analytics defined in YAML.

3.3 Implementation

With a big-picture idea of how the system is structured now in mind, we turn to describe in detail how the system was implemented, starting with smaller preliminary considerations and escalating to the main algorithm behind the analytics pipeline.

Identifying Valid Data

Before calculating any analytics derived from raw data frames (e.g. standard deviation), it was necessary to introduce a distinction between valid and invalid data. Invalid data includes any data frames recorded by a watch when it wasn't actually being worn, and it also includes any data recorded at a time when data from one of the four watches is missing. Conversely, the data from each device at one moment in time is valid if (1) there is data from each device and (2) the devices are all being worn. For example, data between time t_0 and time t_1 is invalid if only the left hand and left leg successfully recorded data within that interval.

Our filter for valid data will ultimately be represented as a list of non-overlapping time intervals $[t_0, t_1], [t_2, t_3], \ldots.$ A simple way of finding these intervals is as follows: first, find the time intervals for which data has been recorded by each device, then find the time intervals induced by the 'On/Off' timestamps which denote when a patient is (supposedly) wearing their watches, then find the intersection of all of these intervals. This will yield a set of all intervals within the user marked timestamps where there is data from every device.

For the first step, we need to calculate the time intervals for which data has been recorded for each device: we load all the integer tick time values for a specific day into memory and scan through them chronologically to find the start and end of each interval. An interval ends if the next time tick is more than 9 seconds in the future, and a new interval begins at that future tick. This threshold is not arbitrary; the Apple Watches record batches of data frames in small files of about 10 seconds. Some of these small files are dropped when transmitting to the laptop base stations, so this threshold for starting a new interval was chosen to correctly detect those gaps in the data whilst still allowing for leniency for individual frames to be dropped. This process, and the use of the 9 second heuristic, is illustrated in Figure 3.2: some frames can be dropped and still be included in one time interval, the frames don't all have to be exactly 1/100th of a second apart, but gaps in the data larger than 9 seconds will be treated as ranges where data is missing.

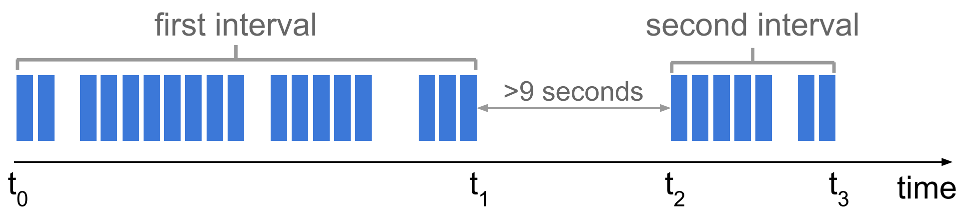


Figure 3.2: Blue rectangles represent a data frame measured at one moment in time. The first interval is $[t_0, t_1]$ and the second interval is $[t_2, t_3]$.

Now that we can determine the time intervals for which we have data for each device, we need to do the same for the manually entered timestamps: we want to take a list of 'On/Off' records and turn them into a list of intervals $(t_0, t_1), (t_2, t_3), \ldots$. To achieve this, we need a clean stream of On/Off pairs, however the interface which patients use on their laptops to enter these state changes allows them to submit duplicate On/Off signals. For example, they can input that they started wearing a watch at time t_1, and that they started again at time t_2, and again at time t_3 etc... with no signal in between that they stopped wearing the watch and only eventually ending with an off signal at some time t_4. This appears in the timestamp records text file as 'On at t_1, On at t_2, On at t_3, Off at t_4', which cannot be interpreted directly as a time interval. To correct this, we employ the conservative strategy to take the *last* 'On' signal when the signal is repeated and to take the *first* 'Off' signal when that signal is repeated. Applying this process leaves a clean list of On/Off pairs that can each be read as the start and end of a time interval. We thus select the smallest time intervals which are consistent with the user supplied On/Off signals. This is illustrated for clarity in Figure 3.3.

With a clean stream of On/Off timestamp pairs, we can easily interpret them as a list of [start, end] intervals. As described above, we also have an algorithm to get a list of time intervals of recorded data from each watch, so we can return to the problem of finding intervals of valid data.

We defined the intervals of valid data as the intersection of the time intervals of data collected from each device *and* the time intervals manually logged by the user. This is illustrated in Figure 3.4. The blue highlights indicate slices of time where the time intervals from each device and the user timestamps all overlap. We treat these blue areas as valid data because in each of those intervals we have data from every watch and it is data collected while the patient was (probably!) wearing the watches. However, this simple way of distinguishing valid and invalid data has some caveats which could not be overcome in the scope of this project.

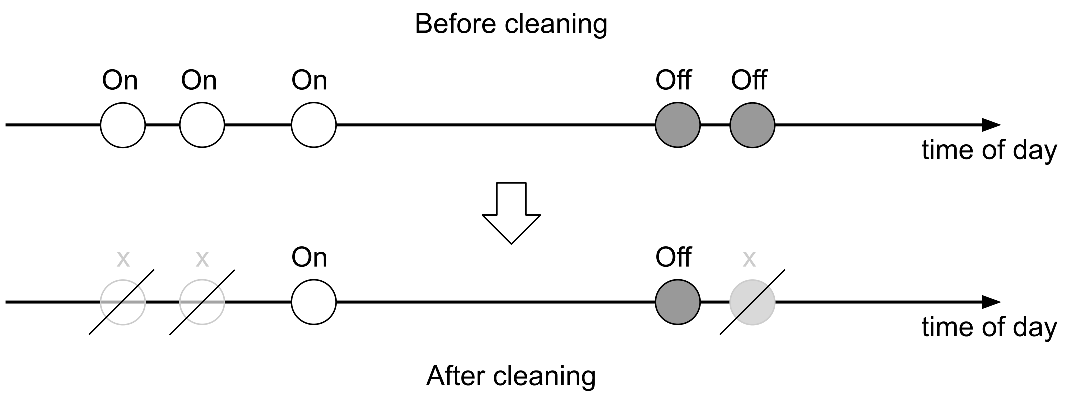


Figure 3.3: The first arrow of time (top) shows a sequence of timestamps with duplicate 'On' and duplicate 'Off' signals. The second arrow of time (below) shows the same timestamps after the duplicates have been pruned, leaving the smallest interval consistent with the originals.

The simplifying assumptions employed here about what constitutes valid data are not immune to anomalies. Namely, (1) the assumption that a user is always wearing a watch during the times they've indicated that they are, (2) the assumption that they always remember to log that they are wearing a watch when they are, and (3) the assumptions about repeated timestamps, namely that we keep only the last repeated 'On' and the first repeated 'Off'. These assumptions break in a few cases. Firstly, children can take their watches off during the day without manually logging that they have done so; the laptop with the activity logging interface may be at home, so they couldn't input this information even if they remembered to, so that data will be treated as valid even though it isn't. Secondly, the trial participants sometimes forget to input when they started or stopped wearing the watches, giving the impression either that they were wearing the watches for a longer or shorter time than they did in fact. Thirdly, they sometimes (accidentally) log that they started wearing a watch just before logging that they've stopped wearing it, which our cleaning process will interpret as only using the watch for a very short time by only keeping the most recent 'On' signal. The only reliable way to prevent these special cases is to use a much more sophisticated analysis to automatically detect when a watch is being worn. Participating patients were instructed to perform a specific sequence of actions (raising each limb) after putting the watches on and before taking the watches off. This signal could be extracted from the motion data and used in place of (or together with) the manual timestamps. Our simple scheme suffices for most cases and can be replaced relatively easily by changing the timestamp cleaning process.

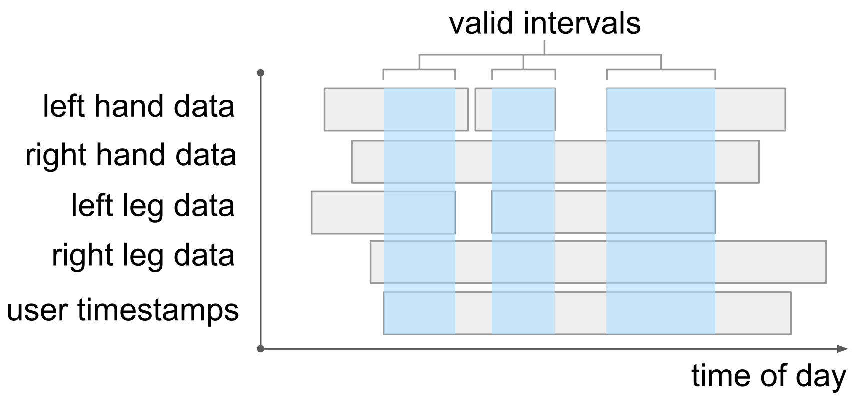


Figure 3.4: The vertical axis lists different groups of time intervals. For the data from different limbs, the grey boxes represent the time spans for which data was recorded. For the user timestamps, the grey box represents the duration of the recording session marked by the 'On/Off' signals manually entered by the user. The blue areas highlight the ranges where all the intervals overlap - the intersection of all five sources.

Each user's cleaned timestmaps are stored in a separate collection in mongodb, `timestamps_cleaned`. The documents inside the collection have the exact same structure as those in the `timestamps` collection. This collection, and the script which refreshes it, `clean_timestamps.py`, can be replaced later to change the way invalid data is filtered out.

Synchronising Watches

The other preliminary processing step needed before computing some analytics is to synchronise the data frames produced by the different watches, because they have different clocks. They all take readings at approximately 100Hz, but the exact time in clock ticks between each frame varies, and the measurements from one watch may be offset from the measurements of another by up to 0.005 seconds (half of the delay between frames). The algorithm described here will take in a list of data frames from each watch and output a single list of combined data frames; each output frame will have all of the fields from all of the watches but with a single time tick value. Analytics which are based on data coming simultaneously from all the watches can then use these combined frames as their input.

The first step to synchronisation is to load all the data frames for each watch for a certain day into memory. Then, identify the watch whose frame tick times will be used as the reference signal - these will be the tick values which appear in the final output frames. We choose the reference watch by finding the watch for which the first frame has the latest tick time compared to the first frame of the other watches. This guarantees that the other watches will have at least one frame before the first reference frame, which is required for interpolating the non-reference frames.

Next, the algorithm will iterate over each frame of the reference watch and generate one combined frame. For each frame of the reference watch, the algorithm needs to find a pair of frames for each non-reference watch to interpolate between, one just before and one just after the reference frame time. So, for each non-reference watch, the algorithm finds the latest frame which is earlier than the reference time and it also finds the earliest frame which is later than the reference time, i.e. the two closest frames from just before and just after the current reference frame. If the reference frame is at time t_1 and the frames for another watch just before and just after are at times t_0 and t_2 (respectively), where $t_0 < t_1 < t_2$, then the interpolation paramater $0 < \alpha < 1$ will be $(t_1 - t_0)/(t_2 - t_0)$. The data frame values for that watch will then be linearly interpolated: if q_0 is the value of the field 'q' in the frame at t_0 and q_2 is the value of the field at t_2, then the new interpolated value for q will be $(1-\alpha)q_0 + \alpha q_2$. This situation is depicted in Figure 3.5.

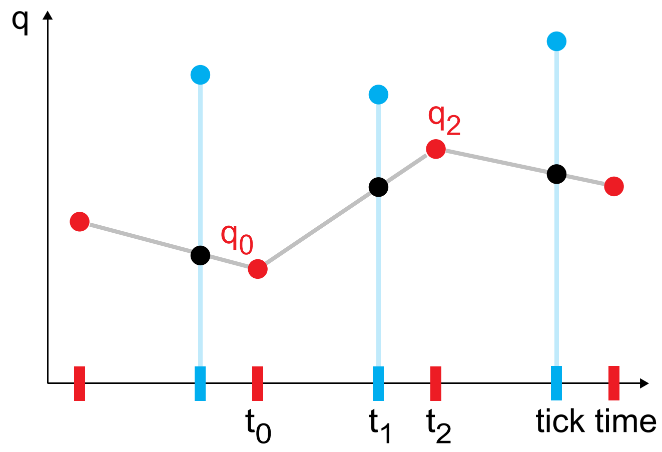


Figure 3.5: The ‘q’ on the vertical axis represents any single field of a data frame (e.g. x acceleration). The tick marks on the horizontal axis mark the times at which measurements have been made by different watches: blue for the reference watch, red for another watch. The black points are the linear interpolations of the red points which have been aligned with the blue tick times.

To see how this works more intuitively, see Figure 3.6. We've now explained how time intervals of valid data are determined and how data from different watches can be synchronised. Both of these preprocessing steps are required before basic analytics can be computed.

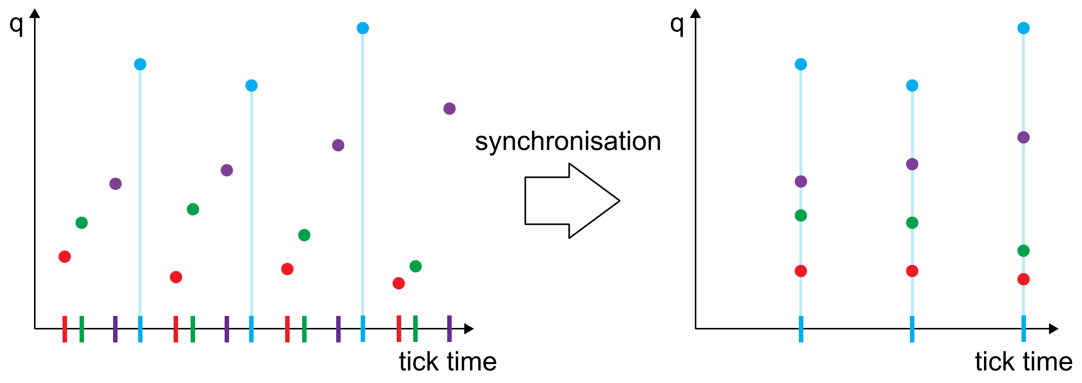


Figure 3.6: Blue signifies the watch whose tick values are the reference for synchronisation. Other colours are data points from other watches. On the left, the tick values of the different watches have not been aligned by interpolation. On the right is the output of the synchronisation algorithm.

Storing Results in MongoDB

Before we go into detail about the core algorithms that make up the data processing pipeline, it will be helpful to describe the way in which results will be stored. The exact format in which results from analytics computations are stored in the database changed throughout development, mainly to support better automatic recomputation detection and to distinguish between device-specific and day-specific analytics. For simplicity we only describe the final version.

Each user already has a database with collections `righthand`, `lefthand`, `rightleg`, `leftleg`, `timestamps` and `timestamps_cleaned`. We add a new collection, `daily`, where each document in that collection contains the results for all of the analytics computed for that user on a specific day.

Every document in `daily` has some mandatory fields: `year`, `month`, and `day` to signify the date for which the document contains results. Each document will also have the field `righthand`, `lefthand`, `rightleg`, and `leftleg` to contain results specific to each device. Finally, each document will have a `last_updated` field to contain all the timestamps denoting when each analytic was last updated.

Every document can have any number of other fields whose values are the output of analytics computations. If an analytic belongs only to a specific device, then the value is stored under a field by the name of the analytic within the object for that device (e.g. the object under the `"righthand"` field).

An example results document is shown in Listing 1. Here `sd_acceleration` is a result specific to the device `"righthand"`, but `logged_hours` is not device-specific, so it is a top-level field in the document. The structure of the JSON document under `last_updated` mimics that of the daily document itself, so it has one field for each of the watches (righthand, lefthand, etc...) which is like a

```
{
    "day": 27,
    "month": 6,
    "year": 2019,
    "logged_hours": 9.28952591288889,
    "righthand": {
        "sd_acceleration": 0.2768680132106585
    },
    "lefthand": {},
    "rightleg": {},
    "leftleg": {},
    "last_updated": {
        "logged_hours": "2019-06-17 11:03:21.017349",
        "righthand": {
            "sd_acceleration": "2019-06-17 11:03:21.017349"
        }
    }
}
```

Listing 1: An example JSON document for one day of analytics results in the `daily` collection of a user's database.

mini-document, and it has top-level fields for each of the analytics which apply to the day as a whole.

The important details to recall about this format (before we dive into the details of the execution scheduler) are these: (1) there is a distinction between analytics computed for the day as a whole and analytics computed for a specific device, (2) each result has a `last_updated` timestamp, and (3) not having a certain field means that a result has not been stored for the analytic with that name.

Execution Scheduler

Having determined the preprocessing scheme to find time intervals of valid data and synchronise data from different watches, and having determined a storage format, it was then possible to begin integrating these into a more functional analytics pipeline. We now turn to the central piece of the analytics pipeline, the execution scheduler. This is the code responsible for determining what analytics to execute and in what order, loading the right data from MongoDB, pre-processing the data, executing the analytics, and storing them back in the database.

The scheduler for daily analytics begins with a call to the Python function `compute_analytics_daily`, which takes 4 optional arguments: a list of user names, a list of dates, a list of the analytics to compute, and a boolean flag to force recomputation. If no specific users are supplied, the scheduler will run the analytics for the specified dates for all users. If no dates are specified, the scheduler will select for each patient all the days between (and including) the first and last dates for which they have data, effectively computing analytics for all available dates. If no analytics are specified, the scheduler will run all available analytics for each user and for each date. If the recomputation flag is not specified, the default is to only compute analytics if they haven't already been computed and don't need to be updated.

Suppose the scheduler has selected a list of users, a list of dates to operate on, and a list of analytics to compute. The main loop will then iterate over every user-date pair, create an ordered queue of the analytics to compute, decide on the most optimal query to MongoDB, load all the required data for the day into memory, execute the analytics one by one and store the results in MongoDB, outputting any errors and skipping users, dates, or certain analytics gracefully.

There are three core components in this system: (1) the algorithm which builds the queue of analytics to execute, (2) the algorithm which constructs queries to make to MongoDB and caches returned data, and (3) the process which executes each of the analytics in the queue with the right data and stores the results in MongoDB.

Loading Analytics Configurations

Before processing a request for some analytics, configuration data is loaded from YAML files. Requested analytics are specified by the names under which they are defined in the config files. The objects encoded in those files are parsed as Python dictionaries (key-value maps). When the objects specified in those files are parsed, the `path` and `function` fields (both string values) are replaced with a single field, `function`, which is a reference to the Python function which was referred to which will have been loaded into memory. The analytics scheduler then has a dictionary which maps names of analytics to dictionaries which contain all the information and settings needed to schedule and execute the named analytic.

Building the Queue

Given a user, a date, and a list of analytics to compute, the execution scheduler will build a queue of analytics to compute. The main constraint that the queue needs to satisfy is that analytic results which are required as prerequisites by other analytics must come first in the queue. For example, if `standard_deviation_rotation` depends on `actual_intervals`, then `actual_intervals` must come first. These kinds of dependencies are depicted graphically in Figure 3.1 on page 19. Also, the same analytic should not appear more than once in the queue.

The function which builds the queue, `create_execution_queue` takes 3 arguments: a connection to the user's mongodb database, a date, and a dictionary of analytics to compute. The dictionary of analytic information objects is a subset of the configuration objects loaded from the YAML configuration files.

The algorithm for building the queue begins by loading the JSON document from the user's `daily` collection which contains all analytics already computed for the requested day; this will be used to decide whether the requested analytics need to be computed or not. The queue builder will then initialise an empty queue of 'commands' (Python objects representing analytics to execute) and an empty set of the names of the commands which will be checked to avoid putting duplicates in the queue. The output of the function will be a queue of command objects which encode analytics computations to execute. The `create_execution_queue` function is depicted in Figure 3.7.

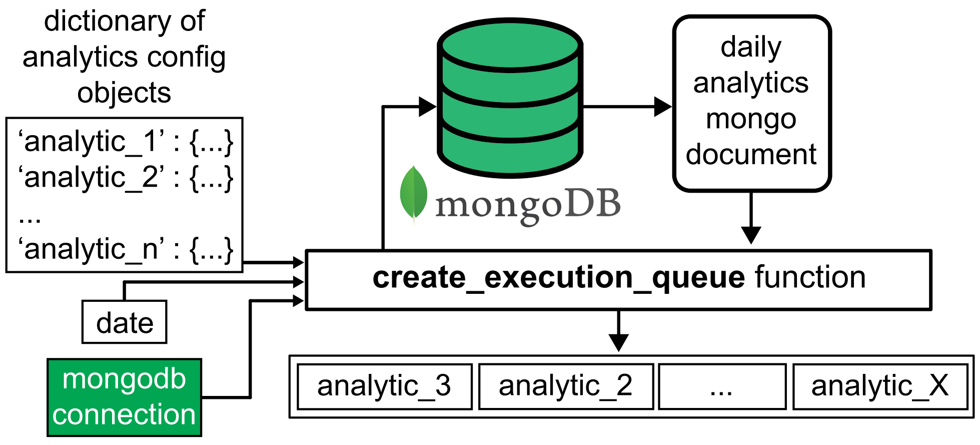


Figure 3.7: On the left are the three inputs, indicated by arrows going into the `create_execution_queue` function box. The arrow going to the mongoDB database shows the function retrieving the analytics results document for the chosen day via a database query. The downwards arrow exiting the function box points to the output, which is a queue of analytics jobs to run.

The main loop of the algorithm will consider each of the requested analytics info objects in turn and, if appropriate, insert command objects into the queue. Command objects may contain some extra information compared to the info objects from which they are constructed, such as the devices on which to calculate an analytic (e.g. only the lefthand).

Suppose the queue building algorithm is looking at analytic info object X. Firstly, it will check if analytic X requires only valid data as input. This requirement can be either explicitly specified as an option in the YAML config or automatically inferred if the analytic requires synchronised data from all watches. If the analytic does require only valid data, it will have the analytic `valid_intervals` appended to its list of dependencies.

The next step of the queue builder is to schedule all the analytics which X depends on, that is, to put X's dependencies in the queue first. The queue builder does this by recursively calling the queue insertion function with each analytic that X depends on. For example, if X depends on Y and Y depends on Z, the queue building algorithm will first look at what X depends on and request to schedule Y first, but it will then look at what Y depends on and decide to put Z in the queue, then it will put Y in the queue, then finally insert the X that was initially requested. However, if Z or Y were already in the queue when the algorithm was checking X's dependencies, X would be added to the queue immediately.

There are some other checks that happen before putting an analytic in the queue. If the `recompute` boolean flag is set to true, the requested analytics will be computed even if a result is already stored in the database. If the flag is set to false so that computation of a new result is not forced, then a few checks are performed to decide whether the analytic needs to be calculated. If it does not need to be calculated, the queue-building item will move on to the next request.

An analytic is pushed onto the execution queue if any of these three conditions are met: (1) the result field of the daily document for the analytic is empty (no result was previously stored), (2) the value of one of the analytics that it depends on has changed, or (3) new data has arrived in the database since the last calculation. Regarding (2), the `last_updated` timestamps for the analytic will be compared (recursively) with those of its dependencies. For example, if X was last updated at time t_1 and its dependency Y was last updated at time t_2, and $t_2 > t_1$, then X will be recomputed. Regarding (3), we don't have a fool-proof strategy for checking if new data has been inserted for a particular day, however we do have a scheme that works for the usual use case since data is always added in daily batches. We assume that if there are no valid intervals of data from at least one watch or from the timestamps, then all analytics are computed afresh, assuming that new data may have arrived where there was none before.

The result of this process is a queue of analytics command objects ready to be operated on by the next stage of the pipeline.

Constructing Queries

Having assembled a queue of analytic command objects, which are dictionaries with all the options and information needed to execute an analytic, the next step of the scheduler is to decide what queries to make to MongoDB. As discussed under the constraints subheading in the Design section of this chapter (page 17), queries to MongoDB are not very efficient, so we aimed to make as few and as minimal queries as possible. In the simple case, we want all the data frames for each watch from the beginning of a day to the end of a day. We find the time tick value for the start of the requested day, and the tick value for the end of the day, and ask Mongo for all documents with a tick field, `ts`, within that range. Such a query takes the form (Mongo shell script):

```
db.lefthand.find({'ts': {$gte: start_ticks, $lte: end_ticks}})
```

Since all of the analytics requested for a given day will be using the same data, a single query can be issued for each watch. Reading a days' worth of data for all four watches usually takes ≈1-2 minutes. However, sometimes the data needs to be sorted, sometimes we don't need all the fields of each document, and sometimes we only need data that has already been identified as valid, so there are a few ways we can save time when making queries.

First of all, we can use projections. Projections return only certain fields from each document matching a query instead of returning the whole contents of each document. Projections filter out data within documents that doesn't need to be returned in the query. We can use projections by inspecting the queue of analytics commands, aggregating all the fields that will be needed, and only projecting necessary fields in the query; this is possible because the YAML specification format requires data frame fields to be specified. That is, when an analytic is defined on data frames as the input (as opposed to timestamps or other analytics), there is a required setting to declare which fields from the data frames are needed. For example, the analytic called `standard_deviation_acceleration` has the `frame_fields` property set to `['ax', 'ay', 'az']` because it only needs the acceleration readings. By default, MongoDB also returns the `_id` field, but since this field isn't needed for anything we can change this default behaviour and force MongoDB not to return that field. When only a few fields are required, projections cut the amount of data transferred down by a huge margin. This will be discussed more in the chapter on Evaluation and Testing.

Secondly, we can choose to make a sorted or unsorted query. The watch data frames are not stored chronologically on disk. This is one of the problems we investigate in the database optimisation section. Additionally, some analytics don't need the frames to be sorted by time, such as standard deviation. As such, we can make a query for sorted documents only when necessary. The queue of analytics commands is scanned to see if any of the scheduled analytics needs sorted data. If at least one needs sorted data, then the query will be sorted, otherwise the data is requested unsorted, which is computationally a lot cheaper.

Thirdly, we can choose whether or not to filter the raw data frames according to our previously determined intervals of valid data. If we do filter by valid intervals, less data will be returned so the query will be faster. Again, we can scan the queue of analytics commands to see if any of the scheduled analytics require all raw frames or only those marked valid. An analytic might require all data frames if, for example, its purpose is to quantify how much data has been stored. There is a special case where some analytics may require all the frames and some only the valid frames. In this case, the query constructor uses a special strategy: query for all the data from Mongo, but make a filtered copy in memory. Filtering the frames by the known valid time intervals in memory is much faster than making a separate query to Mongo.

In all cases, whether using projections, whether sorted or unsorted, whether filtered or not, only one query needs to be made to Mongo for each watch. This query is made before any of the analytic items are popped from the queue and executed, and the queried data is cached to be used by all of the queued computations.

Executing the Analytics

At this point we have a queue of analytics commands to execute and data will have been read from Mongo and stored in Python data structures in main memory. The next step is to pop each analytic command from the queue, do any required preprocessing on the data, pass the data to the function which computes the analytic, then get the result and store it in the database, updating the `last_updated` field.

At this point we make a clearer distinction between the different kinds of analytics. So far we have not explained the key differences between them. There are 3 basic kinds: analytics defined on data frames from the watches, those defined on usage timestamps, and those defined on other analytics. These kinds correspond respectively to the types used in the YAML files for the `defined_on` field: `'frames'`, `'timestamps'`, and `'other_stats'`.

Executing either of the latter two kinds, `'timestamps'` or `'other_stats'`, is relatively straightforward. In both cases, a query is made either to retrieve the sorted cleaned timestamps or to retrieve the document that has all analytics results for the requested day in it. Both of these kinds of queries, for timestamps or results documents, are inexpensive. Each user only has, at most, a few thousand timestamps, and retrieving a single results document is negligible compared to querying for data frames. Either the timestamps or the daily analytics document (depending on the kind of analytic) is then passed to the function which computes the analytic. That function is the one specified in the YAML configuration files, which are parsed and loaded when the scheduler is first invoked. If no errors are produced by the user-supplied function, the result is stored in Mongo.

The third kind, where the YAML field `defined_on` is set to `'frames'`, is slightly more involved. There are a couple of special cases: (1) if the raw data needs to be filtered for valid frames, it may be filtered as part of the Mongo query itself or filtered in-memory; (2) if the analytic requires synchronised data from all devices, the data cached from the database will be synchronised before being passed to the user-defined function for computing the analytic. Filtering the data in memory or synchronising the data from multiple watches into a series of combined data frames requires making copies of the data in memory. This will be discussed further in the Optimisations section below, as making copies became problematic for some days of data.

Configuring Daily Job

The Unix `cron` utility is employed for scheduling time-based jobs. Data is uploaded to the server and inserted early in the morning, so the analytics scheduler is itself scheduled to run every day at 10am. The cron job is configured like-so:

```
0 10 * * * cd /homes/faisallab_u/kinedmd-dashboard;
           source bin/activate;
           bash run_analytics_daily_7.sh >> /tmp/analytics_daily 2>&1;
```

The `cron` job will first navigate to the directory with the analytics pipeline code, activate the virtual environment where all the Python modules are installed, then request the scheduler to run all analytics for the last 7 days, passing the output to a log file. Analytics are scheduled to be computed for 7 days because sometimes patients upload multiple days' worth of data in one go, so data from a a few days ago may have arrived overnight.

Summary of Execution Scheduling Algorithm

This implementation section on the algorithms that make up the analytics pipeline has been dry and dense. To recap briefly, the operation of the pipeline can be summarised as follows:

1. Identify users, dates, and requested analytics to compute based on arguments supplied to the `compute_analytics_daily` function.
2. For each user-day pair, build a queue consisting only of analytics that need to be computed and in an order that preserves their inter-dependencies.
3. For analytics that require data frames from watches, construct one efficient query to retrieve frames for each watch, caching the results to be reused by all analytics and to save time.

4. Execute the queued analytics computations one by one.
5. Save or update results in a MongoDB document belonging to the user.

Optimisations

We already described in brief some optimisations to speed up querying from MongoDB, namely using projections, only sorting when necessary, and filtering frames in memory instead of making multiple queries. There were other performance boosts achieved by adjusting MongoDB query settings and changing the internal representation of data frames. There was also an investigation into the possibility of migrating to a different database or storage solution to improve performance.

MongoDB Query Options

Batch size is one of the configurable options for MongoDB queries. Batch size is a property of a query cursor in `pymongo`; a cursor is a Python object which acts as an interface to the underlying drivers fetching data from a MongoDB instance. Batch size determines how many documents are returned in each response from the Mongo driver. We varied the batch size to determine if doing so had any effect on query performance.

For the batch size tests we chose a user whose data could fit in main memory, `sg01`, who had ≈9GB of data at the time the tests were conducted. A plain query was made for all the data frame documents for all four watches (unsorted) from `sg01`. The query was repeated 5 times and the average time taken was 404 seconds. We then tried the same query (again repeated 5 times) but with different batch sizes of 500, 1000, 2000, 4000, and 8000. The results can be seen in Figure 3.8. Setting the batch size manually overrides the default internal Mongo heuristic to return up to 16MB worth of documents [23]. In our case, setting the batch size to any of the tested values performs better than the default behaviour. Additionally, a batch size of 2000 saves more time than when using the other values.

The other option to try was changing the cursor type. An ‘exhaust’ cursor should reduce latency by serving batches of query results without waiting for the client making the request to ask for each batch individually and sequentially.

We used the user with the most data, `sg04`, who had 173GB of data, to measure changes in query performance. A plain query was made for all the data frames from all four watches (unsorted) of `sg04`, which took 2.2 hours to complete. Using a batch size of 2000, this came down to 1.9 hours, cutting query time by 14%. With an exhaust cursor, that time went up to 2.0 hours, so the

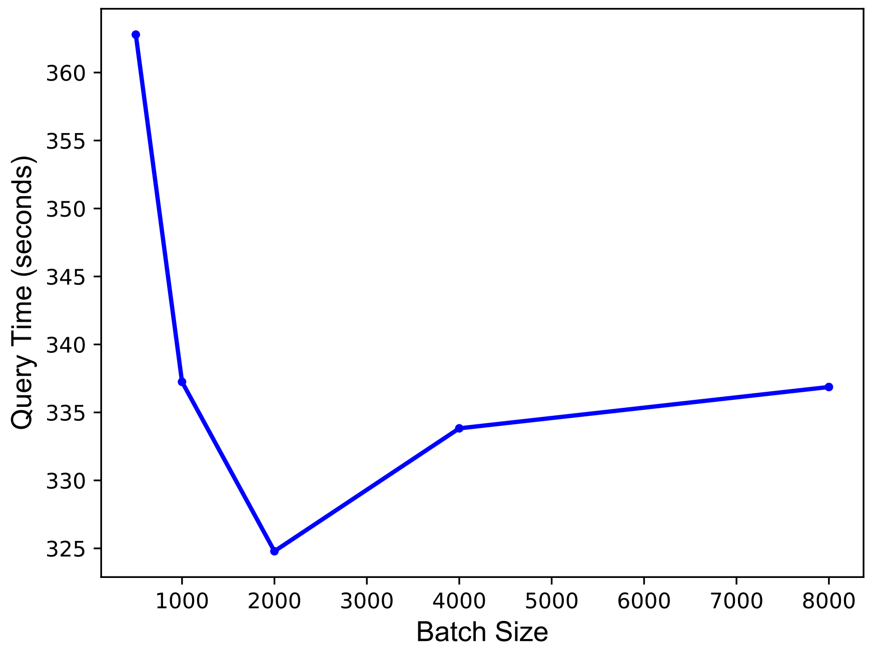


Figure 3.8: Query time in seconds (vertical axis) against batch size (horizontal axis) for batch sizes of 500, 1000, 2000, 4000, and 8000.

exhaust cursor was discarded as a possible optimisation. The overall impact of these optimisations is discussed further in the chapter on Evaluation and Testing.

Internal Representation of Data Frames

Initially, data frames from Mongo queries were stored in Python as a list of dictionaries with the string keys for each field (e.g. `'ts'`, `'ax'`, and `'hr'`). This expands the raw data, which should be 88 bytes, to over 800 bytes! In cases where complicated analytics were running on a day with a lot of data, the pipeline would crash for exceeding the 64GB RAM limit of the server, despite only dealing with one day's worth of data. To remedy this, the list of dictionary objects was replaced with a custom data structure, coined a 'frame store.' A frame store is a dictionary with two keys, `'index'` and `'frames'`. The frames are a list of lists representing frames, so individual frames don't have copies of all the string keys in them, just the values. The index is a mapping from frame field string identifiers (like `'ts'` and `'ax'`) to integers; the integers are the indices where the fields which map to them can be found in the list of values encoding a data frame. Each frame then only occupies about 200 bytes, which is still more than the 88 raw bytes due to the overhead of objects and list structures in Python, but it's 25% of the previous format which is a good improvement. This change cut the RAM usage for a heavy day from exceeding 64GB to peaking at ≈16GB, which includes making multiple copies of the raw data for different stages of analysis. However, loading the data in this format has an impact on read speed. The `pymongo` drivers internally return documents as Python dictionaries, so as each of the millions of frames are returned by the query cursor, they are being converted into lists, a step which takes some extra time.

Replacing MongoDB

Basic sorted range queries in MongoDB are quite slow. Data frames come out from the `pymongo` driver about 5 times slower than when reading them from raw binary files. When it became clear that MongoDB did not perform well when reading large volumes of chronological time series data, I investigated alternative databases and storage solutions with the possibility in mind of making a migration. Mongo was initially chosen by Shuai because it is heralded as a write-intensive database capable of handling big data and because NoSQL databases allow for a unique kind of flexibility in the structure of data. However, all the time series data we have is highly repetitive, unlikely to change, and the way Mongo stores documents on disk is not at all optimal for the kinds of simple range queries we want to run. Instead of fragmenting the data frames into many small files, we want them to be all stored contiguously in sorted day-sized blocks.

Many papers which attempt to benchmark the performance of MongoDB in comparison with popular SQL solutions claim that MongoDB is superior, however several companies still find themselves abandoning MongoDB for mature SQL databases. One study found that Oracle was slower than MongoDB at inserting, deleting, and updating, especially when a table had millions of entries [24]. Another found that MongoDB performed better than Microsoft SQL Server for all queries except aggregate queries (count, sum, and average) [25]. Another study suggests that NoSQL databases like MongoDB are better adapted for distributed environments [26], which is not how Mongo is being used in our setup. These studies have to be taken with a pinch of salt because they don't use a dataset like ours.

On the other hand, some companies advocate moving back to SQL after bad experiences with MongoDB. Shippable had scaling problems as Mongo would be offline for 4 hours to rebuild indexes and they saved lots of disk space by moving to PostgreSQL [27]. The Guardian wanted more tooling for database management than was available for MongoDB, so they migrated to PostgreSQL, a very mature SQL database which also supports JSON objects, which made it relatively easy to transition from MongoBD [28].

At this point it seemed like a good idea to look for database solutions that were optimised for structured read-intensive time series data. A few possibilities were considered but quickly dismissed: Riak-TS is a NoSQL time series database which didn't support macOS (used for development), OpenTSDB is a time series database that only has an HTTP interface (no Python drivers), and InfluxDB is also a purpose-built time series database but optimised for high-speed ingest rather than read-intensive use. The reasons given for not investigating these options further are not definitive, however given the limited time in which to undertake such an investigation some simplifying shortcuts had to be made, so I was looking for a solution which was compatible with macOS, optimised for reading time series data, and which came with a Python driver.

Reading papers and blogs cannot ultimately grant sufficient insight into how different solutions would perform with our data and our hardware, so the decision was made to do some performance tests on PostgreSQL [29] with the TimescaleDB extension [30] which optimises the database for time series data. This seemed like the most promising combination worth investigating further.

A clean install of PostgreSQL with the TimescaleDB plugin was setup on the central server alongside the MongoDB instance. An authoritative batch of synthetic data for a couple of fake users was generated and inserted separately into MongoDB and PostgreSQL so they had identical data. Each user had enough data to occupy over 6GB on disk. A basic unsorted read query was performed to fetch all the data for each user from each database. To make a fair use case comparison, the `psycopg2` Python module was used for querying PostgreSQL from the Python benchmarking scripts, just as the `pymongo` Python driver was used for tests on the MongoDB instance.

PostgreSQL is slow. It performs basic range queries on time series data even slower than MongoDB. MongoDB took 434 seconds to read all the data but PostgreSQL took 542. The reads were each performed 5 times and the mean was taken. PostgresSQL is slower than MongoDB.

Since the quest for a database with better performance was running out of time, alternative data storage solutions were considered, namely (1) using raw binary files and (2) using the Apache Parquet column-oriented compressed binary file format [31]. It quickly became clear that reading sorted binary files (or .parquet files) would be considerably faster than querying data from a database, but adopting a simple storage format like these instead of using a full database would come with significant drawbacks. It would be necessary to replace many of the features included with a database: access routines for inserting, updating, deleting, and querying data; a transaction system to handle parallel queries, validation and backup routines, etc... such a task would have been beyond the scope of this project.

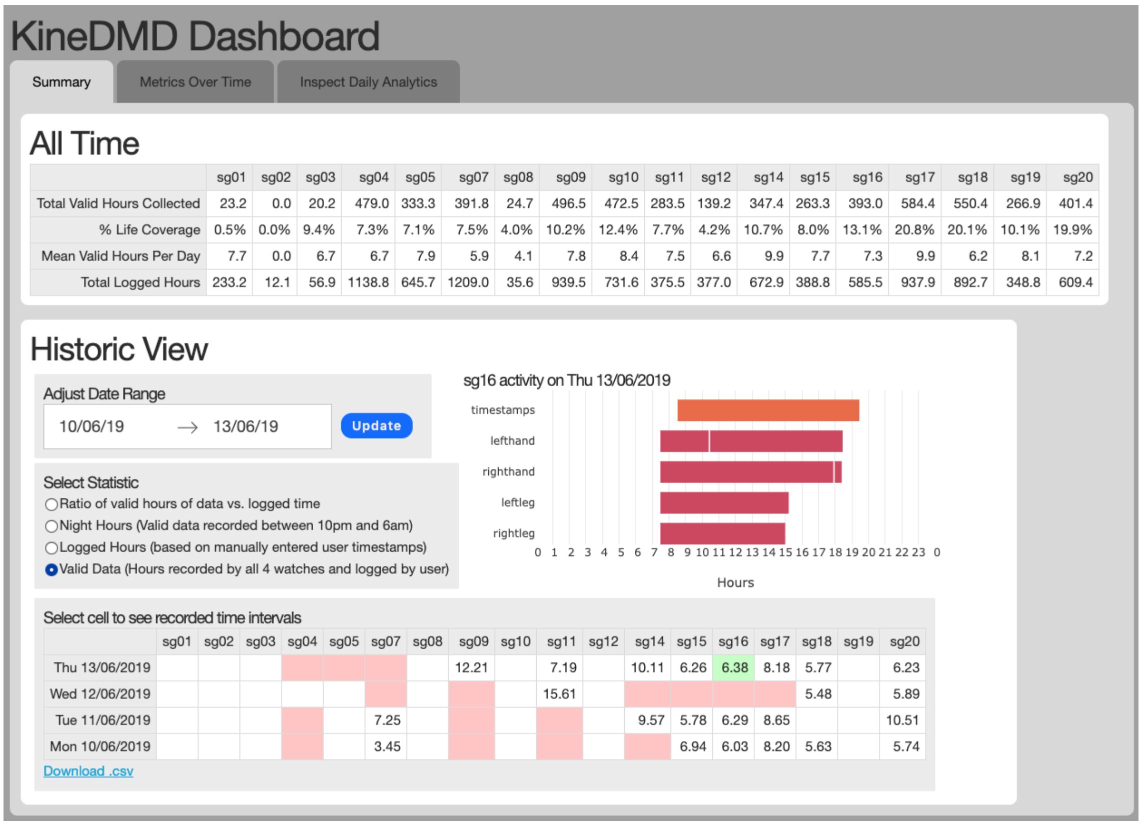
In the end, MongoDB remained. It would have taken too long to fully investigate alternatives or to build a custom solution, it would have taken time to replace the pipeline that had already largely been written to work with MongoDB, and there would be a risk that a new system would have other unforeseen drawbacks. As such, it seemed best to make MongoDB work as well as possible.

Chapter Summary

This chapter has been a trek through the setup, design, and implementation of the Python analytics pipeline which forms the computational backbone of the dashboard. We have looked at the hierarchical format for specifying new analytics in a human-friendly format, how raw data is cleaned and synchronised prior to analytics computations, how analytics are organised by the execution scheduler and how database queries are constructed and optimised, how results are stored in MongoDB, and whether it might be worth migrating to a different storage solution. With a relatively thorough understanding of how analytics are handled under the bonnet, we now turn to the dashboard itself.

4 Analytics Dashboard

The dashboard is the key deliverable of this project. The dashboard serves two main roles: firstly, to display day-to-day information relevant to non-experts supervising the clinical trials and, secondly, to facilitate inspection and visualisation of any analytics computed by the pipeline. It was developed iteratively in tandem with the analytics pipeline but is presented here separately for clarity. The landing page of the application is displayed in Figure 4.1 below; each component will be presented in more detail throughout the chapter. This chapter will cover the choice of tools and frameworks, the features of the dashboard and how they were implemented, and the deployment strategy for the web application.



KineDMD Dashboard

Summary | Metrics Over Time | Inspect Daily Analytics

All Time

	sg01	sg02	sg03	sg04	sg05	sg07	sg08	sg09	sg10	sg11	sg12	sg14	sg15	sg16	sg17	sg18	sg19	sg20
Total Valid Hours Collected	23.2	0.0	20.2	479.0	333.3	391.8	24.7	496.5	472.5	283.5	139.2	347.4	263.3	393.0	584.4	550.4	266.9	401.4
% Life Coverage	0.5%	0.0%	9.4%	7.3%	7.1%	7.5%	4.0%	10.2%	12.4%	7.7%	4.2%	10.7%	8.0%	13.1%	20.8%	20.1%	10.1%	19.9%
Mean Valid Hours Per Day	7.7	0.0	6.7	6.7	7.9	5.9	4.1	7.8	8.4	7.5	6.6	9.9	7.7	7.3	9.9	6.2	8.1	7.2
Total Logged Hours	233.2	12.1	56.9	1138.8	645.7	1209.0	35.6	939.5	731.6	375.5	377.0	672.9	388.8	585.5	937.9	892.7	348.8	609.4

Historic View

Adjust Date Range

10/06/19 → 13/06/19 Update

Select Statistic

○ Ratio of valid hours of data vs. logged time
○ Night Hours (Valid data recorded between 10pm and 6am)
○ Logged Hours (based on manually entered user timestamps)
● Valid Data (Hours recorded by all 4 watches and logged by user)

Select cell to see recorded time intervals

	sg01	sg02	sg03	sg04	sg05	sg07	sg08	sg09	sg10	sg11	sg12	sg14	sg15	sg16	sg17	sg18	sg19	sg20
Thu 13/06/2019				🟥	🟥	🟥		12.21		7.19		10.11	6.26	6.38	8.18	5.77		6.23
Wed 12/06/2019						🟥		🟥		15.61		🟥	🟥	🟥	🟥	5.48		5.89
Tue 11/06/2019				🟥		7.25		🟥		🟥		9.57	5.78	6.29	8.65			10.51
Mon 10/06/2019				🟥		3.45		🟥		🟥		🟥	6.94	6.03	8.20	5.63		5.74

Download .csv

Figure 4.1: Landing page of the dashboard web application displayed in Chrome.

4.1 Setup and Design

The dashboard will have interactive tables and data visualisations. A variety of potential front-end technologies were considered to meet these needs: CanvasJS for HTML5 JavaScript charts [32], Bokeh [33] for rendering visualisations in Python, Bowtie [34] for writing dashboard web applications in Python, and others. Without time to try all of them, I opted to use the Dash framework by Plotly [35].

Dash is a Python framework for building analytical web applications. It is built on Facebook's React.js framework [36], Plotly's own graphing library plotly.js [37], and the Python web microframework Flask [38]; it is like a Python wrapper for React with a rich bundle of graphing modules. Dash was chosen so that the whole analytics dashboard, back and front, can be written in Python and because Dash seemed more mature than the other similar Python frameworks. As with the analytics pipeline discussed in the previous chapter, keeping the implementation in Python makes it easier for other team members to edit and maintain the code later.

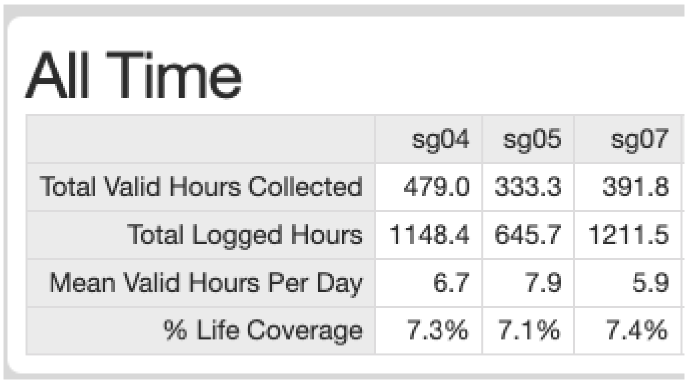
The dashboard application is hosted on the central Apollo server because the application needs access to the analytics results stored on the server. Since Dash applications are written in Python, the `pymongo` driver can be used, as with the pipeline, to query data from the server's local MongoDB instance.

4.2 Features and Implementation

The interface evolved gradually through discussion with and feedback from relevant team members. It began as a single page application with a plain HTML table showing numbers calculated for each patient for the last seven days, and ended with interactive graphs and multiple pages with modular components for user-defined analytics. The application has three main views: (1) the summary tab with interactive tables of daily and all-time analytics, (2) the graphing tab for visualising numerical analytics calculated over the duration of the trial, and (3) an inspection tab to see user-defined visualisations of individual analytics for certain users on certain days. We now explore the design and implementation of each of these views, however in less technical detail than the analytics pipeline because most front-end programming simply involves using existing frameworks and making lots of small tweaks to appearance and behaviour which would be tedious to catalogue.

Summary View

The two most important features requested from the beginning were to have a weekly view of what data was collected by which patients and to have summary statistics for each patient from across the course of the whole clinical trial. The first table displayed on the summary page contains 'all-time' analytics. An excerpt of this table is shown in Figure 4.2.



All Time

	sg04	sg05	sg07
Total Valid Hours Collected	479.0	333.3	391.8
Total Logged Hours	1148.4	645.7	1211.5
Mean Valid Hours Per Day	6.7	7.9	5.9
% Life Coverage	7.3%	7.1%	7.4%

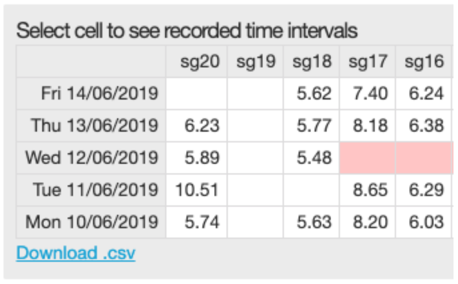
Figure 4.2: Screenshot of the all-time analytics table on the Summary page of the dashboard. Each column is a different patient and each row is a different analytic.

In Figure 4.2 there are 4 different analytics; these are all metrics of interest requested by clinicians and researchers. 'Total Valid Hours Collected' is the total duration of valid data frames recorded according to the validation scheme described in pipeline Implementation section (page 21). These numbers are produced by using native Mongo aggregation operations on the `daily` collection of each user, in this case using the `$sum` operator on the `actual_hours` field. 'Total Logged Hours' is the time collected according the usage timestamps logged by the user. 'Mean Valid Hours Per Day' is self-explanatory. '% Life Coverage' is the ratio of the total number of valid hours of data that were collected to the total number of hours that could have been collected throughout the trial - that is:

$$\frac{\text{valid hours recorded}}{24 \times \text{no. of days participating in trial}}$$

The YAML configuration file for these analytics has an option to specify a string format, `dashboard_format`. For example, the 'Total Valid Hours' analytic has this option set to `'%.1f'` to show one decimal place.

The second table on the Summary page provides a per patient per day view of different analytics. A snippet is shown in Figure 4.3. Each column of the table is for a different patient and each row is for a different day. Each cell contains the value of an analytic chosen from a radio button menu (shown in Figure 4.4).

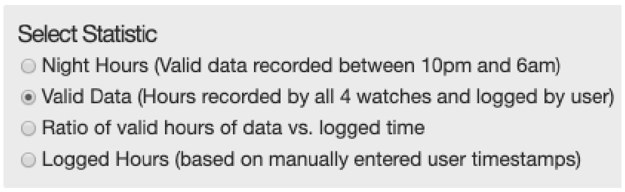


Select cell to see recorded time intervals

	sg20	sg19	sg18	sg17	sg16
Fri 14/06/2019			5.62	7.40	6.24
Thu 13/06/2019	6.23		5.77	8.18	6.38
Wed 12/06/2019	5.89		5.48	🟥	🟥
Tue 11/06/2019	10.51			8.65	6.29
Mon 10/06/2019	5.74		5.63	8.20	6.03

Download .csv

Figure 4.3: Screenshot of the daily per-patient analytics table on the Summary page of the dashboard.



Select Statistic

- Night Hours (Valid data recorded between 10pm and 6am)
- [✓ Valid Data (Hours recorded by all 4 watches and logged by user)]
- Ratio of valid hours of data vs. logged time
- Logged Hours (based on manually entered user timestamps)

Figure 4.4: Screenshot of the radio button menu on the Summary page of the dashboard to select which analytic to display in each cell of the table.

The options on the radio button menu are populated from the YAML configuration file for daily analytics. Any analytic which has a `dashboard_name` field will appear in this list; the dashboard uses the same configuration files as the analytics pipeline itself.

Initially, the table was a plain HTML table, but this couldn't be hooked into the Dash callback system to make it interactive. I tried using the 'Data Table' component developed by Dash instead, which provides interactive and editable tables with row and column selection, but found it had a fatal limitation. It only supported conditional formatting with a special 'language' of basic arithmetic and logic operators; it wasn't possible to supply a Python function for conditional styling of table cells. This was needed to highlight cells red to flag potential problems to investigate; these alerts could not be programmed in the limited formatting language. So, I decided to implement a custom interactive table in React.js and convert it to a Dash-compatible Python component using their provided compilation tool chain.

Each cell of this table shows the pre-computed result of the selected analytic for that patient on that day. This table is also interactive; clicking a cell brings up a time chart to see the activity of that patient for that day. An example of one of these graphs is shown in Figure 4.5.

The graph shown in Figure 4.5 depicts the time intervals during which the user logged that they were wearing watches (the orange timestamps bar) and the time intervals for which data frames were recorded by each watch. The small gaps or blips in the watch data indicate data that was lost in transmission between the watches and the laptop base stations. The graph was programmed in Dash with the plotly graph library Gantt chart template. Dash automatically converts plotly graphs created in Python into JavaScript and HTML5.

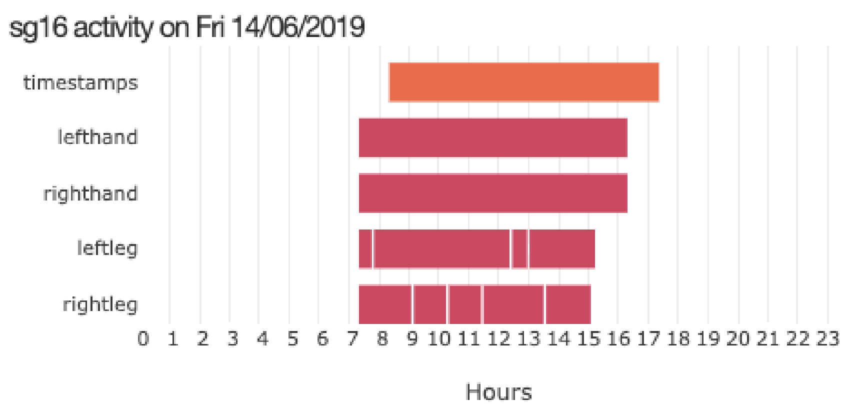


Figure 4.5: Screenshot of the time graph displayed for a patient-day pair when selected in the daily table.

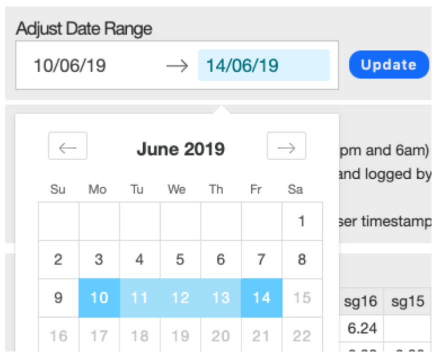


Figure 4.6: Screenshot of the dash date picker component used to select a date range for which to display analytics.

There is also a date selection component to enable browsing of historic data. The default range is automatically set to the last 7 days as the general use case is to check recent patient activity. The date selection widget is a ready-made Dash component, shown in use in Figure 4.6.

The cells highlighted in red correspond to warnings that may need to be investigated further. Specifically, cells are highlighted in red when there is a discrepancy between the duration of logged usage time and actual valid data collected. An example of the user activity corresponding to one of these red cells is shown in Figure 4.7. In that case, data from the right hand watch was missing entirely and data from the left hand watch was dropping a lot of data - these could be indicative of a hardware problem, especially if the same pattern occurs multiple days in a row. These alerts are to help clinicians and relevant personnel to quickly identify potential problems with the data collection process and to take a prompt course of action to resolve any issues.

Altogether the table for daily analytics appears as shown in Figure 4.8. In summary, this view displays recent analytics for all patients, allows users to inspect each day of patients activity, and highlights potential problem cases. There is also a button to download the table data in Comma Separated Variable (CSV) format for convenience.

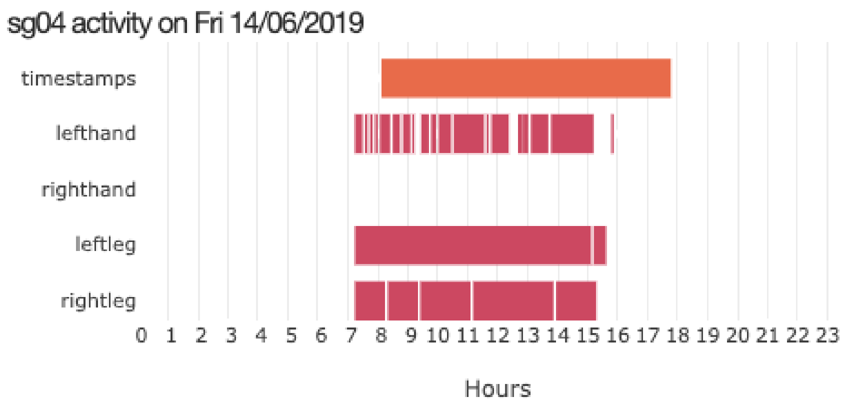
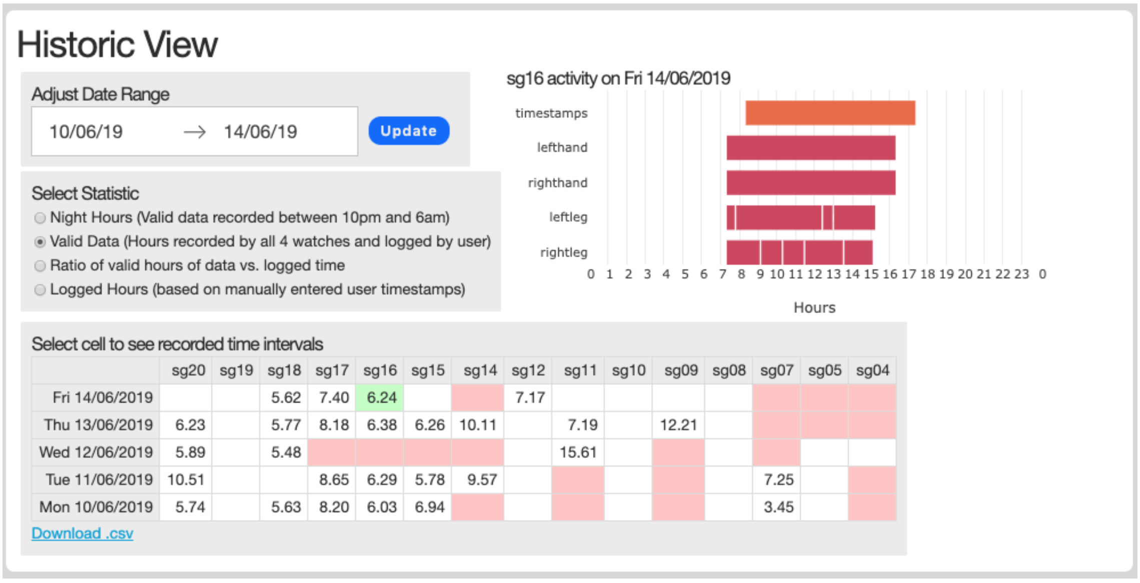


Figure 4.7: Screenshot of the time graph displayed for a patient-day pair which has a red-coloured cell table as a warning.


Historic View

Adjust Date Range

10/06/19 → 14/06/19 Update

Select Statistic

- Night Hours (Valid data recorded between 10pm and 6am)
- [✓ Valid Data (Hours recorded by all 4 watches and logged by user)]
- Ratio of valid hours of data vs. logged time
- Logged Hours (based on manually entered user timestamps)

Select cell to see recorded time intervals

	sg20	sg19	sg18	sg17	sg16	sg15	sg14	sg12	sg11	sg10	sg09	sg08	sg07	sg05	sg04
Fri 14/06/2019			5.62	7.40	🟩 6.24		🟥	7.17					🟥	🟥	🟥
Thu 13/06/2019	6.23		5.77	8.18	6.38	6.26	10.11		7.19		12.21		🟥	🟥	🟥
Wed 12/06/2019	5.89		5.48	🟥	🟥	🟥	🟥		15.61		🟥		🟥		
Tue 11/06/2019	10.51			8.65	6.29	5.78	9.57		🟥		🟥		7.25		🟥
Mon 10/06/2019	5.74		5.63	8.20	6.03	6.94	🟥		🟥		🟥		3.45		🟥

Download .csv

Figure 4.8: Screenshot of the 'Historic View' daily table with the activity inspection graph selected for a specific user.

Graph View

The second tab of the web application is the graph view, which renders scatter plots of analytics with numerical values across the course of a patient's history. If an analytic has been defined per device then the results from each device will be displayed in different colours, as in the case of `standard_deviation_acceleration`, otherwise a single colour is used, as with `actual_hours`. These two use cases are shown in Figure 4.9.

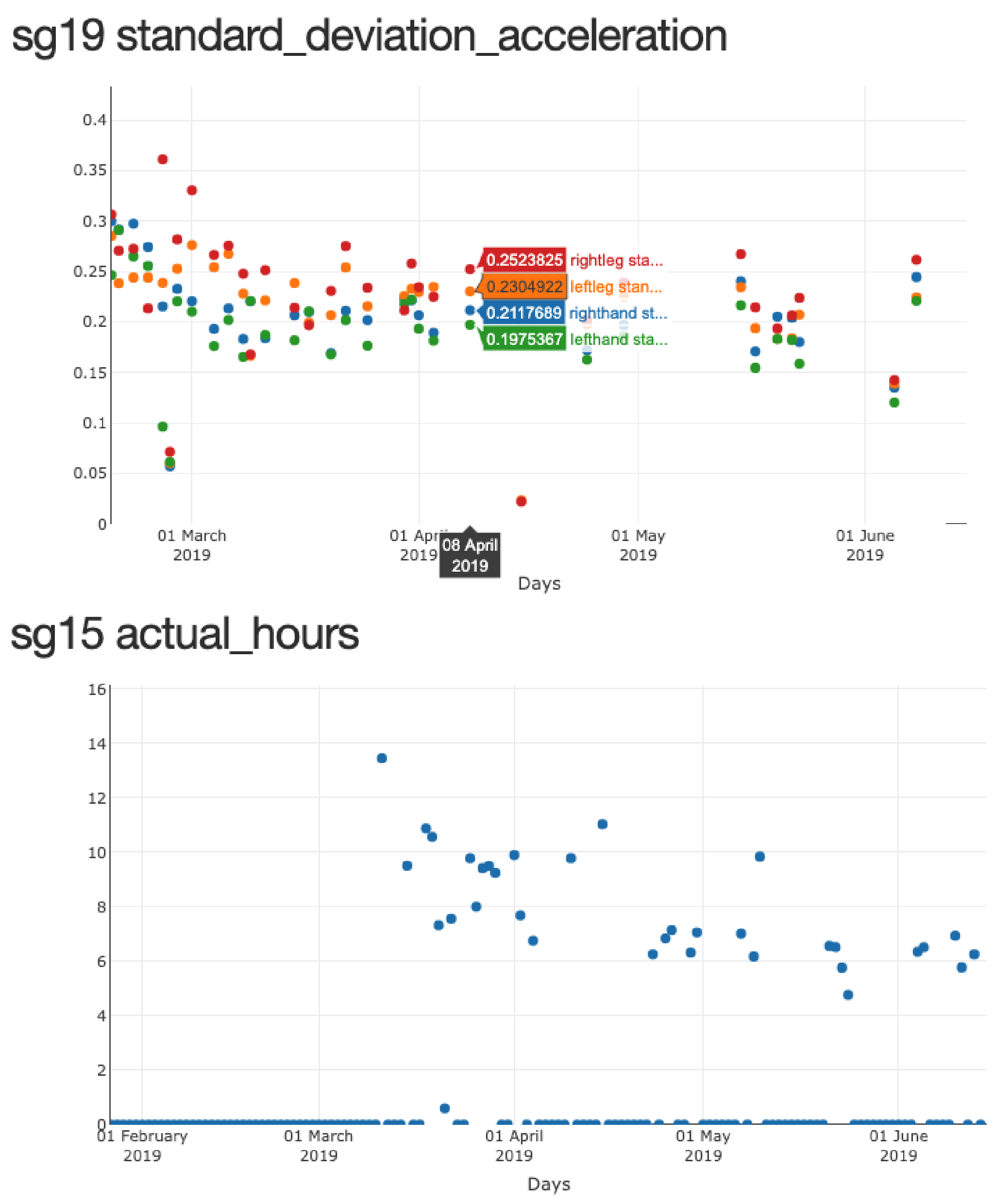
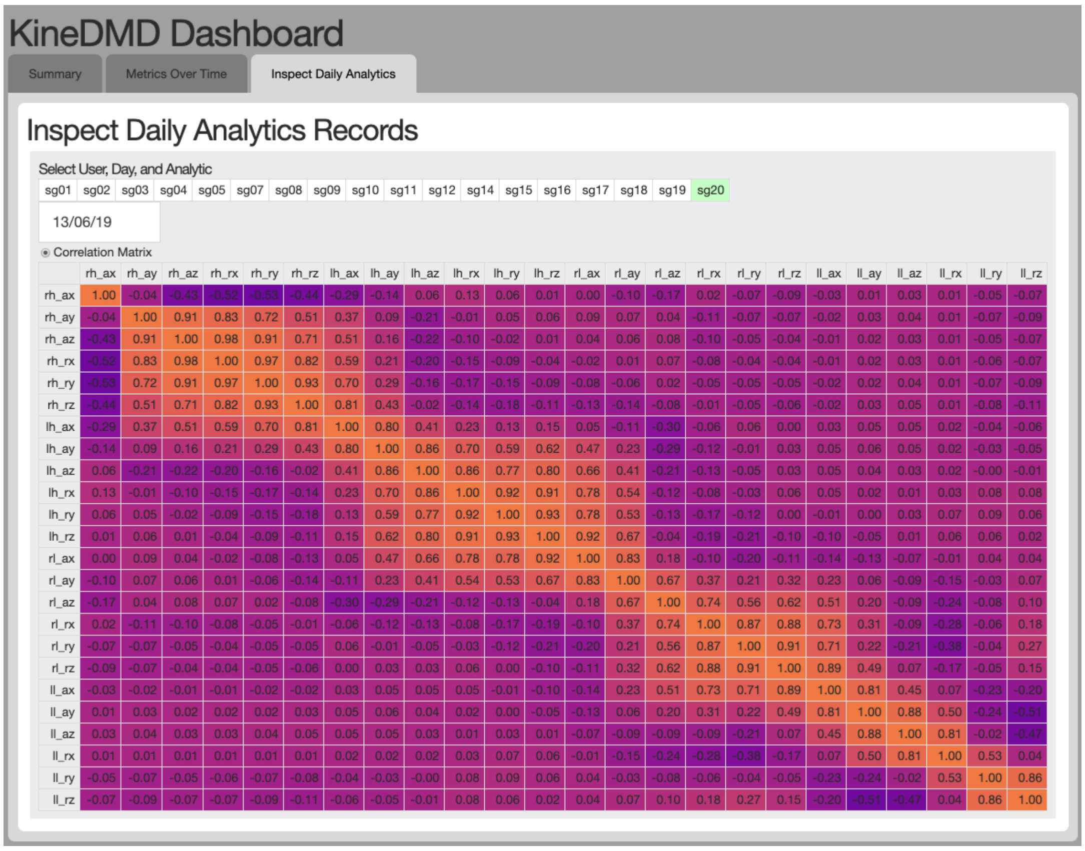


Figure 4.9: Screenshots of scatter graphs for a per-device numerical analytic (top) and a single-valued daily metric (bottom). Labels attached to the points at one moment appear when the user hovers over those points.

Inspection View

This view is for inspecting analytics for a specific user and day which have a user-defined rendering function. The interface allows a user, day, and analytic to be selected and renders the result according to the provided display function. User-made rendering functions are defined in Python using the core Dash components (including the suite of Plotly graphs) and Python HTML objects. An example of this is the correlation matrix shown in Figure 4.10. The function which renders the coloured table takes the result of the analytic computation as an input and returns the descendents of an HTML `<div>` element as the output.


KineDMD Dashboard
Summary
Metrics Over Time
Inspect Daily Analytics
Inspect Daily Analytics Records
Select User, Day, and Analytic
sg01 sg02 sg03 sg04 sg05 sg07 sg08 sg09 sg10 sg11 sg12 sg14 sg15 sg16 sg17 sg18 sg19 sg20
13/06/19
● Correlation Matrix

	rh_ax	rh_ay	rh_az	rh_rx	rh_ry	rh_rz	lh_ax	lh_ay	lh_az	lh_rx	lh_ry	lh_rz	rl_ax	rl_ay	rl_az	rl_rx	rl_ry	rl_rz	ll_ax	ll_ay	ll_az	ll_rx	ll_ry	ll_rz
rh_ax	1.00	-0.04	-0.43	-0.52	-0.53	-0.44	-0.29	-0.14	0.06	0.13	0.06	0.01	0.00	-0.10	-0.17	0.02	-0.07	-0.09	-0.03	0.01	0.03	0.01	-0.05	-0.07
rh_ay	-0.04	1.00	0.91	0.83	0.72	0.51	0.37	0.09	-0.21	-0.01	0.05	0.06	0.09	0.07	0.04	-0.11	-0.07	-0.07	-0.02	0.03	0.04	0.01	-0.07	-0.09
rh_az	-0.43	0.91	1.00	0.98	0.91	0.71	0.51	0.16	-0.22	-0.10	-0.02	0.01	0.04	0.06	0.08	-0.10	-0.05	-0.04	-0.01	0.02	0.03	0.01	-0.05	-0.07
rh_rx	-0.52	0.83	0.98	1.00	0.97	0.82	0.59	0.21	-0.20	-0.15	-0.09	-0.04	-0.02	0.01	0.07	-0.08	-0.04	-0.04	-0.01	0.02	0.03	0.01	-0.06	-0.07
rh_ry	-0.53	0.72	0.91	0.97	1.00	0.93	0.70	0.29	-0.16	-0.17	-0.15	-0.09	-0.08	-0.06	0.02	-0.05	-0.05	-0.05	-0.02	0.02	0.04	0.01	-0.07	-0.09
rh_rz	-0.44	0.51	0.71	0.82	0.93	1.00	0.81	0.43	-0.02	-0.14	-0.18	-0.11	-0.13	-0.14	-0.08	-0.01	-0.05	-0.06	-0.02	0.03	0.05	0.01	-0.08	-0.11
lh_ax	-0.29	0.37	0.51	0.59	0.70	0.81	1.00	0.80	0.41	0.23	0.13	0.15	0.05	-0.11	-0.30	-0.06	0.06	0.00	0.03	0.05	0.05	0.02	-0.04	-0.06
lh_ay	-0.14	0.09	0.16	0.21	0.29	0.43	0.80	1.00	0.86	0.70	0.59	0.62	0.47	0.23	-0.29	-0.12	-0.01	0.03	0.05	0.06	0.05	0.02	-0.03	-0.05
lh_az	0.06	-0.21	-0.22	-0.20	-0.16	-0.02	0.41	0.86	1.00	0.86	0.77	0.80	0.66	0.41	-0.21	-0.13	-0.05	0.03	0.05	0.04	0.03	0.02	-0.00	-0.01
lh_rx	0.13	-0.01	-0.10	-0.15	-0.17	-0.14	0.23	0.70	0.86	1.00	0.92	0.91	0.78	0.54	-0.12	-0.08	-0.03	0.06	0.05	0.02	0.01	0.03	0.08	0.08
lh_ry	0.06	0.05	-0.02	-0.09	-0.15	-0.18	0.13	0.59	0.77	0.92	1.00	0.93	0.78	0.53	-0.13	-0.17	-0.12	0.00	-0.01	0.00	0.03	0.07	0.09	0.06
lh_rz	0.01	0.06	0.01	-0.04	-0.09	-0.11	0.15	0.62	0.80	0.91	0.93	1.00	0.92	0.67	-0.04	-0.19	-0.21	-0.10	-0.10	-0.05	0.01	0.06	0.06	0.02
rl_ax	0.00	0.09	0.04	-0.02	-0.08	-0.13	0.05	0.47	0.66	0.78	0.78	0.92	1.00	0.83	0.18	-0.10	-0.20	-0.11	-0.14	-0.13	-0.07	-0.01	0.04	0.04
rl_ay	-0.10	0.07	0.06	0.01	-0.06	-0.14	-0.11	0.23	0.41	0.54	0.53	0.67	0.83	1.00	0.67	0.37	0.21	0.32	0.23	0.06	-0.09	-0.15	-0.03	0.07
rl_az	-0.17	0.04	0.08	0.07	0.02	-0.08	-0.30	-0.29	-0.21	-0.12	-0.13	-0.04	0.18	0.67	1.00	0.74	0.56	0.62	0.51	0.20	-0.09	-0.24	-0.08	0.10
rl_rx	0.02	-0.11	-0.10	-0.08	-0.05	-0.01	-0.06	-0.12	-0.13	-0.08	-0.17	-0.19	-0.10	0.37	0.74	1.00	0.87	0.88	0.73	0.31	-0.09	-0.28	-0.06	0.18
rl_ry	-0.07	-0.07	-0.05	-0.04	-0.05	-0.05	0.06	-0.01	-0.05	-0.03	-0.12	-0.21	-0.20	0.21	0.56	0.87	1.00	0.91	0.71	0.22	-0.21	-0.38	-0.04	0.27
rl_rz	-0.09	-0.07	-0.04	-0.04	-0.05	-0.06	0.00	0.03	0.03	0.06	0.00	-0.10	-0.11	0.32	0.62	0.88	0.91	1.00	0.89	0.49	0.07	-0.17	-0.05	0.15
ll_ax	-0.03	-0.02	-0.01	-0.01	-0.02	-0.02	0.03	0.05	0.05	0.05	-0.01	-0.10	-0.14	0.23	0.51	0.73	0.71	0.89	1.00	0.81	0.45	0.07	-0.23	-0.20
ll_ay	0.01	0.03	0.02	0.02	0.02	0.03	0.05	0.06	0.04	0.02	0.00	-0.05	-0.13	0.06	0.20	0.31	0.22	0.49	0.81	1.00	0.88	0.50	-0.24	-0.51
ll_az	0.03	0.04	0.03	0.03	0.04	0.05	0.05	0.05	0.03	0.01	0.03	0.01	-0.07	-0.09	-0.09	-0.09	-0.21	0.07	0.45	0.88	1.00	0.81	-0.02	-0.47
ll_rx	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.03	0.07	0.06	-0.01	-0.15	-0.24	-0.28	-0.38	-0.17	0.07	0.50	0.81	1.00	0.53	0.04
ll_ry	-0.05	-0.07	-0.05	-0.06	-0.07	-0.08	-0.04	-0.03	-0.00	0.08	0.09	0.06	0.04	-0.03	-0.08	-0.06	-0.04	-0.05	-0.23	-0.24	-0.02	0.53	1.00	0.86
ll_rz	-0.07	-0.09	-0.07	-0.07	-0.09	-0.11	-0.06	-0.05	-0.01	0.08	0.06	0.02	0.04	0.07	0.10	0.18	0.27	0.15	-0.20	-0.51	-0.47	0.04	0.86	1.00

Figure 4.10: Screenshot of the inspection page with the correlation matrix selected for a certain user on a selected day.

A custom render function can be specified in the YAML file by setting the `html_render_function` field with the string name of the rendering function, which should be defined in the same file pointed to by the `path` field for the analytic computation function. For example, the definition for the correlation matrix has these fields:

```yaml
path: 'analytics/correlation_matrix.py'
function: 'correlation_matrix'
dashboard_name: 'Correlation Matrix'
html_render_function: 'correlation_matrix_render'
```

Data Caching

All of the views presented (summary tables, graphs, custom render) pull some data from MongoDB. Some of those queries can take a noticeable time to complete, enough to make the interface feel slightly sluggish, so queried data is cached on an hourly basis. The cache was implemented as a Python decorator, so functions which retrieve data can be annotated with `@cache(timeout=<time in seconds>)` to avoid making fresh queries to Mongo every time. Since new daily analytics for the previous day of uploaded data are usually finished by 11am, the dashboard will have the analytics corresponding to the previous day of uploaded data available by mid-day at the latest.

4.3 Deployment and Configuration

By default, Dash apps uses the Flask application server under the hood, however Flask is recommended for use only as a development server, not for production. Instead we use Gunicorn [39], which is a Python WSGI HTTP server for Unix which automatically runs multiple instances of an application and routes traffic to different workers for load-balancing. Gunicorn is configured to run on `localhost` port 8060 (`127.0.0.1:8060`) with 2 workers. We do not expect to have many concurrent users on the dashboard and each worker can serve many users, however having at least 2 workers allows the dashboard to be restarted gracefully by waiting for one worker to restart before restarting the other. Gunicorn is also supplied with an SSL certificate for the Apollo server to support HTTPS for secure communication with clients.

With Gunicorn running multiple instances of the dashboard on a local address, an NGINX server was setup as a reverse proxy. NGINX is used for 3 things: (1) authentication, (2) exposing the dashboard to the remote clients through a URL, and (3) forcing HTTP requests to use HTTPS instead.

NGINX authentication was configured according to an online tutorial [40]. First, a new user is created with an encrypted password which are stored together in `/etc/nginx/.htpasswd`. The password is encrypted with the `openssl` command line utility by calling `openssl passwd -apr1`. Authentication is then activated in the NGINX configuration file, `/etc/nginx/nginx.conf`, where the password file is supplied as an option. The dashboard is exposed through the URL `https://apollo.doc.ic.ac.uk/dashboard/`. The authentication configured only applies to the `/dashboard/` location. Requests to that location are forwarded to the local Gunicorn servers and their replies are returned as if from `/dashboard/` (this is the reverse proxy). The final job for NGINX is to redirect requests made via HTTP (port 80) to use HTTPS instead (port 443), ensuring that data exchanged between client and server is always secure.

This setup with NGINX, Gunicorn, and the dash application is depicted in Figure 4.11. The final step to configure the dashboard was to register the Gunicorn application with the Linux `systemd` service management system. This will make the

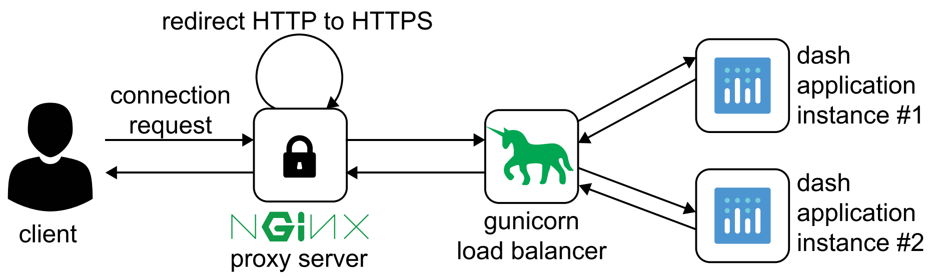


Figure 4.11: Dashboard network configuration. Arrows represent web requests and responses.

application start automatically when the system is rebooted and allows the server to be shut down or restarted easily. A new service file, `kinedmd-dashboard.service` was created in `/etc/systemd/system/`. The service file contains the commands to start, stop, and restart the dashboard application with Gunicorn. With the dashboard web application registered with the service manager, it can be controlled with these simple commands:

```
systemctl start kinedmd-dashboard.service
systemctl stop kinedmd-dashboard.service
systemctl restart kinedmd-dashboard.service
```

Chapter Summary

We've given an overview of the three main pages of the dashboard application, demonstrating how it can be used to inspect patient data and visualise analytics. The dashboard allows clinical trial supervisors to get quick and easy insights into the ongoing data collection process, to identify potential problems which may require further investigation, and the dashboard enables them to draw conclusions about the overall progress of the study in terms of patient participation over time, total hours of data collected, and other metrics of interest.

The dashboard is deployed behind a proxy server which forces connections to use HTTPS and which requires authentication for security; multiple instances of the dashboard are running under a load balancing server to provide graceful restarts and a smooth experience for multiple users; and the dashboard is integrated with the Linux system service manager for automatic initialisation on boot and convenient management.

5 Evaluation and Testing

Having toured the design and implementation of the analytics pipeline and the user-facing dashboard, we now take a step back to reflect on the merits and shortcomings of these systems and of the methodology by which the systems were produced. In the wider context of the *KineDMD* study, the motivation for this engineering project was to give clinicians and researchers high-level insights into the ongoing clinical trials and to facilitate the computation of analytics which could support the search for new digital biomarkers. In this relatively short chapter, we expose the general qualities of the analytics pipeline and the dashboard with respect to their utility in furthering the *KineDMD* research. As these two systems (pipeline and dashboard) serve different roles, we treat them each in turn.

5.1 Analytics Pipeline

We begin with a technical evaluation: surveying the limited formal testing procedures employed throughout development, quantifying the overall performance of the analytics pipeline, then estimating the capacity of the system. We will conclude with some limitations of the pipeline.

Testing Correctness

One of the greatest mistakes of this project was in not establishing regression unit tests early on. The `pytest` framework was added with some simple tests towards the end of the project, more as a formality than for their utility. The neglect towards incorporating tests from the beginning was rationalised (incorrectly) in a couple of ways.

Firstly, when a system is still in a stage of radical evolution, maintaining automated tests can create an unhelpful overhead as the relationship between tests and code has to be constantly updated. The specification for the system was changing a lot, so tests were not written early to avoid that overhead. Secondly, with limited time to get a working product it can be beneficial to focus on adding functionality rather than safeguarding existing features from the unintended side effects of ongoing changes to the system. This seemed like a reasonable trade-off at the time.

In hindsight, automated regression tests should have been introduced earlier. Once the functional specification for the pipeline had been established, the short time left was invested in adding features instead of writing rigorous tests. This backfired when the pipeline had to be rewritten to use a different internal representation of data frames to avoid running out of memory; it is easy to make mistakes when applying that kind of pervasive refactoring and there was no battery of unit tests to reveal unintended side effects caused by the structural changes. Instead of relying on comprehensive automated tests, outputs from previous iterations of the pipeline were compared manually with outputs from the new version by travelling back in time with `git`. Mongo documents written by one version of the pipeline would be put aside (copied to a text file), then a previous version of the pipeline would be loaded with the same patient data, then a sample of results documents from each version were compared to check that the pipeline behaved as before. This was tedious and unsustainable but still yielded confidence that nothing had been broken.

The wider downside of not having automated tests, beyond slowing down development at some junctures, is that it becomes much harder to transfer responsibility for the code to others. People unfamiliar with the system will not be able to make significant or complicated changes with confidence. If there were a full-orbed suite of tests, it would be much easier to know if something had been accidentally broken and therefore to gain confidence when making more involved modifications.

Almost all of the testing performed on the pipeline was manual and very incremental. Every time something was added or changed, the output was inspected for correctness manually. The Python interactive shell was used very often to try different inputs and edge cases for all the functions. The Python graphing library `pyplot` was used to render countless one-off graphs to check that motion data, timestamps, and time intervals were being processed correctly. None of these small and gradual checks were documented or preserved. They seemed so natural and seamless to carry out but, looking back in hindsight, these undocumented manual checks are unfit as evidence for the correctness of the system. The only guarantee of functional correctness is, ultimately, the author's claim that the different calculations and implementations are correct.

Performance and Capacity

In Shuai's report [17], he claims the central server can support about 70 patients uploading data every day. At that limit, the server CPU will be churning at 100% for 24 hours per day just receiving, processing, and storing data. This number is arrived at by solving the inequality:

$$\text{(Upload Time For N Users)} + \text{(Processing Time Per User)} * N \leq 24$$

The upload time is non-linear because the server can download data from a finite number of patients simultaneously, so if there are more patients than the server can handle at once there need to be multiple transmission windows. Shuai assumes a lower bound 5 Mbit/s upload rate from patients' homes and a 100 Mbit/s download rate on the central server, thus the server can receive data from 20 patients simultaneously. Each user is given a conservative 2 hour window to upload data, so the server requires at least 2 hours for every batch of 20 users. The upload time in hours for N users can then be expressed as $2 * \lfloor 1 + N/20 \rfloor$. It takes no more than 15 minutes to unzip, convert, and insert data for one patient. These steps happen sequentially as they require full CPU usage. The above inequality can then be formulated thus (N is the number of users the system can support):

$$2 * \lfloor 1 + N/20 \rfloor + 0.25 * N \leq 24$$

This yields $N \leq 64$, which is slightly less than Shuai's estimate, perhaps because in this formulation the non-linearity of the upload time is made explicit by rounding with the floor function. Now, to quantify what impact the newly introduced analytics system has on this upper limit on the number of users, we need to estimate how long the analytics pipeline takes to compute one day of results for one patient.

All the analytics defined during development (though more may be added) were computed for the whole clinical data set. It took 27.53 hours to compute analytics for 2308 patient days. However, not all days have much data (some don't have any) so we can't use those measurements alone to quantify how long analytics take in the average case. Instead, we measure how long it takes to compute analytics for each of those thousands of days but only take an average of those that take longer than 15 seconds. If they take less than 15 seconds it is because there are no (or not many) data frames stored for that day.

We find the mean time across all patients, for all days which take longer than

15 seconds to operate on, to be 106 seconds per day, based on 927 days. The maximum, however, is 343 seconds. Taking the maximum as indicative of the upper bound time to execute analytics per day, the inequality can be re-written (in hours):

$$2 * \lfloor 1 + N/20 \rfloor + (0.25 + \frac{343}{3600}) * N \leq 24$$

This yields $N < 53$. So, without the addition of further daily analytics, the server can support at most 53 users.

Regarding the optimisations that we spent so long tweaking and investigating, they pale in significance against the fundamental constraint of running on a single machine. Using queries with projections and an optimised batch size can cut query time by up to 40%. However, even if we were to reduce the time taken to query data and compute analytics by a factor of 10, this would only allow the server to support a handful more users than it will at present.

Another Master's student, Rohan Padmanabhan, has developed a new data collection pipeline on AWS that can process data for 20 users in 3 minutes instead of 5 hours. As discussed in the background, this system is unlikely to replace the current setup. It became apparent in making comparing the current system with that implementation that, to allow the server to support more users, the existing decompression and insertion scripts could be written in a language that performs better than Python. However, rebuilding the data collection system was beyond the scope of this project.

The steps taken to ensure reasonable memory usage were worthwhile: loading data one day at a time, using a more compact internal representation in Python, and only reading data from Mongo that is needed for some requested analytics. For days with the largest amount of data and running the most computationally demanding analytic, the correlation matrix, RAM usage doesn't go above 25% (≈16GB). This means that space-intensive daily analytics can safely be defined.

Overall, the analytics pipeline does the job it needs to do in a manner that is efficient given the constraints of the hardware and the performance of MongoDB queries. For the current number of users, and for the number of patients likely to join the trial in the near future, the analytics system works reliably well.

Limitations of the Pipeline

1. Given the constraint of running on a single machine, the system can only support about 50 active patients, which is about 15 less than it can support with the current setup without the analytics jobs running.

2. There is no automated high-level use of CPU time. When analytics are requested, they will be executed regardless of what else is happening on the server at the time. For example, if a user logs in to the server and runs a new analytic on the whole data set while the server is busy receiving data or inserting new data into MongoDB, it is unknown exactly what will happen. To run analytics over old historic data it the responsibility of the user to decide when to start the computations so as not to overlap with a mission critical operation.

3. The pipeline employs no explicit parallelisation, for example computing analytics while loading the next batch of data. MongoDB has very high CPU utilisation during queries, so it is not obvious that this would be a beneficial kind of optimisation to explore. However, regardless of these preliminary considerations no attempts were made to introduce parallelisation.

4. The pipeline depends quite intimately on MongoDB as a database. It would not be straightforward to replace MongoDB. This could be construed as a structural weakness in the design of the pipeline, especially given the likelihood that MongoDB is not an optimal long-term storage solution.

5. The pipeline is not at all general. Assumptions specific to the particular data from the clinical trial are hard-coded into the system in different ways. Adapting it to operate on other data sources would require major changes.

6. The programmatic interface for specifying new analytics is only designed to work with Python scripts. It would be more useful to support other languages, for example by using a generic message passing interface, so that the analytics scheduler could organise analytics computations specified in a variety of languages. Some of the researchers in the *KineDMD* team write a lot of code in MATLAB.

7. A comprehensive set of automated tests has not been included in the code repository, which increases the risk and difficulty of making major changes.

8. There is no direct support for running machine learning models, though it would not be hard to extract analytics results from the database. As such, feature extraction analytics can be defined as part of the pipeline, then the extracted features will be stored in MongoDB and the documents in there can be retrieved as training inputs. Since many useful pre-processing steps have been built into the pipeline, such as determining time intervals of valid

data and synchronising data from different watches, it would be useful to have an interface that can stream batches of training data. Alternatively, a good solution would be to use the analytics pipeline to export pre-processed raw binary data for training, which can then be moved relatively easily to a machine with GPUs or other hardware optimised for training on big data.

5.2 Dashboard

The dashboard does what it was created to do: it gives non-experts and relevant medical personnel ease-of-access to high-level information which augments their ability to oversee and learn from the clinical trials. The computational limit of the underlying analytics pipeline to cope with a larger numbers of participating patients is, from this vantage point, irrelevant. The needs of the clinicians and researchers involved have been met. There are, however, some limitations to address and feedback to review, which we will present here.

Limitations of the Dashboard

From a technical standpoint, the Dash framework was helpful for getting an initial version of the dashboard working, but was difficult to tweak in subtle ways to improve the user experience. The framework was chosen so that the whole network stack could be written in Python, but in the end custom components had to be created in React.js to achieve the desired interface behaviour. Integrating React components with Dash is more cumbersome than using the React framework directly. Dash provides simplification at the expense of control.

Regarding functionality, there are a few kinks in the interface. For example, the Dash date picker component doesn't remember the last selected date, so it shows the current month every time it is opened. This is impossible to fix without writing a new custom component. The graph view doesn't have an option to display multiple graphs at once or to overlay data from different patients, again because selecting multiple patients would require writing custom components in React and converting them into Dash components.

The alerts system (table cells highlighted in red) could be more informative, with annotations like 'righthand data missing' or 'no timestamps.' In particular, there are some cases where there is a very tiny amount of data for one day, and no timestamps, which appears as an 'alert' cell, but when the cell is clicked to inspect the activity graph, it is empty. This is because the duration of data to display on the graph is smaller than one pixel, thus invisible. Cases like these should be reported with greater clarity so as not to confuse an observer or give the impression that there is a problem when there isn't.

User Feedback

Dr. Valeria Ricotti is the project manager for the *KineDMD* study and an Honorary Lecturer at the Great Ormond Street Institute of Child Health. She is a paediatrician and has been involved in designing gene therapy clinical trials for Duchenne muscular dystrophy; she has also completed a higher degree focusing on characterising the evolving natural history of DMD. She was asked, "How might the dashboard help you fulfil your role in the clinical trials?" to which she replied:

> "The dashboard is excellent. The interface is aesthetically pleasing, and very user-friendly.
>
> The dashboard provides all the information which is necessary for the clinical team to monitor how the subjects are using their watches, and if any issue occurs, we can immediately contact the families or flag it to the team at Imperial.
>
> In addition, it has very nice features like providing the correlation matrix, standard deviation of acceleration/rotation and the breakdown of night and day hours usage.
> ...
> Thank you for your hard work!"

- Dr. Valeria Ricotti, 14/06/2019

6 Conclusions

6.1 Achievements

Clinicians and researchers now have a tool to monitor the *KineDMD* clinical trials and to run analytics which can guide the ongoing investigation. The analytics pipeline and dashboard, despite their respective limitations, directly improve the quality of the wider investigation. Since the dashboard can help diagnose problems with data collection, it can prevent the loss of patient data and therefore increase the value of their precious time participating in the trial. These systems together allow stakeholders to make strategic decisions about the progress and evolution of the clinical trials and to gain a better understanding the collected data.

6.2 Future Work

This report describes systems which are limited in scope to the narrow domain of a specific clinical trial and research group, however similar studies are already underway elsewhere in the world and could benefit from similar supporting technology. Thus, there is an opportunity to create generalised solutions that can be adapted for many different research situations.

The next phase of the *KineDMD* study will be to use the vast data set to extract biomarkers, which will involve machine learning. It would be beneficial to have an interface to the clinical data which can export only selected fields of valid data frames in a compact format (binary or parquet), and which can be streamed in batches over a network for consumption by other machines for hardware accelerated training.

The most important next step for the *KineDMD* study will be to develop a more robust system for separating valid data from invalid data, that is, a method to detect automatically when a watch is being worn. Finally, the analytics pipeline will need to be updated or re-written to work in a distributed environment, which will be necessary to support larger studies.

6.3 Final Remarks

My chief aim in undertaking a project with the *KineDMD* team was to use the abilities and opportunities made available to me at Imperial College London to have a positive and concrete humanitarian impact - to make life really and truly better for someone in some tangible way. A few people are already using the systems which I have created, and they will keep doing so as long as the clinical trials continue, which may be many years. Patients, parents, and future families affected by Duchenne muscular dystrophy will also benefit in a small and indirect way from this project, which for me is a great joy and personal triumph. Even though it is not perfect, I am glad that the final product has a practical use and will not quickly be relegated to the abyss of mediocre theoretical discoveries.

Bibliography

[1] DMD HUB, "Kinedmd." https://dmdhub.org/trials/kinedmd/. Accessed: 2019-01-23.

[2] Duchenne Research Fund, "Kinedmd study: developing an activity monitoring biomarker." https://www.duchenne.org.uk/project/activity-monitoring-biomarker/, 2018. Accessed: 2019-01-23.

[3] R. P. Lisak, D. D. Truong, W. Carroll, and R. Bhidayasiri, *International Neurology - A Clinical Approach*, pp. 222–222. New York: John Wiley & Sons, 2011.

[4] E. Mazzone, S. Messina, G. Vasco, *et al.*, "Reliability of the north star ambulatory assessment in a multicentric setting," *Neuromuscular Disorders*, vol. 19, no. 7, pp. 458 – 461, 2009.

[5] P. L. Enright, "The six-minute walk test," *Respiratory Care*, vol. 48, no. 8, pp. 783–785, 2003.

[6] E. Mazzone, D. Martinelli, A. Berardinelli, *et al.*, "North star ambulatory assessment, 6-minute walk test and timed items in ambulant boys with duchenne muscular dystrophy," *Neuromuscular Disorders*, vol. 20, no. 11, pp. 712–716, 2010.

[7] Y. Hathout, H. Seol, M. H. J. Han, A. Zhang, K. J. Brown, and E. P. Hoffman, "Clinical utility of serum biomarkers in duchenne muscular dystrophy," *Clinical Proteomics*, vol. 13, p. 9, Apr 2016.

[8] M. A. Anaya-Segura, F. A. García-Martínez, L. Á. Montes-Almanza, *et al.*, "Non-invasive biomarkers for duchenne muscular dystrophy and carrier detection," *Molecules*, vol. 20, no. 6, pp. 11154–11172, 2015.

[9] R. Willcocks, I. Arpan, S. Forbes, *et al.*, "Longitudinal measurements of mri-t2 in boys with duchenne muscular dystrophy: Effects of age and disease progression," *Neuromuscular Disorders*, vol. 24, no. 5, pp. 393 – 401, 2014.

[10] K. G. Hollingsworth, P. Garrood, M. Eagle, *et al.*, "Magnetic resonance imaging in duchenne muscular dystrophy: longitudinal assessment of natural history over 18 months," *Muscle & nerve*, vol. 48, no. 4, pp. 586–588, 2013.

[11] "Actimyo." www.institut-myologie.org/imotion/actimyo/. Accessed: 2019-01-25.

[12] A.-G. Le Moing, A. M. Seferian, A. Moraux, *et al.*, "A movement monitor based on magneto-inertial sensors for non-ambulant patients with duchenne muscular dystrophy: a pilot study in controlled environment," *PloS one*, vol. 11, no. 6, p. e0156696, 2016.

[13] "aparito." `www.aparito.com/`. Accessed: 2019-01-25.

[14] Duchenne UK, "Aparito." `https://www.duchenneuk.org/aparito`, 2017. (Accessed: 2019-06-10).

[15] M. El-Gohary, S. Pearson, J. McNames, *et al.*, "Continuous monitoring of turning in patients with movement disability," *Sensors*, vol. 14, no. 1, pp. 356–369, 2014.

[16] F. B. Horak and M. Mancini, "Objective biomarkers of balance and gait for parkinson's disease using body-worn sensors," *Movement Disorders*, vol. 28, no. 11, pp. 1544–1551, 2013.

[17] S. Zeng and A. Faisal, "In-the-wild motion monitoring system for duchenne muscular dystrophy patients," September 2018.

[18] The GitHub Blog, "New year, new github: Announcing unlimited free private repos and unified enterprise offering." `https://github.blog/2019-01-07-new-year-new-github/`. Accessed: 2019-06-05.

[19] PyPI, "virtualenv." `https://pypi.org/project/virtualenv/`. Accessed: 2019-06-05.

[20] pytest, "pytest: helps you write better programs." `https://docs.pytest.org/`. Accessed: 2019-06-05.

[21] MongoDB, Inc., "Pymongo 3.8.0 documentation." `https://api.mongodb.com/python/current/`. Accessed: 2019-06-06.

[22] O. Ben-Kiki, C. Evans, and I. d. Net, "The official yaml web site." `https://yaml.org/`. Accessed: 2019-06-12.

[23] MongoDB, Inc., "Iterate a cursor in the mongo shell — mongodb manual." `https://docs.mongodb.com/manual/tutorial/iterate-a-cursor/`, December 2018. (Accessed: 2019-06-14).

[24] A. Boicea, F. Radulescu, and L. I. Agapin, "Mongodb vs oracle–database comparison," in *2012 third international conference on emerging intelligent data and web technologies*, pp. 330–335, IEEE, 2012.

[25] S. H. Aboutorabi, M. Rezapour, M. Moradi, and N. Ghadiri, "Performance evaluation of sql and mongodb databases for big e-commerce data," in *2015 International Symposium on Computer Science and Software Engineering (CSSE)*, pp. 1–7, IEEE, 2015.

[26] L. Bonnet, A. Laurent, M. Sala, *et al.*, "Reduce, you say: What nosql can do for data aggregation and bi in large repositories," in *2011 22nd International Workshop on Database and Expert Systems Applications*, pp. 483–488, IEEE, 2011.

[27] A. Cavale, "Why we moved from nosql mongodb to postgresql." http://blog.shippable.com/why-we-moved-from-nosql-mongodb-to-postgressql, November 2017. Accessed: 2019-06-14.

[28] The Guardian, "Bye bye mongo, hello postgres." https://www.theguardian.com/info/2018/nov/30/bye-bye-mongo-hello-postgres. Accessed: 2019-06-14.

[29] The PostgreSQL Global Development Group, "Postgresql: The world's most advanced open source database." https://www.postgresql.org/. Accessed: 2019-06-14.

[30] Timescale, Inc., "Time-series data simplified." https://www.timescale.com/. Accessed: 2019-06-14.

[31] Apache Software Foundation, "Apache parquet." https://parquet.apache.org/documentation/latest/. Accessed: 2019-06-14.

[32] Fenopix, Inc., "Beautiful html5 javascript charts." https://canvasjs.com/. Accessed: 2019-06-14.

[33] Anaconda and Bokeh Contributors, "Welcome to bokeh — bokeh 1.2.0 documentation." https://bokeh.pydata.org/en/latest/. Accessed: 2019-06-14.

[34] J. Kvam, "Github - jwkvam/bowtie: Create a dashboard with python!." https://github.com/jwkvam/bowtie. Accessed: 2019-06-14.

[35] Plotly, "Dash by plotly." https://plot.ly/products/dash/. Accessed: 2019-06-14.

[36] Facebook Inc., "React – a javascript library for building user interfaces." https://reactjs.org/. Accessed: 2019-06-14.

[37] Plotly, "plotly.js | javascript graphing library." https://plot.ly/javascript/. Accessed: 2019-06-14.

[38] A. Ronacher, "Flask (a python microframework)." http://flask.pocoo.org/. Accessed: 2019-06-14.

[39] "Gunicorn - python wsgi http server for unix." https://gunicorn.org/. Accessed: 2019-05-12.

[40] J. Ellingwood, "How to set up password authentication with nginx on ubuntu 14.04." https://www.digitalocean.com/community/tutorials/how-to-set-up-password-authentication-with-nginx-on-ubuntu-14-04, August 2015. Accessed: 2019-05-30.

All the 'hand drawn' doodles of people throughout this report were traced from free stock photos by the author of the report, so they are derivative works from sources which did not require credit for personal use.