Jacob Moxham

Centralised and Decentralised Clouds: managing processing in the context of sensitive data

Computer Science Tripos – Part II

St John's College

May 1, 2019

Proforma

Name:	**Jacob Moxham**
College:	**St John's College**
Project Title:	**Centralised and Decentralised Clouds: managing processing in the context of sensitive data**
Examination:	**Computer Science Tripos – Part II, July 2019**
Word Count:	**1587**[1]
Project Originator:	**Jacob Moxham**
Supervisor:	**Dr Jatinder Singh**

Original Aims of the Project

To build a middleware which supports policies which allow granular control of the location of processing for HTTP requests and of the data retrieved from database queries.

Work Completed

All success criteria have been met and a couple of optional extensions completed. I have built a middleware allowing nodes to specify which HTTP requests they can compute results for and whether the response is raw data or the actual result and requesters to specify their preferred location of processing. I have also implemented a database wrapper for MySQL allowing transforms, which must be applied to columns before use, to be specified for groups of requester identities.

Special Difficulties

None.

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Declaration

I, Jacob Moxham of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

List of Figures

Chapter 1

Introduction

This project concerns providing the user control over the location of data processing as well as what data can be processed. The recent surge in cloud computing has raised many questions about the privacy of user data as well as operational concerns about the feasibility of centralising all computation. I introduce a Policy Aware Middleware (PAM) which supports user defined policies governing whether data processing is done locally or remotely, and providing granular control of what data is processed. This overcomes the challenges of providing location transparency to the user as well as being flexible enough to allow a range of privacy mechanisms from the literature to be implemented. My results show that this is achieved with minimal overhead and provide examples where exploiting the policies can provide privacy guarantees and performance gains.

1.1 Motivation

"Cloud computing is the delivery of computing services — servers, storage, databases, networking, software, analytics and more — over the Internet" [7]. Centralising computation in this way has become increasingly common in recent years. It provides more efficient multiplexing of resources between clients, provides elasticity to demand and supports virtualisation which allows users to operate in familiar environments. Additionally it spreads the initial costs of setting up a local data centre as well as the operational costs, capital requirirements and maintenance costs.

However, centralising computation wastes the computing potential of all of the other devices in the network, a quantative study [18] has shown that edge computing can reduce both poweer consumption and latency for mobile applications over both WiFi and LTE **ROB COMMENT: numbers here? latency gain from local computation, power costs? MY REPLY: I added the reference to try and address this point, the paper has some nice graphs but i'm unsure precisely what numbers to quote.**. Central computation requires all data which needs to be processed to be sent from its source, to a data centre, this has a latency and power cost as well as requiring users to trust the cloud provider with the data. A survey of recent cloud computing

developments [15] indicates that the majority of cloud providers do not currently guarantee security or privacy levels in their service level agreements (SLAs), this is supported by a review of cloud computing law [3] published in 2013, the SLAs are highly specific regarding performance and uptime guarantees but far more vague regarding privacy and data protection. However, the providers are still bound by law to employ good practice, this means that there is not only an incentive for end users to protect their data, and stop it from being centralised when possible, but also for cloud providers to offer means for you to do so in order to limit their liability.

Edge computing refers to using more of the devices in the network in order to move some computation closer to the user, including at the source of the data and at intermediate points such as routers. This reduces bandwidth requirements for the network, the latency for the end user, and also the resource requirements of the central data centres. A series of trade-offs is presented regarding where we process data, what data we send over the network and how we keep our data secure. It has been suggested that we will move into a new era where many designs make use of edge computing to distribute more processing and give users more control of their data [13].

The majority of work of edge computing aims to boost performance by moving some computation closer to users in order to reduce round-trip times and make use of unused compute power on intermediate nodes within the network[1]. However, whilst increased user privacy is always mentioned as a motivation it is rarely focussed upon even though many distributed mechanisms for ensuring anonymity or privacy exist [20, 24, 12, 22, 8]. There is an opportunity to explore how privacy mechanisms such as these can be supported in an edge centric environment.

1.2 Previous Work

A survey done on open issues in data security and privacy preservation in Edge Computing [29] motivates the need for dynamic switching between mobile computation and offloading to a central server. Two examples are discussed below.

MAUI [9] is a middleware which profiles edge device's power consumption characteristics; the running time and resource requirements of methods; and network characteristics, to inform dynamic offloading from mobile devices to a central server. This is tightly coupled to a specific keyword used to mark offloadable functions for applications written in Microsoft's .NET Common Language Runtime. It focusses on reducing the energy consumption of mobile devices whilst also considering the performance implications of offload.

CloudAware [21] seeks to combine offloading with awareness of the user's context (i.e. the availability of WiFi) in a generic framework. This is illustrated by the example of deciding whether to offload image processing tasks from a mobile device to a more powerful edge

[1]Edge computing differs from traditional Content Delivery Networks in that computation, as well as data is moved closer to the user.

device. It uses a machine learning classifier to predict the future context and uses this to decide whether to offload processing based on execution time and the probability of a result being returned to the mobile device. **ROB COMMENT: add in applications, done I think**

The following middlewares focus on allowing data collection from heterogeneous devices. Usually in an internet of things (IoT) context.

MECA [28] provides an abstraction for collecting data from a range of mobile devices, illustrated through the example of disaster management. Applications can express their data requirements declaritively and then a three layer architecture routes data from mobile devices, through an edge layer where aggregation can occur, to a cloud hosted backend layer which selects edge nodes for different data collection and processing activities. The security and privacy concerns which arise from having all of the data pass through this infrastructure is left as future work.

Mosco [26] supports fine grained access control policies for collecting data from IoT devices. It uses medical data and location as key examples, and supports policies which allow users to set filters, control the granularity of released data, only release data summaries or only release data if it is similar enough to another person's data. This gives users granular control over what data is released, however it requires the processing to be done in two centralised cloud components which we must trust. **ROB COMMENT: perhaps use an example, e.g. the monitoring of the heart problem, at the moment your description is quite vague. What are the challenges? what do the policies look like? Is this similar to what you have done or total different. Again making this clearer would be useful as you make reference to the filters/summaries on p.11. You want the user to understand what these area. I needed to take a peak at the Mosco paper. I have written an alternative paragraph below.**

Mosco [26] seeks to solve the problem of providing fine grained access control policies for collecting data from IoT devices. The examples of user location and self-monitored medical data are used to illustrate the transforms supported by these policies. Filters can be specified which limit the data which we allow an entity to access, for example we may only make our location available for a certain period of time. The granularity of released data can be controlled, this requries us to define the levels of granularity for an attribute, for instance the location of a user to within a certain distance. Similarity policies ensure that data is only released if it is similar enough to another tuple of data, this could be used to only show a user's location to nearby friends rather than everybody. Finally, summary policies allow the aggregation of several data records, this could be used to only report the average heart rate for each day rather than every measurement. Supporting these policies would meet *PAM*'s goal of providing control of what data we allow applications to user, however Mosco relies on two central cloud components to enforce and apply these policies which contradicts *PAM*'s other goal of providing control over the location of data processing.

All of these frameworks neglect to consider the user's preference for what processing is

performed on their devices and what data leaves their devices, even Mosco insists we allow all our data be to sent at least as far as the cloud component. MAUI, CloudAware and other similar frameworks also recreate a lot of the mechanisms required to offload processing. *PAM* seeks to provide a framework which gives the users more direct control over these aspects and allows the kind of offloading and privacy policies supported by the middlewares above to be expressed in one unifying framework.

1.3 The Project

ROB COMMENT: You could put the final paragraph of the first chapter in its own section named "The project" perhaps? Again, you could also summarise your achievements. Briefly outline what you evaluate and what metrics you use? Perhaps you could also say what the challenge is here? what problems will you go on to solve? - what policies to define, how to specify? - the implementation itself? If so, what are the challenges, are there concurrency issues? My response: I have attempted to do this, thoughts?

PAM provides policies which could be used to support mobile offload, the decision could be automated or instead set manually by a user based on their preferences. It provides data policies which are more general than those used in Mosco [26] as they allow arbitrary transforms to database columns. These may require more programmer effort to specify, but allow for a range of privacy preserving mechanisms to be implemented (see §4.2.2) and do not require any centralised cloud components.

This solves the problem of providing a unified framework for expressing computation offload and data privacy mechanisms. The overhead of these policies is quantified through the database query benchmarks and specific examples chosen to illustrate the usecases of the policies. The potential performance improvement in terms of memory and request latency is also illustrated by these examples.

The key challenges in the development of the polcies governing the location of data processing was making them modular and extensible, implementing a dynamic version of these mechanisms also required concurrent update and use of the polciy to be considered. The largest challenges lay in supporting policies allowing data to be transformed before it is used, this required disovering all tables affected by database queries and then creating transformed versions of these tables. Caching these transformed tables also raised the issues of establishing the validity of a transformed table and concurrent queries both attempting to create a transform simultaneously.

Chapter 2

Preparation

Building a middleware from the ground up takes a large amount of preparation to ensure that the interaction of the various components has been carefully considered. In the remainder of this section I first introduce the concept of a middleware. I then explain the requirements of my middleware and discuss my choice of tools used to implement it and my developmental approach. Finally I detail the design of *PAM*.

2.1 Introduction to Middlewares

A middleware lies between the operating system and the application in order to provide additional functionality to the applications developer. In a distributed environment they commonly abstract away some details of network communication in order to provide a simpler interface to the programmer. For example, a middleware may provide authentication of user identities for HTTP requests or log data about database accesses for audit purposes. However middlewares can be much more complex, performing processing on requests to collect data or alter their contents or destination. *PAM* sits between the user and basic HTTP libraries and also provides a database wrapper, this is illustrated in Figure 2.1.

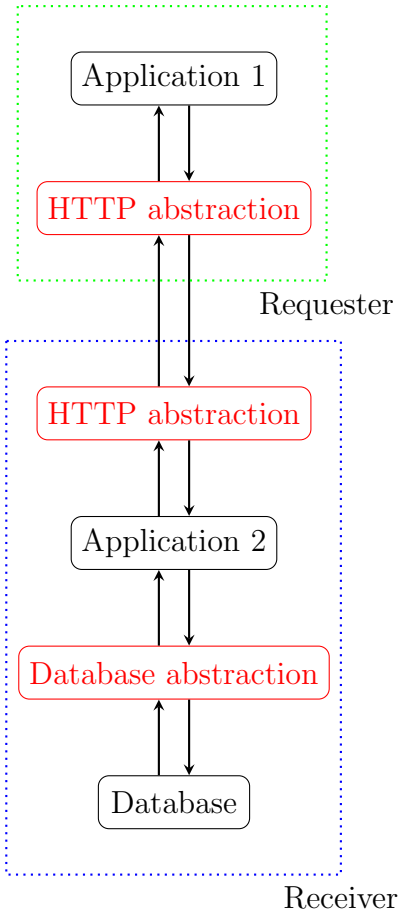


Figure 2.1: Diagram illustrating in red where HTTP and database middlewares can add abstractions for the programmer.

2.2 Requirements Analysis

Before I could begin designing my middleware I first had to expand upon my success criteria in order to establish the requirements of *PAM*.

The first aim of my project was to provide a policy framework allowing granular control over data and processing of data. In order to achieve this I considered what data can be processed and where data is processed separately.

I deemed that data policies should specify, for groups of requester identifiers, which columns of a table should be accessible and whether any transforms should be applied to data before it is made accessible. I considered supporting a range of specific transforms such as the filters and summaries used in Mosco [26] but instead opted for more generic, programmable transforms for this project as it simplified the implementation and allowed a wider range of privacy policies to be implemented as transforms.

Regarding the location of processing the main decision was whether to allow requesters to have final say on where processing occurs. As a core aim is to provide users with more control over where processing is done I opted to allow nodes to specify which requests they can compute a result for in the form of computation policies. This means requests

Requirement	Sections
Data policies specifying arbitrary transforms to be applied to database columns before they are read are supported.	§2.5.3, §3.4, §3.7
Data policies specifying arbitrary rows to be excluded from data are supported.	§2.5.3, §3.4, §3.7
If multiple data policies apply to a user then they are resolved appropriately.	§3.2.1
Computation policies specifying which HTTP requests can be handled and at what level of computation are supported.	§2.5.1, §3.2, §3.5, §3.6
Request policies specifying a preferred location for data processing are supported.	§2.5.2, §3.3, §3.5, §3.6
Some common privacy mechanisms can be implemented as data policies.	§4.2.2
Supports the example of performing image processing locally or in the cloud based on policies.	§2.7.2, §3.8.3
Supports the example of returning no power consumption data, raw power consumption data or a local aggregate based on policies and the identity of the requester.	§2.7.1
Supports the example of applying transforms specified by data polices to data based on the identity of the requester.	§2.7.1
Extension: Caching optimisation for transformed database tables.	§3.7.2, §4.1.2
Extension: Dynamic offloading of computation can be implemented as combinations of request and computation policies.	§3.2.2

Table 2.1: Requirements of the middleware

to merely specify a preference for the processing location in case multiple options exist.

The second aim of the project was to assess the applicability of *PAM* in various scenarios. Many options were considered but each of the chosen examples isolate the main capabilities of the middleware: offloading computation, applying transforms to data and returning raw data or full results based on computation and request policies. This is discussed further in §2.7.

Table 2.1 shows the requirements and the sections detailing how they have been satisfied.

2.3 Choice of Tools

The tools used for this project were chosen based on a combination of familiarity and applicability. The remainder of this section discusses my choices for programming languages, development and testing environment, tools for performance evaluation, libraries, and datasets.

2.3.1 Programming Languages

Go

Go was chosen as the primary langauge for the middleware for a number of reasons:

- I had prior experience developing HTTP request handlers in Go which made creating the example use cases easier.
- It is statically typed which improved code readability and understanding for a project of this size.
- I was familiar with JetBrains' GoLand[1] IDE which has many features to aid development and testing including an extension for Docker integration.
- Go's interfaces allow for flexible designs where multiple implementations can be provided without large code changes. **NOTE: should I add an appendix explaining these interfaces?**
- Chaining middlewares[2] is very simple in Go, this meant I could focus on the unique features of my middleware and rely on existing middlewares for other commonly required functionality. This is described further in §3.6.1. **NOTE: Add some examples with references?**

Go 1.11[3] was used for this implementation.

SQL

In order to implement a database wrapper to support my data policies. I needed to become familiar with a database query language, MySQL 5.7[4] was chosen as its driver is supported by the standard Go `sql` library, it is a common SQL dialect, and SQL is the industry standard in query languages.

Other

For data analysis and graphing Python 3.5.2[5] was used. This was mainly due to my familiarity with matplotlib (version 2.1[6]), the canonical graphing library for Python.

Several simple scripts were written in bash to update dependencies and push docker images to a remote registry.

[1] https://www.jetbrains.com/go/new/
[2] https://medium.com/@chrisgregory_83433/chaining-middleware-in-go-918cfbc5644d
[3] https://golang.org/doc/go1.11
[4] https://dev.mysql.com/downloads/mysql/5.7.html
[5] https://golang.org/doc/go1.11
[6] https://matplotlib.org/2.1.0/

2.3.2 Development

Careful pre-planning and research into the best ways to manage and back-up code was required before implementation could begin. The section below covers my development environment as well as how dependencies were managed and what backup strategy was used.

Development Environment

The IDE I chose to use was JetBrains' GoLand, the other popular choice for Go is Atom[7] which is a universal text editor with a large number of plugins available. I prefer the language specifc features of JetBrains products such as autocomplete, the ability to jump between the declarations and implementations of interfaces and formatting-on-save which would require substantial effort to set up for Go in Atom. For similar reasons JetBrains' Python IDE, PyCharm[8], was used for developing data anlysis scripts.

Containerisation

In order to simulate a network of separate machines locally in my example usecases I used Docker[9], an open source container orchestration tool. It allowed me to create networks of containers which appeared to be isolated from one another. Docker Swarm also allowed me to place resource constraints on individual containers which facilitated performance analysis of examples by simulating different memory and processing power ratios between a central cloud server and edge nodes.

GoLand allowed me to view the logs of containers within it but did not support Swarm and so when running resource constrained analysis I used Kitematic[10], a graphical tool for viewing and managing Docker containers.

Network Simulation

As I ran my tests locally on a simulated network I also had to simulate network latency. Pumba[11] allowed me to add a delay to all egress traffic with a specific jitter and distribution. I used this to simulate communication with a central server hosted on Amazon Web Services in my example use cases.

[7] https://atom.io/
[8] https://www.jetbrains.com/pycharm/
[9] https://www.docker.com/
[10] https://kitematic.com/
[11] https://github.com/alexei-led/pumba

Dependency Management

In order to create concise Docker images for my example use cases it was necessary to have a local copy of all of the dependencies for each example. This allowed static linking and hence a single binary to be included in the final image. To faciltate this I used `dep`[12], a dependency management tool built specifically for Go. Once a folder has been initiated as a `dep` project it contains a `/vendor` directory which contains all of the dependencies, source code, a dependency tree and a readable description of the dependencies.

Version Control and Backup Strategy

Three local`git`[13] repositories were maintained for documents, data analysis and middleware code respectively. Local commits were made regularly, and then pushed to a remote repository hosted on GitHub[14]. The repository of documents was kept private so as to avoid plagiarism.

DockerHub[15] was used to host docker images for my example usecases. This was required in order to use Docker Swarm which was needed in order to impose resource constraints on containers. I tried using the auto-build feature on DockerHub to build new images whenever the middleware code GitHub repository was updated, however this resulted in long build times as the builds were performed on the DockerHub cloud servers. I instead wrote a bash script to build the images locally and push them to DockerHub, this not only resulted in faster build times but allowed me to separate version control of my source code from updating images which meant I could test new code before comitting it.

2.3.3 Performance Evaluation

Go supports benchmarks written in test files which are run multiple times in order to profile CPU and memory usage. I used these to establish the performance overhead of the middleware, particularly the overheads of the databse wrapper. Unfortunately the benchmarks do not report any measure of variance by default and so I had to run them multiple times in order to establish confidence intervals for measurements.

For holistic meauremnts which required network communication I used ApacheBench[16]. This allowed me to send large numbers of concurrent requests and produced a summary statistics for latency as well as a comma separated value (CSV) output which I parsed as part of my data analysis. This tool was also useful for simulating load on a server.

[12]https://github.com/golang/dep
[13]https://git-scm.com/
[14]https://github.com/JacobMoxham
[15]https://hub.docker.com/
[16]https://httpd.apache.org/docs/2.4/programs/ab.html

2.3.4 Libraries

The Go SQL package was used, along with the Go MySQL driver[17], to implement a database wrapper supporting data policies. User SQL queries had to be parsed in order to find the tables and columns which they affected, a Go SQL parsing library, `sqlparser`[18], was used for this.

I used the Go tensorflow library[19] to build the image processing example discussed in §2.7.2 and adapted an example object recognition application[20].

To produce graphs when doing data analysis I used the matplotlib (version 2.1[21]) library for Python due to my familiarity with it and the wide range of graphs it can produce.

2.3.5 Dataset

The examples discussed in §2.7.1 concern power consumption data from IoT home monitoring systems. I used a household power consumption dataset[22] rather than live readings. This had a number of advantages:

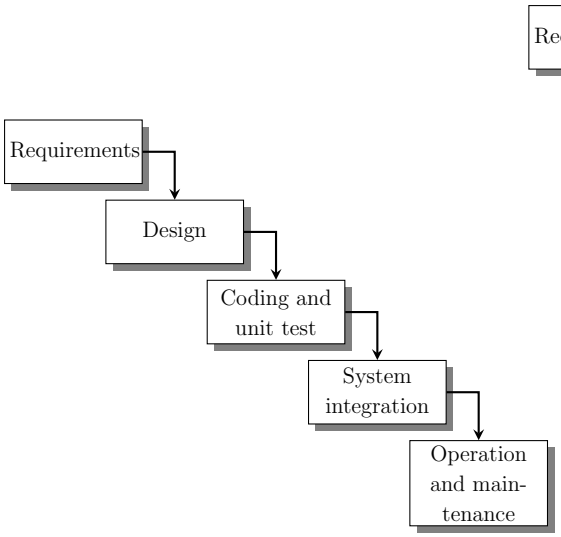
1. When testing the example, results were reproducible.
2. The measurements were not simulated.
3. I did not have to acquire and set up the actual hardware for home monitoring.
4. The database has a large number of measurements[23] which allowed me to benchmark my database wrapper on a wide range of database sizes.

2.4 Developmental approach

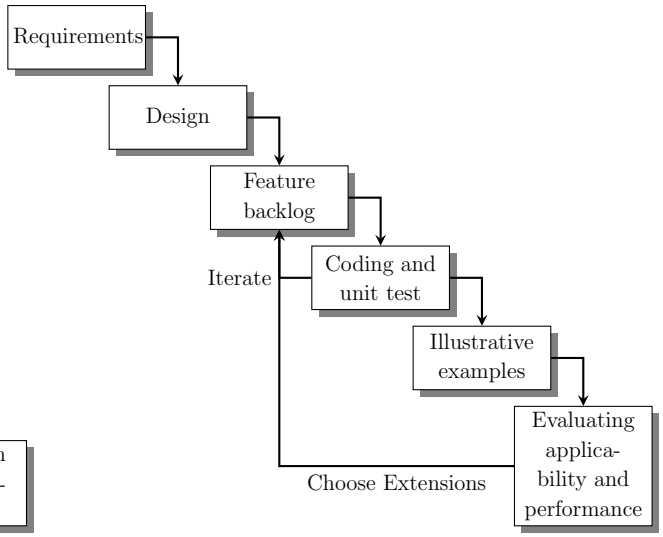
My developmental approach for this project bears elements both of the waterfall methodology and agile, feature driven development.

The waterfall model (Figure 2.2a) is a linear model where each stage must be completed before progressing to the next. As this is a new, prototype, middelware, it is not used in production or part of a larger system and so the "System integration" and "Operation and maintenance" stages were replaced with implementing "Illustrative examples" and "Evaluating applicability and performance".

[17] https://github.com/go-sql-driver/mysql
[18] http://github.com/xwb1989/sqlparser
[19] http://github.com/tensorflow/tensorflow/tensorflow/go
[20] https://github.com/plutov/packagemain/tree/master/04-tensorflow-image-recognition
[21] https://matplotlib.org/2.1.0/
[22] https://data.world/databeats/household-power-consumption
[23] 2,075,259 measurements are included in the dataset.



(a) The waterfall model for development.



(b) My development model.

Feature driven development centers around a backlog of potential features which are chosen one at a time, developed, unit tested and then integrated with the rest of the project. It is part of agile methodology, which is largely team centered, but this specific part is still applicable to an individual project. During the implementation stage I adopted this method as I knew that my middleware was fairly modular and many features could be implemented and unit tested before others were fully complete.

Optional extensions also do not fit nicely into the waterfall model. Upon completing the core parts of the middleware and taking some preliminary measurements, I chose to focus on the extensions which would provide the optimum utility based on the work so far.

A diagram representing my actual developmental approach is shown in Figure 2.2b

2.5 Middleware Design

Before development could begin a detailed plan for the middleware had to be made.

PAM supports three kinds of user-specified policy:

- **Request policies:** are attached to requests and specify:
  - The identity of the requester.
  - Whether all of the data needed to compute the query result is contained within the query.
  - Whether it is preferred to perform the computation locally or remotely.
- **Computation policies:** specify which HTTP request paths can be handled and whether the result of handling them will be the raw data required to compute a result, or a full result.

- **Data Policies:** specify, for groups of requester identities, any columns of database tables which should be excluded from queries, and any transforms which should be applied to data before it is used in a query.

The remainder of this section details how each of these policies is supported by *PAM*. The terms request and response are used to mean requests and responses made through *PAM* whereas HTTP request and HTTP response are used to refer to the underlying primitives used to send messages between nodes which use *PAM*.

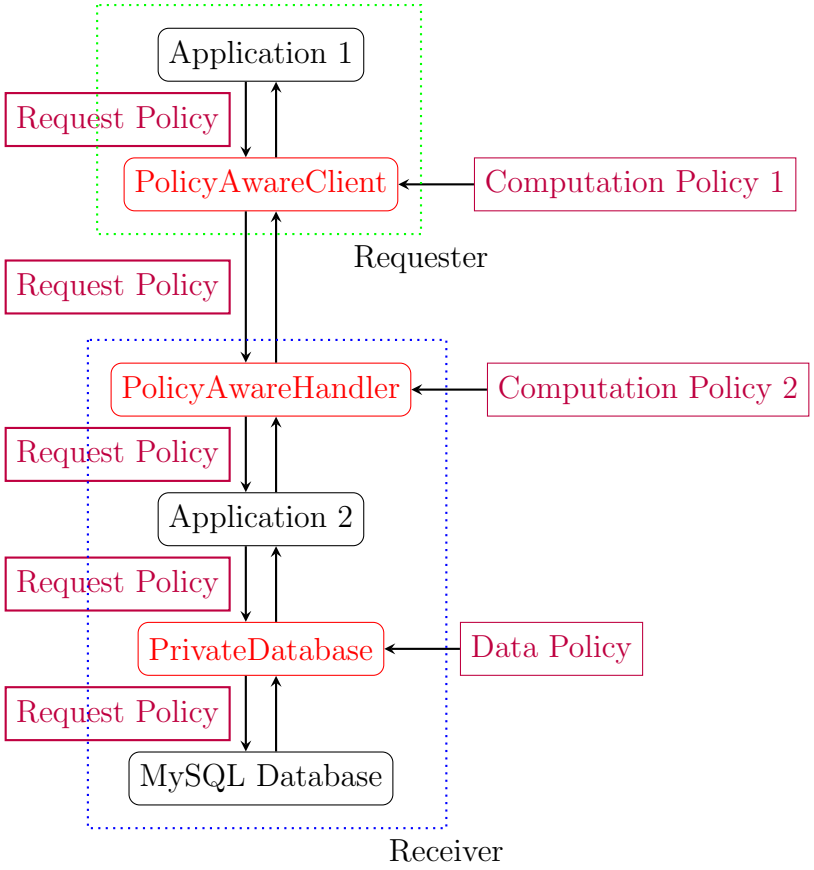


Figure 2.3: Diagram illustrating where *PAM* (shown in red) sits between the user and the Operating System in order to support various policies (shown in purple). **TODO: display request policies more nicely?**

2.5.1 Computation Policies

ROB COMMENT: What might a system look like? do you need a concrete example to help the reader, help describe what nodes are etc.? MY REPLY: example has been addded, thoughts?

In many modern environments it may be preferable to not do local computation even if we have the raw capability to. In the case of IoT this is often due to energy constraints, but we may also want to choose not to do processing because of load or user preference.

An example would be requesting the important locations (i.e. home, work, the gym) from the user's of a specific mobile phone app. If we assume the app stores a portion of the location history of the user it may be that they do not have the spare computational resources to compure the important locations, but they do have the power and bandwidth to send the data over the network. In a tradtional architecture the only choices are to compute the important locations, provide no response (so the server must wait for a timeout) or return an error. *PAM* instead allow apps to return the raw data, a full result, or an explicit reply stating that no computation can currently be carried out. This decsision is based on a computation policy which can be set automatically by the app based on available resources or explicitlt by the app user.

PAM's computation policies are designed to reflect the varied nature of the *preferred computation capabilities* of nodes in a network. Usually a node will listen on a specific port, at a specific path, if it can process a HTTP request sent to its IP address on that port and path. If a node does not currently want to compute a result for that request it could just not listen for requests however this raises two key issues:

1. It may be that the node can afford to provide the data needed to satisfy a request but cannot spare the computational resources to process it.
2. Simply returning no response mean the requester must wait for a time-out which cannot be distinguished from a network failure.

To solve these issues I introduced the concept of a `ComputationLevel`, this value is returned with the response to a request to indicate what level of computation was performed. I chose to support 3 levels: `NoComputation`, `RawData` and `CanCompute`. The first of these indicates that no computation could be done at this time and should only be returned when no handler could be run for the receievd request, this solves problem 2 as it gives explicit feedback to requesters. The second two levels attempt to solve problem 1 by allowing the receiver to explicitly inform the requester whether or not they could compute a full result or only provide the data so that the requester can then go on to use that as they please. I could have given a more granular selection of options i.e. raw data, partial result, full result but as there has to be a clear understanding between the requester and receiver of what each level of computation means I felt that too many levels of granularity would actually cause confusion.

For many possible usecases the desirable level of computation is likely to be variable, possibly based on network conditions, energy levels and load. Many of the middlewares discussed in §1.2 perform offload based on factors such as these and so to allow mechanisms such as these to be implemented it would be ideal if dynamic changes to the computation policy could be supported. This was an optional extension.

In order to support the policy described above I decided to create a `PolicyAwareHanlder`, this satisfies the Go built in `HandlerFunc` interface which allows it to be chained with other Go middlewares (this is discuessed further in §3.6.1). The function which creates this handler takes a computation policy as its argument, this is essentially a map from HTTP request paths, to a map from `ComputationLevel`s to HTTP handlers.

2.5.2 Request Policies

The concept of offloading computation is dealt with extensively by some of the work discussed in §1.2. It is a key part of optimisng the energy use of lower power and mobile applications. However, the resources available on mobile and edge devices should not be underestimated [14] and (especially when connectivity is poor) sometimes local computation is cheaper than transferring data over the network. Request policies are designed to support offloading in the form of a `preferredProcessingLocation` parameter which can be set to `Local` or `Remote`. This extends further, to the case where we need to aggregate data from a number of sources (a common requirement of IoT sensor networks), where we can use this parameter to express whether or not we would like to receive raw data or a parial aggregate (remote computation).

It is feasible that many nodes in a network may be listening for HTTP requests on the same path. This causes confusion between requesting raw data from another node and so setting `preferredProcessingLocation` to `Local` but not wanting to handle the entire request locally as we don't want to use our own data. For this reason an extra field, `hasAllRequiredData`, is added to differentiate between these two cases. An example of where this is used **NOTE: should I add figure to illustrate?** is in a peer-to-peer network of sensor nodes where they need to exchange readings and so request data from each other using the "/get-sensor-readings" path.

The phrasing "preferred" is used above because the capabilities of a node are considered of higher importance than the preference of the requester. This is in contrast to a system such as MAUI [9] where we can offload computation based entirely on a local decision rather than allowing the server to reject the processing request based on load. The algorithm described below is used:

NOTE: should I instead use pseduocode?

1. Attempt to send request.
2. If we have a registered handler for the request path in our local computation policy AND the preferred processing location is local AND we have all of the required data then execute the local handler.
3. Else send a HTTP request, with the request policy attached, to the target node.
4. The target node checks if it has any registered handlers for the specified path.
    - If it has a `CanCompute` handler and a `RawData` handler then use the former if `preferredProcessingLocation` = `Remote` and the latter otherwise.
    - Else, if it has a `RawData` handler then use that.
    - Else, return `NoComputation`

Identifying the requester is an essential part of providing the user with the ability to protect their data on a granular basis. Whilst cryptographically proven identities are out

of the scope of this project, the data policies described below rely on the honesty of the requester and use the `requesterID` field of the request policies.

Request policies are primarily supported by a wrapper around the Go HTTP client, `PolicyAwareClient`, this deals with deciding whether to perform computation locally or to offload it. However in order to support the users preference regarding processing location the `PolicyAwareHandler`'s must also help in supporting these policies.

2.5.3 Data Policies

The popularity of cloud computing and big data analytics has lead to a trend in the upload of large amounts of data to central cloud servers. Even if unique identifiers are removed from this data, it is still possible to infer the identity of specific individuals from it in some cases [25]. It also requires us to trust the cloud provider to keep our data secure.

Some users may prefer not to rely on the cloud provider to protect their privacy. Several privacy mechanisms have been divised to protect user's privacy against statistical attacks, a few are outlined below.

k-anonymity [24] requires us to have k records which are identical in terms of values which could be used to reconstruct a subjects identity. This is usually achieved through the generalisation and suppression. **Should I include an appendix for generalisation and suppression?**

L-diversity [20] goes further and seeks to avoid attacks based on background knowledge of the statistical distributions of datasets or individuals data. This is again achieved through generalisation and suppression of records.

ϵ-differential privacy [12] defines privacy based on how much more likely we are to be able to infer information about a subject if information about them is in the database as opposed to if it is removed. Several methods exist to ensure this, some of which add noise to the results of queries and others which add noise to the data itself.

PAM aims to be able to support all of these privacy mechanisms as well as more intuitive mechanisms such as removing specifc rows or columns as the user requires. It also aims to be flexible so that, as more privacy mechanisms are developed, they are also likely to be able to be implemented as *PAM* data policies.

Part of the middleware is a database wrapper, `PrivateDatabase`. I chose to focus on SQL databases because of how ubiquitously it is used, however the principals could be applied to other databases. I decided to support four common queries which are likely to be used by HTTP handlers: `SELECT`, `UPDATE`, `INSERT`, and `DELETE` but left table creation and deletion[24] out of scope for this project as it is not something which is commonly required by remote nodes and does not lend itself well to data policies which are specified in advance, per table.

[24]Specifically `CREATE` adn `DROP` are not supported.

2.6 Starting Point

The previous work on offloading computation and privacy mechanisms discussed in §1.2 lead to the decision to provide flexible policies which can be used to express these mechanisms rather than building in native support. Mosco [26] supports a range of specific transforms to data whereas I have chosen to provide more generic transforms which gives the advantage of being able to express more transforms which a user may desire at the expense of them possibly being more complex.

My choice of the image processing use case is borrowed from the CloudAware [21] paper, Mosco uses location sharing, mobile health monitoring and participatory sensing data as examples where privacy preserving mechanisms can be applied but I instead opted for household power consumption monitoring due to the availability of the dataset.

SQL was chosen as the query language to support due to its ubiquitous use in industy. I chose to create a wrapper for the MySQL database rather than implementing my own database as this would have been a large project in its own right and, whilst possibly more efficient, would not have illustrated the effectiveness of the data policies any better than a wrapper around a widely used, existing database.

2.7 Example Usecases

In order to achieve my goal of discovering where a middleware such as the one I built is applicable I decided to build some example use cases. These helped me test the functionality more holistically than unit tests could, provided end-to-end systems to benchmark the middlewares performance and also allowed me to see in what situation it was most effective. To this end I chose three examples which isolated different parts of the middleware.

Figure 2.4 and Figure 2.5 illustrate the systems used for the examples. The numbers denote the order of requests sent over the network with dotted circles used to mark requests that may occur depending on the policies selected. What happens at each stage in the different examples is described in the following subsections.

2.7.1 Power Consumption Monitoring

Big data analytics and crowd sourcing has become a large part of the business model for many large companies. This has many advantages, such as better pollution management, the possibility of smart cities and a greater awareness of how much energy we use relative to other people [23, 11].

I selected energy consumption monitoring as a key example because we can imagine collecting this data from IoT devices and also potentially storing data from multiple

devices at another intermediate edge location. The free availability of a large dataset was also a deciding factor.

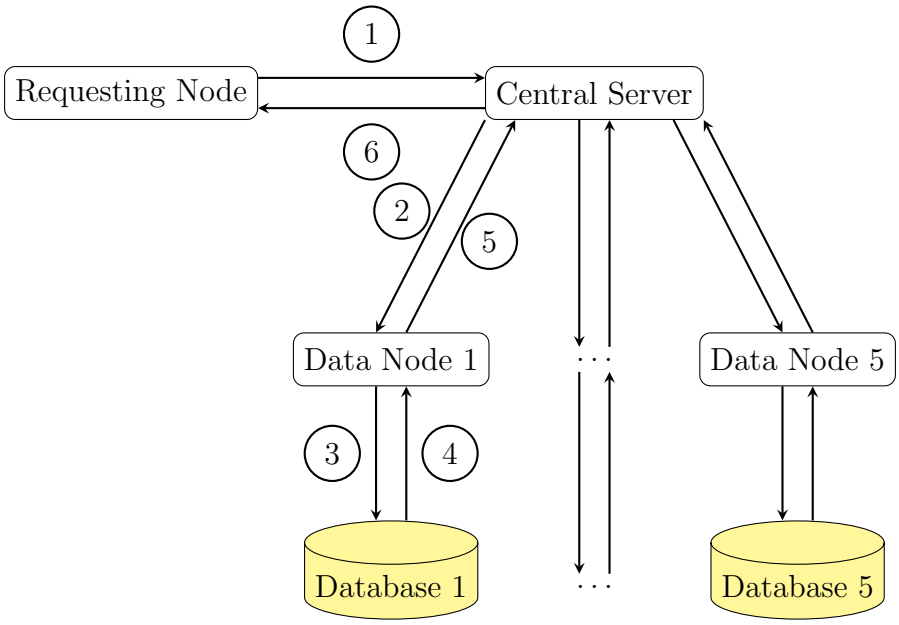


Figure 2.4: Diagram showing the energy consumption example use case.

Figure 2.4 displays the example in terms of "Nodes" and a "Central Server". The Server is imagined to be a central cloud server and the nodes could represent IoT devices or other edge devices which hold data from multiple IoT devices.

Each database holds a full copy of the dataset, I could have split the dataset however as the data has a date and time attached to it I felt that it was best to have data covering the full time period on all of the nodes. The precise data contained in the databases is not of importance as it merely served to illustrate the effectivness of the policies.

Combinations of the policies were used to control the action of the architecture in the diagram to form two separate examples as explained below.

Applying Data Policies

This example seeks to illustrate how data policies can be used to restrict requester's access to data. In this case the numbers in the diagram correspond to the following actions:

1. A request is sent to the server for the average energy consumption across all of the Data Nodes for a specific time period (i.e. all other houses in a user's local area for the last week). This has a `preferredProcessingLocation` of `Remote`.
2. The Central Server can compute the request from 1, it does so by sending a concurrent request to each Data Node requesting their local average energy consumption for the time period with a `preferredProcessingLocation` of `Remote`.
3. The Data Nodes can compute the result for the request and so access their databases, passing the request policy from the Central Server to their `PrivateDatabase`.

4. The `PrivateDatabase` applies the Data Policy specified for the Central Server, in this case it requires dates to be reported at the month granularity and so all times are set to '00:00:00' and all days are set to the 1st of the month as part of a transform. The query is then executed on a transformed version of the dataset and result is returned to the Data Node.
5. A local mean is computed by the Data Node and it is returned to the server with computation level `CanCompute`.
6. The Central Server aggregates the means from each Data Node and takes the mean of these. It returns this to the Requesting node with computation level `CanCompute`.

By comparing this to the same example without using the middleware (no data policies are applied) we can illustrate the effect of applying them. The complexities of policies used here could be extended to provide stronger guarantees as discussed at the start of this section but as those are the subject of other work it was considered out of scope to implement them here. A discussion of how they could be implemented can be found in §4.2.2.

In the remainder of this document this example is referred to as *EnConsData*.

Retrieving Raw Data or Partial Aggregates

The next example displays how we can control the location of processing even when the data needed for processing in not containedd within the request we receive. The numbers on the diagram corresond to the following actions:

1. A request is sent to the server for the average energy consumption across all of the Data Nodes for a specific time period (i.e. all other houses in a user's local area). This has a `preferredProcessingLocation` of `Remote`.
2. The Central Server can compute the request from 1, it does so by sending a concurrent request to each Data Node requesting their local average energy consumption for the time period with a `preferredProcessingLocation` of `Remote` or `Local` depending on a parameter of the request to the Central Server.
3. The Data Nodes can compute the result for the request or return raw data. If the `preferredProcessingLocation` is `Local` they run a handler to return raw data otherwise they run one to return a local mean. In either case they access their databases, passing the request policy from the Central Server to their `PrivateDatabase`.
4. The `PrivateDatabase` applies the Data Policy specified for the Central Server, in this case there are no transforms to apply. The query is then executed and result is returned to the Data Node.
5. If the full result handler is run a local mean is computed by the Data Node and it is returned to the server with a `ComputationLevel` of `CanCompute`. If it is the raw data

handler the data from the database is instead returned with a `ComputationLevel` of `RawData`.

6. The Central Server aggregates the results from each Data Node, computing a mean if raw data is returned, and takes the mean of these. It returns this to the Requesting node with computation level `CanCompute`.

Here we see the ability to control the processing location even when we do not have the data required as part of the request (or stored locally). The example can be extended to include data nodes which have computation polcicies specifying only one handler, either for `RawData` or `CanCompute`. In this case the available handler is run even if it does not match the `preferredProcessingLocation` and the server handles either case based on the `ComputationLevel` attached to the result.

In the remainder of this document this example is referred to as *EnConsComp*

2.7.2 Image Processing

Offloading computation to a server has been shown to reduce energy requirements and sometimes provide a latency gain as discussed in §1.2. However the requirements of the server have not generally been considered, neither have the users specific preference regarding processing location.

This example of performing image processing either on an Edge Node or a Central Server shows how the combination of computation and request policies can be used to provide support for the kind of offload policies described previously to be implemented on top of *PAM* but also remains flexible to user and server defined policies. Image processing was also used as an example use case in the CloudAware [21] paper to evaluate the effectiveness of their offloading strategy and so it felt like a natural choice to evaluate that capacity within my middleware.

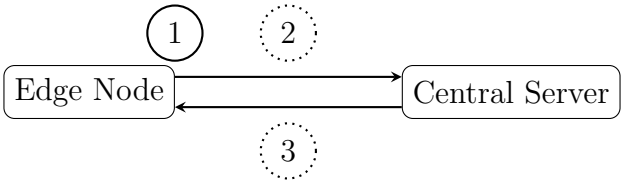


Figure 2.5: Diagram showing the image processing use case.

In Figure 2.5 the numbers refer to:

1. A request to perform object recognition is set with a `perferredProcessingLocation` of `Local` or `Remote`. If the computation policy of the Edge Node has an active handler registered for object recognition, and the preferred location is `Local` then this is executed and we stop.

2. Otherwise the request is sent to the Central Server with the preferred location and the image.

3. The server has a registered handler to compute a full result, it runs the object detection on the image and returns object tags with probabilities along with the `ComputationLevel` of `CanCompute`.

This means that the edge node must both be have an active handler and a preference to compute locally in order for it to occur. We can imagine setting the capability to compute based on i.e. current battery level or CPU load. The preference may be decided based on an offloading policy or by the user. This example could be extended to support the server returning `NoComputation` if it was too busy and so falling back to local computation even if it was not preferred.

A dynamic version of the computation policy supporting frequent changes to which handlers are active would be useful for this example as the criterion for it being available may be transient i.e. power.

In the remainder of this document this example is referred to as *ImPro*.

Chapter 3

Implementation

PAM itself contains 3857 lines of Go code with the example usecases consuming another 1635 lines. The most demanding piece of work was building the privact database (§3.7) to support the data policies, although building the policy aware handler (§3.6) in a modular and extensible way also required considerable thought.

The remainder of this section will give a general overview of the repository and then discuss the key components of the middleware in turn:

1. **Computation Policies** (§3.2): describes the format of computation policies as well as the methods defined upon them to support both dynamic and static computation capabilities.
2. **Request Policies** (§3.3): gives the format of request policies.
3. **Data Policies** (§3.4): outlines how transforms for database tables can be specified in request policies.
4. **Policy Aware Client** (§3.5): details the wrapper around the normal Go HTTP client which supports offloading of requests.
5. **Policy Aware Handler** (§3.6): introduces the middleware layer which resolves computation and request policies for incoming requests.
6. **Private Database** (§3.7): introduces the interface for private database wrappers to be used by handlers built on the middleware, as well as detailing an implementation of this interface which wraps a MySQL database.

3.1 Repository Overview

Below is the directory structure for my code repository with explanations for their contents where necessary. Subfolders are shown underlined.

- PartIIProjectImplementation
  - image_recognition
    - Methods for object labelling in images using tensorflow adapted from a tutorial.[1]
  - middleware
    - All middleware components with tests inluding: request, computation and data policies; the policy aware handler and client; and the private databse wrapper.
    - bench_to_csv
      - Code to export benchmark results with confidence intervals to CSV files.
  - usecases
    - Each subfolder contains an example use case, scripts to update dependencies and files for deploying the examples in Docker.
    - aggregate_power_consumption_comp_policies
      - Code for *EnConsComp*.
      - data-client-configurable
        - Command line arguments dictate the client's computation policy.
      - mysql-init
        - Code for a node with a copy of a local MySQL databse, adapted from a tutorial.
      - requesting-client
      - server
    - aggregate_power_consumption_data_policies
      - Code for *EnConsData*.
      - data-client
        - Code for data clients which generalise dates of readings.
      - mysql-init
        - Identical to in the folder above.
      - requesting-client
      - server
    - image_processing
      - Code for *ImPro*.
      - client
        - Code for the client which uses a dynamic computation policy.
      - server
    - no_mware_aggregate_power_consumption
      - The energy consumption examples without *PAM*, the subfolder structure is identical.
    - no_mware_image_processing
      - Code for the image processing example without *PAM*, the subfolder structure is identical.
    - simple_usecases
      - Code for a simple test case for the middleware.

[1] https://github.com/plutov/packagemain/tree/master/04-tensorflow-image-recognition

3.2 Computation Policies

In the rest of this section I describe a static and dynamic computatioin policy implementation.

Computation Policies are used to specify HTTP handlers for specific request paths. For each path it needed to be possible to support a ahndler which would return the raw data required to compute the result to the request and also a handler which would compute the full result locally.

Two different implementations of these polcies are included in the middleware. One designed to be set statically (and possibly updated infrequenly offline) and another which is designed for dynamic updates. In order to support both types of policy (and potentially future variants) an interface was designed specifying the minimum required method a policy needs to support. These methods are:

1. **Register**: takes a path, `ComputationLevel` and a handler and stores a mapping from the path and level to the handler.
2. **UnregisterAll**: removes all mappings from a specific path which are registered.
3. **UnregisterOne**: removes a single mapping which was previously registered.
4. **Resolve**: takes a path and preferred processing location and returns the appropriate handler and the computation level for which it was registered.

3.2.1 Static Computation Policies

The static implementation consist of maps from strings representing request paths, to a map from `ComputationLevel`s to HTTP handlers. As we do not intend to update the policy online it can be assumed that the data structure is read only after setup.

Only the methods detailed in the interface are implemented on the policy, the first three simply add or remove entries in the underlying maps. The resolve method contains an important algorithm for combining request and computation policies.

Request and Computation Policy Resolution Algorithm

The goal of this algorithm is to return a handler for the request path it is given based on the request policy attached to the request and the computation policy specified at the received. Its priorities are as follows:

1. Return a handler if one is regsitered.
2. Return a handler which will compute a result at the `preferredProcessingLocation` specified in the request policy.

As such it only return `NoComputation` if the node is incapable of fulfilling the request at all.

NOTE: Should I pseduocode this algorithm? It is essentially a couple of nested ifs

3.2.2 Dynamic Computation Policies

The capacity for nodes to compute results which are requested of them may vary temporally based on factors such as energy levels, load or user preference. For example when a phone is in low battery mode it may disable some of its computational resources and hence not be able to execute some processing with adquate performance.

In order to support these usecases I added a second computation policy implementation to my middleware. This extension supports concurrent update and access of policies as well as adding the concept of an active handler.

The interface gives methods for registering and unregistering handlers however if the availability of a handler is constantly changing it may be more convenient to simply activate and deactivate the handler for a specific path and computation level. This is reflected in two additional public methods:

- **Deactivate**: takes a request path and `ComputationLevel` and marks it as deactivated, this means it will appear as though it is has been unregistered.
- **Activate**: takes a request path and `ComputationLevel` and marks it as active, this will make a handler which was previously deactivated appear as though it was registered again.

In order to support this functionality a boolean was associated with each handler to indicate whether or not it was active. Concurrent update and access was suported by adding a mutex associated with each handler, methods to get, activate and deactivate handlers from `dynamicComputationCapabilities`[2] take a lock on these mutexes before performing updates to the active variable or accessing the handler.

By altering the data structure in this way, only minimal changes to the **Resolve** algorithm had to be made.

3.3 Request Policies

Request policies are designed to communicate the identity of the requester, whether they would prefer processing to occur locally (meaning on the requesting node) or remotely (on the receiving node), and whether or not the request contains all of the required data

[2] A `dynamicComputationCapabilities` is a map from `ComputationLevel`s to structs containing a handler, a mutex and an "active" variable.

necessary to perform the request. In my implentation of the middleware this is represented by a struct with three fields:

- `hasAllRequiredData`: a boolean which specifies whether all of the data needed to compute a result is in the body of the query. This may be useful when we have the capacity to compute a full result to a query but only if the data is provided.
- `preferredProcessingLocation`: can be `Local` or `Remote`, this specifies whether we would prefer raw data to be returned so that we can compute locally (or for the whole request to be handled locally if we have this capacity and all of the required data is held within the request body), or insetad if we would prefer a result to be computed remotely and returned.
- `requesterID`: identifies the requester. In PAM these are chosen by the user and not required to be cryptographic as this was considered out of scope for this project. They are assumed to be unique and not forged as the technology exists to ensure this **add reference**.

Every request made in the middleware needs to have a request policy attached. I considered using various mechanisms for sending this data along with the request such as Google's Protocol Buffers[3] or simply formatting the request body in JSON[4] with fields for the policy and request body separately. However, as HTTP GET requests do not traditionally have a body and the amount of data which constitutes a request policy is small, I instead opted for the simpler approach of martialling the request policy fields into HTTP query parameters. This has the downside of reserving 3 of the available parameter keys but as the key space is very large[5], this seemed preferable to adding a parsing library for a structured data format as a dependency.

3.4 Data Policies

Data polcies seek to allow users to specify policies which restrict requester's access to data. I decided to support two types of table operation:

1. **Column Transforms:** these are specified to operate on a specifc column of a specifc table. They are applied to each value in the column before the data is returned. Each transform has access to the value in that column for the row it is acting on and can either return another value of the same type or indicate that the row should be excluded alltogether. If any transform acting on a table indicates that a row should be excluded then it should be. An example use case would be to generalise location data to only be given to within a kilometer rather than as precise as possible.
2. **Excluded Columns:** a list of columns can be specified, for each table, which must

[3]https://developers.google.com/protocol-buffers/
[4]https://www.json.org/
[5]Most HTTP parsers have a cap on the header length which restricts the number and length of keys.

be excluded from use i.e. we may remove the names of patients from a table of medical data.

These data policies need to be able to be different for different requesters. In order to keep them as brief as possible they can be specified per `PrivacyGroup`, which is simply a list of requester identities.

Several data structures are used to construct these policies:

1. `ColumnTransform`: takes a value, performs a transform and returns a value along with a boolean specifying whether or not to ignore the row containing the passed value.
2. `TableTransform`: Maps from column names to `ColumnTransform`s.
3. `TableOperations`: Contains a map from table names to `TableTransform`s and a map from table names to a list of columns to exclude.
4. `DataTransforms`: Contains a map from `PrivacyGroup`s to `TableOperations`

A privacy policy then consists of a list of `PrivacyGroup`s, a `DataTransforms` object and a timestamp for when it was created (the use for this is discussed in §3.7).

An interface was specified for data policies in a similar way to for computaton policies. They must support a method to `Resolve` requester IDs with the stored data polciies and return a set of `TableOperations`, additionally a `LastUpdated` method must be supported[6]. Only the static case was implemented which simply returns its creation time as its time of last update.

Moved the below from prep, edit it in

When reading from the database (using `SELECT`) I decided that queries should be run on copies of tables which have had transforms applied. The nature of these transforms are specified as part of the data policies and consist of columns to remove from tables in the database, and functions which must be applied to all values in a column. These are specified for `PrivacyGroup`s which consist of lists of requester identities. The functional transforms must preserve the type of the data in the column and should return values which can still be used to gain useful information from the query. A prime example is generalisation, this is when we make sure the released data is of a coarse enough granularity i.e. only returning dates and times at the month granularity rather than the second granularity. The functions can also indicate that we should exclude a row from the transformed table based on the value in the column.

For writes to the database (using any of the other commands) it should not be possible to write to any column which is supposed to be excluded from view. Allowing writes such as this would expose a flaw in the security of the database as you would get a different response if a write to a column secceeds, to if that column did not exist which may allow you to infer the names (or values in the case of `UPDATE`) of columns which you were are

[6] `LastUpdated` is required for caching of transformed tables as discussed in §3.7.2.

meant to be aware of.

This allows mechanisms removing arbitrary rows and columns, and applying arbitrary transforms to values to be implemented as data policies within the middleware

NOTE - should I mention this?: Initially I thought we could specify, for each handler, which data they *may* access and then have a policy for groups of users (privacy groups) which specified what data they could access and we would only run the handler if a handler only accessed data that the requester had a right to. This is essentially a very coarse form of access control.

NOTE: I could talk about other options I had here, such as to allow transforms to take into account multiple values in a row or column or to allow there to be many-to-1 or 1-to-many functions acting on rows (i.e. to produce summaries of multiple rows). I am not sure whether this is worth doing?

Note: it is clear that some of this came up whilst I was implementing the policies, I don't know whether to mention what I originally intended here and then say what the difficulties were and why I changed it later or just leave it like this?

3.5 Policy Aware Client

A wrapper for the HTTP client, `PolicyAwareClient` from the Go `http` library is provided. Upon creation we pass this a computation policy for it to enforce. Before detailing the functionality it provides the definition for the requests and responses it deals in must be given:

1. A `PamRequest` consists of a request policy and a normal Go HTTP request.
2. A `PamResponse` contains a `ComputationLevel` and a normal Go HTTP response.

The `PolicyAwareClient` implements a `Send` method which takes `PamRequest` and returns a `PamResponse` by either sending the request to the target node (with the request policy attached) or computing a result locally as described in the algorithm in §2.5.2. Aside from implementing potential local computation of a request this was trivial to implement. The request policy fields are marshalled as query parameter on the HTTP request, the wrapped HTTP client then executed the request and the `ComputationLevel` of the result is unmarshalled from the header of the received response (how this is added is described in the next subsection). This results in an error if the receiver of the request is not using *PAM* as there will be no `ComputationLevel` in the response header.

To faciliate local computation we must `Resolve` the path of the request with the local computation policy[7]. A check is then done for whether local computation is preferred,

[7]When resolving a request path locally we must actually pass `Remote` as the preferred processing location to simulate receiving the request from elsewhere with a desire to compute on the receiving node.

and that there is a registered handler and that all of the required data to compute a result is present in the request. If so run the request locally.

Retreiving the result of running a HTTP handler locally, without sending it over the network with a `localhost` destination, proved to be difficult. I achieved it through the use of a testing library[8] which allowed me to record the response. Utilising a testing library in this manner felt slightly inappropriate as they are often not optimised for performance, I consdidered implementing my own response recorder but after analysing the library code decided that it would be almost identical and so leveraging the existing code was better overall.

3.6 Policy Aware Handler

This section details the part of the middleware which supports computation and request policies (aside from in the case of local computation of a request which was discussed in the section above). First I describe the building blocks Go provides for implementing this part of the middleware and then I explain how the required functionality is supported in an intuitive and extensible way.

3.6.1 HTTP handlers in Go

In Go a HTTP handler must satisfy an interface which dictates that it implements the `ServeHTTP` method. This takes a response writer and a request, computes the result to the request and then writes this result to the response.

For ease of use, a concrete type is available, `HandlerFunc`, which simply aliases the function type of a method with the same signature as `ServeHTTP`, it implements the `ServeHTTP` and simply calls itself to satisfy it. This trickery allows any function which has the signature of `ServeHTTP` to be asserted as a `HandlerFunc` in order to satisfy the `Handler` interface.

HTTP handler middlewares are then simply functions which take a `HandlerFunc` and return another one, an example would be a logging middleware which writes the header of each request to the logs before calling the passed function. In this way we can layer functionality and build custom middlewares out of modular blocks.

I wanted my middleware to be able to fit into these middleware chains easily to allow it to be extensible as possible and so satisfying these interfaces became a key design requirement.

[8] https://golang.org/pkg/net/http/httptest/

3.6.2 Handler Implementation

Computation and request policies are supported mainly through a function which takes a computation policy and returns a `HandlerFunc`. This function is called `PolicyAwareHandler` and performs the following actions:

1. Logs that a request has been received.
2. Unmarshals the request policy.
3. Resolves the computation policy with the request policy as described in §3.2.1
4. Based on the `ComputationLevel` returned by the policy resolution algorithm it sets the computation level field in the response header and then calls the appropriate handler (unless none was regsitered).
5. Logs that the request has been served.

The handlers which are specified in the computation policy may consist of many chained middlewares, this allows us to chain middlewares "below" *PAM* (they will run after it). As the returned value is a `HandlerFunc` it can be used directly as a `Handler` or chained with other middlewares "above" it. This makes *PAM* just another modular component which can be used in the building of Go middlewares for specific use cases.

3.7 Private Database

A key part of meeting the goals for the middleware was being able to provide users with granular control over not only where their data is processed, but what data is processed. Having decided to implement this as a wrapper which sits between the writer of a handler and an SQL database I decided to write an interface which supports almost all of the methods which the `DB` type from the standard Go `sql` library supports. The only difference in the supported signatures is that all of the methods which execute an SQL query also take a request policy as an argument which is needed so that we can apply the data policies correctly.

The methods which are not supported concern database transactions (`Begin` and `BeginTx`) and connections (`Conn`), these were left as they were not required to illustrate the effectiveness of middleware in supporting data policies and would have simply required a wrapper for transactions and connections with almost identical code to that used in the main wrapper. This was considered out of scope for this project as it would have added significant work to implement and test these wrappers without demonstrating the concepts which the middleware supports any better.

I also provide an implementation of the interface which wraps a MySQL database. When instantiated, the wrapper must be given a data policy, it must also be provided with a boolean to indicate whether tables should be cached (this is the subject of §3.7.2).

Many of the methods on this implementation simply pass on the call to the underlying

database which we wrap[9]. The `Connect` method takes a username, password, database name, uri and port and connects to the database accessible at this uri and port. It receives a normal `DB` object which it stores a reference to, this method must be run before any other otherwise they will return errors.

The methods which are of interest are: `QueryContext`, `QueryRowContext` and `ExecContext` (the versions of these methods without "Context" simply pass a call on to the "Context" version with the "Background" context). Each of these first transforms the query and any necessary tables, executes the query on the wrapped database instance and then (if caching is not enabled) drops all transformed tables which were created. The remainder of this section discusses how data policies are supported through transformed tables and then how caching was implemented to potentially mitigate the overhead of creating them.

3.7.1 Supporting Data Policies Through Transformed Tables

Figure 3.1 dispalys the algorithm used to apply data policies to queries. I explain the steps involved below.

The first step towards supporting the data policies using the design specified in §2.5.3 is to establish what kind of query is being run. For this I used a third party library, `sqlparser`[10], which can parse the query and then allows the parsed tree to be "walked", applying a function recursively to nodes in the tree. I implemented a walk which found all statements within the query and creates a list of them. The next step is to determine if the query reads or writes, because reads require transformed versions of tables and writes do not. We do not allow queries which combine reads and writes, they must be executed as two separate queries (splitting the execution of complex queries into read and write phases automatically was considered out of scope for this project). If the query both reads and writes an error is returned stating that the query should be split into mutiple queries.

Another walk then finds all nodes of type `TableName` in the query and builds a list of these table names. The algorithm then forks based on whether the query reads or writes, I will discuss these in turn.

Queries that Read

The only supported query that reads is `SELECT`. Before a `SELECT` query is executed we must achieve two things:

1. Ensure that transformed versions of any tables accessed by the query exist and have had the transforms specified by the data policy (for this requester) applied to to

[9]The methods which simply pass on the call are: `Close`, `Stats`, `SetConnMaxLifetime`, `SetMaxOpenConns`, `SetMaxIdelConns`, `Ping` and `PingContext`.

[10]http://github.com/xwb1989/sqlparser

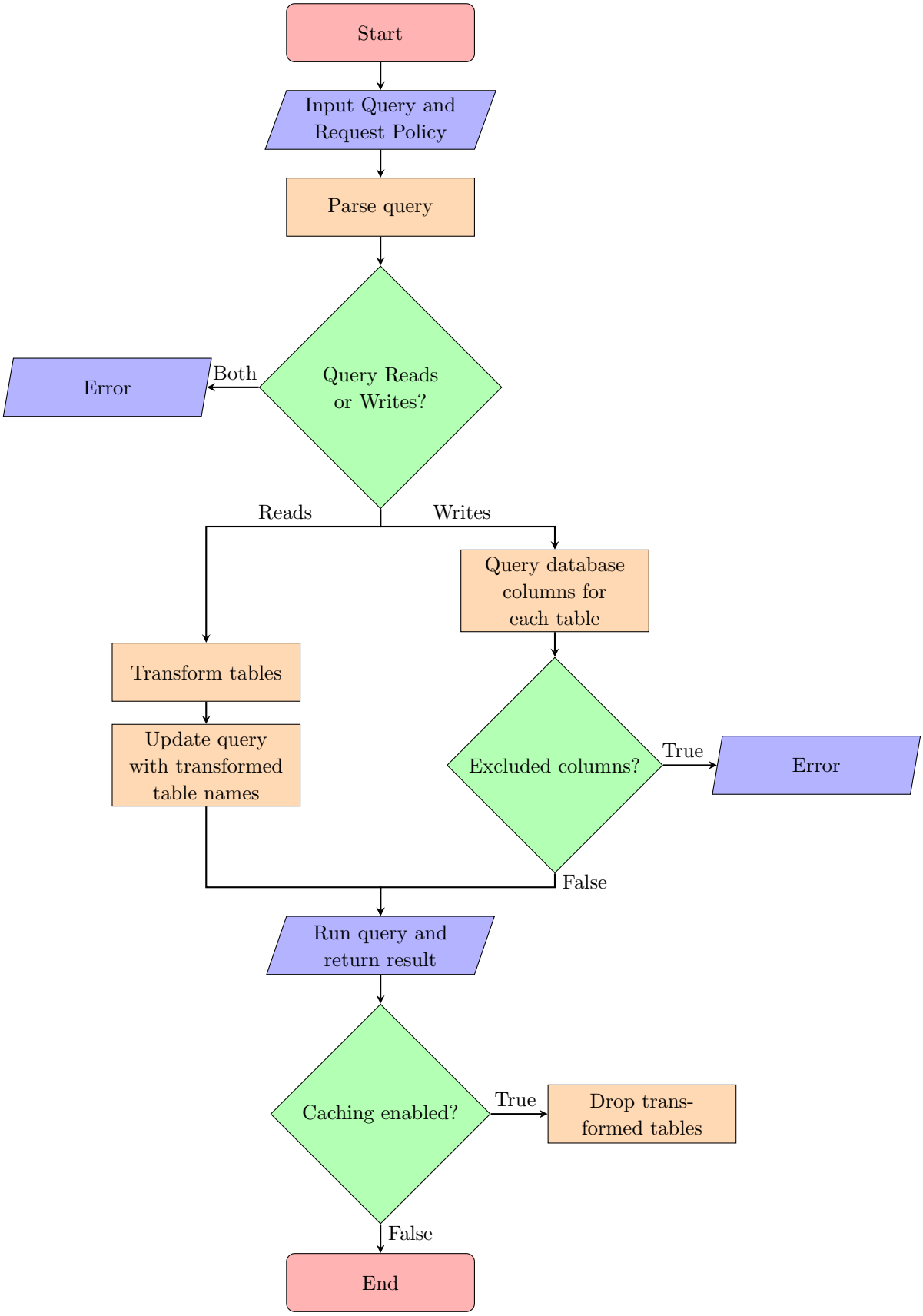


Figure 3.1: A flowchart illustrating how tables are transformed to support data policies.

them.

2. Replace any table names mentioned in the query with the name of the transformed version of the table.

A transformed version of each table names is generated using a prefix based on the requester ID and a random number (unless caching is enabled as discussed in §3.7.2). We then look up the requester ID in the data policy to find the assoicated `TableOperations` and then for each table we apply the relevant transfroms and exclude the relevant columns, this proceeds as follows:

1. A list of the columns of the table to transform are queried and any excluded columns are removed from the list.
2. We drop any table with the transformed table name[11].
3. A new table is created with the same definition as the original table but with the transformed table name using the `CREATE TABLE <new-table> LIKE <existing-table>` syntax.
4. We query the data for the non-excluded columns from the original table.
5. For each row:
   5.1 Variables of the correct type for each column are generated.
   5.2 The values from the original table are scanned into these variables.
   5.3 Any column transforms are applied to the variables.
   5.4 If no transform indicates that the row should be excluded then the (possibly transformed) values are written to the transformed table.

Regular expressions are then used to replace any ocurrences of table names in the original query which have been transformed, with the transformed table names. We then execute the updated query and return the result to the user. Finally, any transformed tables (assuming no caching) are dropped so as not to use up memory and storage space.

The hardest part of this piece of the implementation was generating variables of the correct type to scan the SQL data into. The Go `sql` library does this internally but it is not possible to extract and so some of the logic had to be recreated as a large case statement. The key things to consider were not only the SQL types of the column but also whether or not the data is nulllable or had other parameters such as precision or scale.

In order to limit the memory requirements of applying a transform, rows are read from the database in batches of no more than 1000 rows. **NOTE: Maybe I should make this configurable?**

[11] There should never be an existing table at this point unless the same random number is picked for concurrent queries, this will result in an error.

Queries that Write

Supporting `DELETE` and `INSERT` queries was relatively simple. The database is queried for a list of columns in each table which a query accesses, if any of these tables have excluded columns listed for the user then an error is thrown as explained in §2.5.3. Otherwise we simply execute the query on the raw tables and return the result to the user.

Ideally for `INSERT` queries it would have been possible to allow queries which update non-excluded columns even if other columns in a table are exluded. Unfortunately the parser that I used did not consistently provide the information for the table which a column name was from, especially if the table as aliased. I decided that adding the limitation of not being able to update tables with any excluded columns is not too restrictive as usually if a requester is allowed to update the table, they will also allow them to see all of the database columns as they are likely to be trusted. No other query parsing libraries were readily available for Go and reimplementing the SQL parser was decided to be out of scope for this project.

3.7.2 Caching Transformed Tables

One of the extensions which I decided to implement after initial tests was the caching of transformed tables. The aim was to reduce the overhead of applying database transforms.

Caching was implemented as a configurable for the `MySQLPrivateDatabase` instance, when it is turned on we do not add random numbers to the transformed table name, and instead query when the origial table was last updated as well as calling `LastUpdated` on the data policy. If a table of the transformed table name exists and was created after these times then we just execute the query on that, otherwise we create the table as in the uncached case.

This carries a storage overhead although potentially reduces the memory requirements compared to the uncached case as we may be able to execute concurrent requests from the same requester ID concurrently on the same table rather than having to generate a unique transformed table each time. In order to limit the storage requirements it would be prudent to implement a cache replacement policy such as Least Recently Used or Least Frequently Used **NOTE: do I need to add refs, these are standard algorithms which have been part of previous courses**. This was not implemented as part of this project and is left as future work.

3.8 Example Usecases

A large part of implementing this projects was implementing 3 key examples to demonstrate its effectiveness. I sought to reuse existing data and code wherever possible in creating these examples as the actual use cases are not part of the middleware, however,

significant effort was still put in to building them within the middleware and deploying them in order to test and evaluate *PAM*.

The remainder of this section first covers the deployment strategy for the examples and then details the specific implementation of each.

3.8.1 Deployment Strategy

I decided to run all of the examples locally but needed a method of simulating a network. Docker provided the means to deploy and isolate different applications as containers. I then used Pumba to simulate network conditions in order to assess the examples under relatively realistic, but controlled and reproducible, conditions. The delay induced by Pubma was normally distributes with a mean of 43ms and a jitter of 3ms. This was calculated by collecting latencies for ping commands to the EU-WEST Amazon Web Services region using CloudPing[12]. **Should I include the measurements as an appendix?**

Each example consists of a folder with subdirectories for each type of node. Each of these node directories is a `dep` project meaning it contains a copy of all of its dependencies locally. Where possible, command line arguments were used to make nodes configurable (i.e. to choose their computation policies) in order to reduce the amount of repeated code. Each directory also contains a Dockerfile which builds an image containing the code for the node, this is done following the Docker best practice of using intermediate containers for the build, but having the final, resulting container which we produce an image for only contain the necessary binaries[13]. The Dockerfiles ensure that the Go package dependencies are up to date and install `iproute2` as this is a Pumba dependency.

At the top level of each example directory there are several `docker-compose` files. These can create multiple instances based on images created by the Dockerfiles in the subfolders. For each example we include at least one compose file designed to be run with the command `docker compose up` as this can be done through the GoLand IDE, however this command does not support resource constraints which were needed for my evaluation and so Docker Swarm compatible files run with `docker stack deploy` are also included. These were used to evaluate each example in different configurations.

Also included at this top level are scripts to update the dependency lists for all of the nodes in an example and also scripts to push the images to DockerHub where they are hosted for use by Docker Swarm.

3.8.2 Energy Consumption

In order to implement the two examples described in §2.7.1, I needed to find a dataset of energy consumption data. I chose the Household Power Consumption dataset[14] because

[12] https://www.cloudping.info/

[13] The exception to this is where dynamic linking was needed in the case of tensorflow

[14] https://data.world/databeats/household-power-consumption

it had a large number of measurements and was freely available. In order to use it in the example I first cleaned the data by removing any rows which were incomplete. I then created an SQL database with a table containing all of the measurements. The Dockerfile for database nodes copies this database and starts a MySQL server locally, this Dockerfile was adapted from the answer to a StackOverflow question[15].

The different computation policies which are required for *EnConsComp* were implemented as command line arguments specified in the compose files.

In all of the power consumption examples an extra handler is added to the Data Node which, when queried, triggers the example.

A version of the power consumption example which does not use the middleware and so applies no data policies and performs computation centrally was also built as a baseline approach to compare against.

3.8.3 Image Processing

The image processinge examples uses the go tensorflow library and the code for the actual object recognition performed on images was adapted from a tutorial[16]. In order to allow the preferred processing location to be changed easily an extra HTTP handler was added to the Edge Node specified in §2.7.2 which could be queried in order to trigger the example to run, this has a parameter for the preferred processing location.

A version of the image processing example which does not use the middleware and simply decides on the processing location based on a query parameter to the Edge Node was also implemented as a baseline approach to compare against.

[15]https://stackoverflow.com/questions/35845144/how-can-i-create-a-mysql-db-with-docker-compose
[16]https://github.com/plutov/packagemain/tree/master/04-tensorflow-image-recognition

Chapter 4

Evaluation

The goals of this project were to:

1. Develop an edge orientated middleware with a policy framework which allows graular control over data and the processing of dataset
2. To asses the applicability of such a middleware in various scenarios

To facilitate this evaluation I use several example usecases described above. I also perform benchmarking on some aspects of the middleware to illustrate its performance as reasonable performance is necessary for it to be applicable. Finally I identified privacy mechanisms and computation offload policies from the literature and showed how they can be expressed as *PAM* policies.

My original success criteria were refined in order to make them easier to evaluate Figure 4.1 shows the complete list and how each was met. In order to meet my first three criteria I validated the output and logs of each of my example usecases when running them using a range of valid and invalid requests. Additionally, extensive unit testing gave me confidence that my implementation works to specification with tests covering 100% of files and 82.6% of statements, specifically 78.7% of the statements in the database wrapper were covered with most exclusions being simple error checks which are not expected to occur. **I have more coverage statistics but I feel like this gives enough to make it clear I tested functionality fully**

The remainder of this section first evaluates the effect of PAM on the performance of a system in terms of request latency, storage requirements and memory requirements. It then goes on to evaluate its effectiveness in providing granular control over what data is released and the location of data processing.

Success Criteria	Component	Evidence
PAM supports data policies which allow arbitrary transforms to be applied to data before they are used by queries.	MySQL database wrapper	*EnConsData*, Unit Tests
PAM supports handling HTTP requests locally or remotely based on capabilities specified in computation policies for nodes, and preferred processing locations specified by request policies.	PolicyAwareHanlder, PolicyAwareClient	*ImgPro*, Unit Tests
PAM supports returning no result, a partial result or a full result based on the capabilities specified by computation policies for nodes	PolicyAwareHanlder, PolicyAwareClient	*EnConsComp*, Unit Tests
I can establish the overhead of applying each policy supported by PAM under various conditions.	PolicyAwareHanlder, PolicyAwareClient, MySQL database wrapper	Performane analysis (§4.1)
I can establish which existing privacy mechanisms can be expressed as a data policy.	MySQL database wrapper	Example policies (§4.2.2)
I can indicate potential applications for a framework similar to the one I build.	N/A	Real world use cases (§4.1.3, §4.3)
MAYBE: I can establish which offload policies can be expressed an combinations of request and data policies.	PolicyAwareHanlder, PolicyAwareClient	Example policies

Table 4.1: Success criteria along with the components which meet them and the associated evidence.

4.1 Performance Analysis

In order to make a decision on whether or not to use *PAM* as part of a future project the performance implications of using it need to be considered. Hence, one of my success criteria is to establish the overheads of appplying each kind of policy, this is the subject of the analysis in this section. Additionally the performance impact of using *PAM* on the example use cases is evaluated in order to indicate where potential applications for the middleware may lie.

4.1.1 Overhead of Computation and Request Policies

Applying request and computation policies requires minimal logic and so is expected to represent a fairly insignificant overhead. In the image processing example (*ImPro*) data policies are not used and so the effect is isolated. Figure 4.1 shows the latency of requests with and without the middleware, for remote requests the difference is negligible, locally the error bars overlap but PAM seems to incur a slight overhead. This is likley due to the fact that local handlers are run like normal HTTP handlers in *PAM* and the result that would be written back is extracted, whereas without *PAM* it is simply a function call.

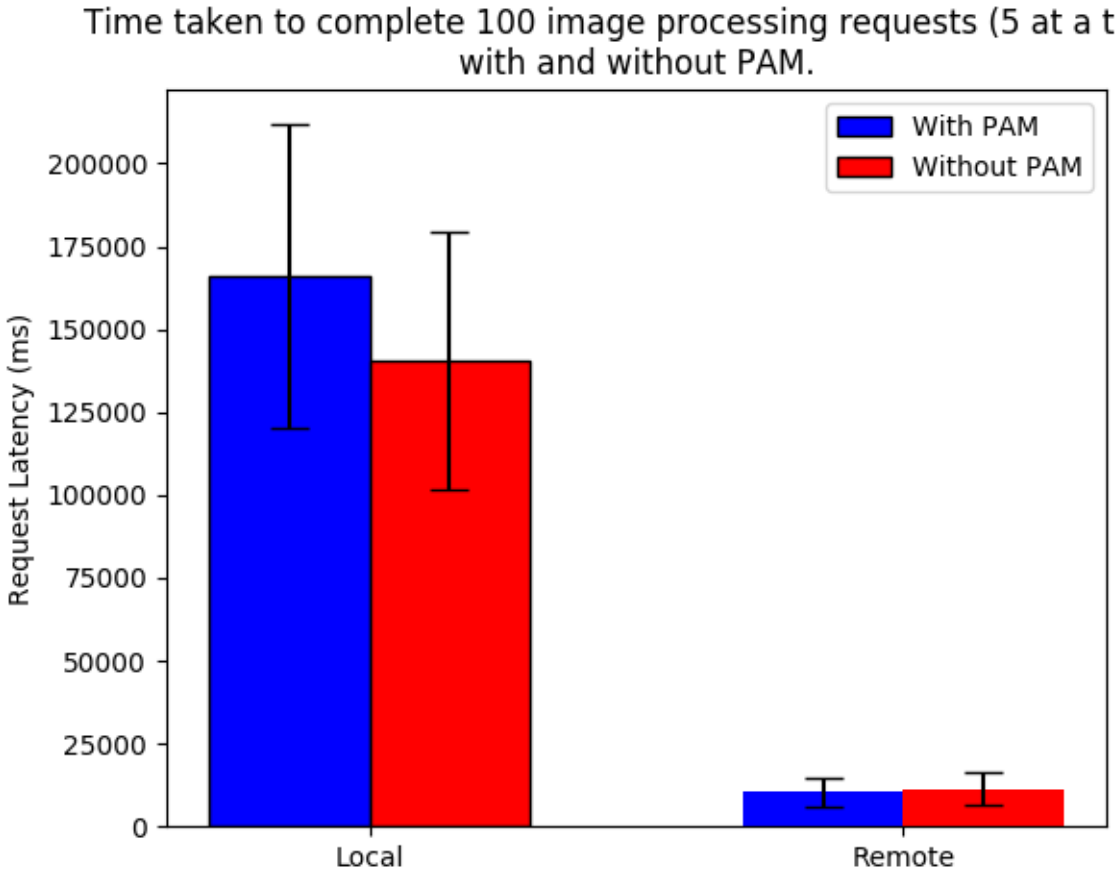


Figure 4.1: Latency of image processing requests with and without PAM.

4.1.2 Overhead of Data Policies

Applying data policies requires a copy of each table which which us accessed to be made with transforms applied to its values. Figure 4.2 illustrates that applying data policies to read queries without caching incurs an overhead which is linear in the size of the table when compared to a normal MySQL query. However, with caching this overhead is removed from all but the first request. This in turn incurs a cost to storage and so caching is a configurable paramater for the database wrapper as some devices may only have very limited storage. Currently no mechanism to evict tables from the cache has been implemented, in future this may reduce the storage overhead and make caching applicable to a wider range of devices.

For write queries (Figure 4.2) we see a fairly small overhead to query duration which is likely caused by having to parse the query and then walk it in order to check for an excluded table names. The memory overhead appears to be linear, analysis of the output of GoLands' memory profiler after running the benchmarks shows that this comes from allocating the nodes of the parse tree for the query and then walking it to build the array of table names.

Storage Requirements: Not sure if this section is worthwhile but I can state

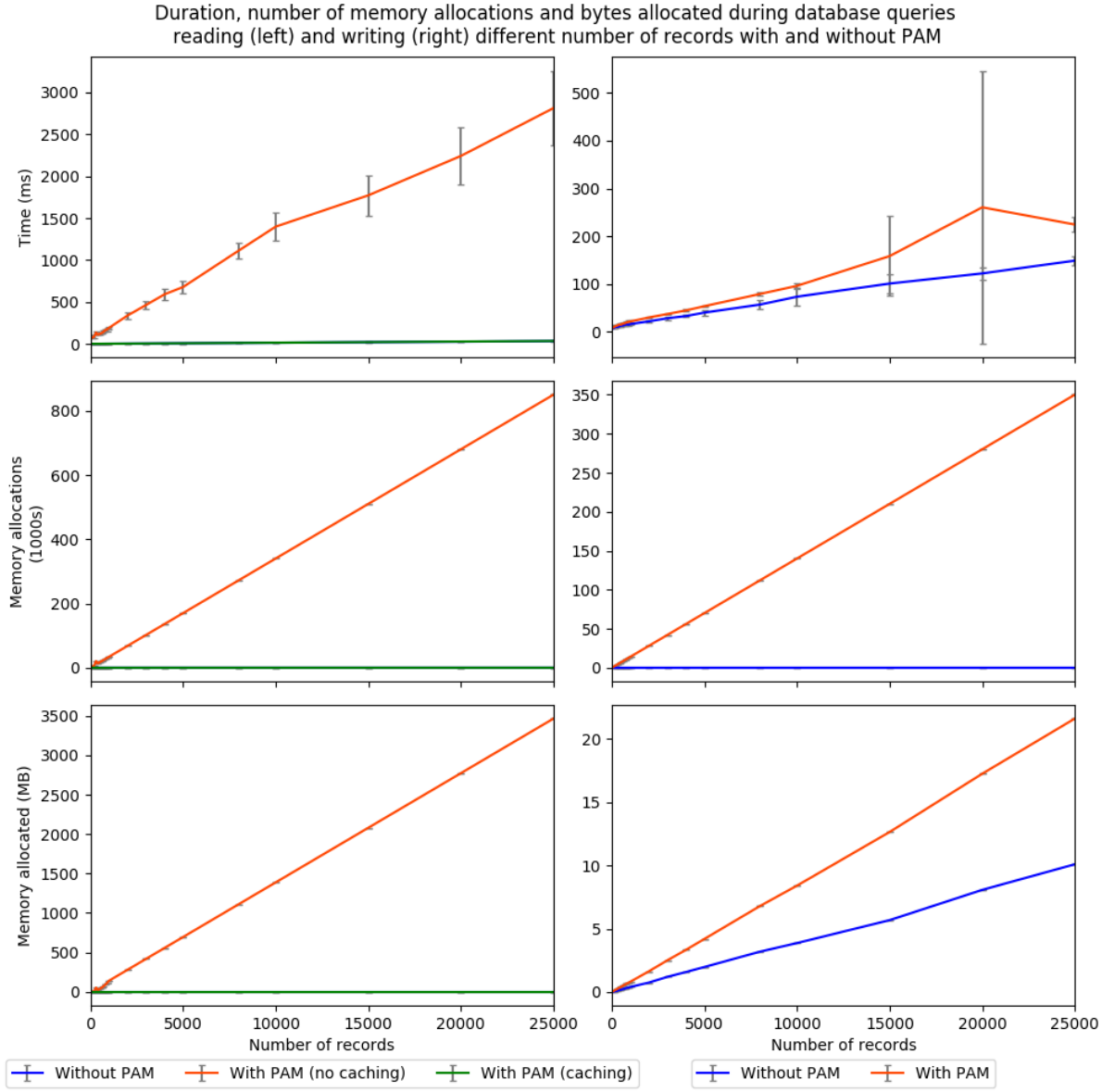


Figure 4.2: Graph showing *PAM*'s overheads on database queries.

that by storing the data on data nodes rather than centrally then we incur the overhead of fetching it but we can trade this off with the latency gain of distributed computation (where it is applicable). Can also mention caching reducing this issue.

4.1.3 Performance Gains in Examples

EnConsComp illustrates the latency and resource saving that distributing processing to the edge can give us. Figure 4.3 shows that for data nodes with 20%, 10%, 5% and 2% of the processing power of the server, pushing the computation to the data nodes results in a latency gain. For data nodes with only 2% of the processing power they started to fail when trying to send data back to the server, this further supports the argument that decentralising computation can reduce load not only on central servers but also the edge nodes. It may be argued that the power of data nodes in some contexts, for example IOT, may be much lower than this however using my full dataset and running locally with resource constraints, 2% was the lowest proportion of the server power I could simulate before the data nodes began to fail under load. However, the trend which arises here is that, invariant of the actual power of the data nodes, an increase in performance is grained by computing the aggregates across many nodes rather than sending all of the data over the network and computing centrally[1]. The same appears true even when the ratio of memory between the central server and the data nodes is varied. Pushing the processing, even to relatively lightweight nodes, yields a latency gain compared to centralising it.

NOTE: maybe try to evaluate how much of that is to do with sending data over the network (I have found a tool which breaks down how long is spent in each stage of the processing) i.e. Figure P shows the reduction in bytes sent over the network when we compute local aggregates. However this doesn't really evaluate my middleware I don't think?

NOTE: Figure 4.4 shows that even when data nodes have much less memory than servers we still get a latency improvement by decentralising whereas 4.5 shows we can get away with less server memory memory if we decentralise computation. I need to decide whether to retake measurements in order to update the first figure or simply cut it out.

Distributing computation can provide a latency improvement compared to centralising it whilst also reducing the requirement for central server memory. Figure 4.5 shows the results of computing an average across all of the energy data using a centralised and decentralised approach for different amounts of server memory. Once the server has "enough" memory, there is little effect on latency by increasing it, however for low amounts of memory there is a spike in latency and then the server begins to fail under load (even with just 5 concurrent requests). Using a centralised approach there is a latency spike with 20MB of server memory and then failure at 10MB, for the decentralised

[1] Aggregates on the server were still computed in parallel.

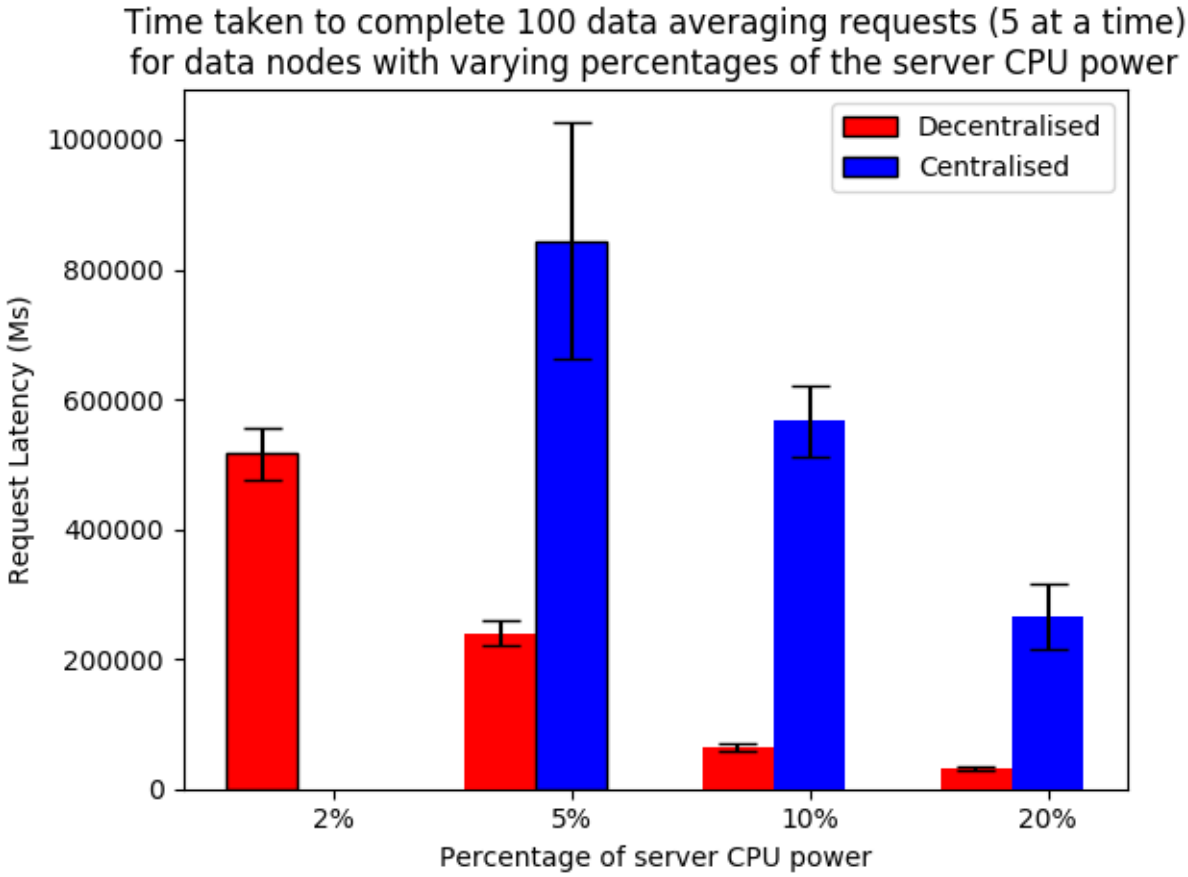


Figure 4.3: Request latencies for 100 requests sent 5 at a time for data nodes with varying percentages of server CPU power.

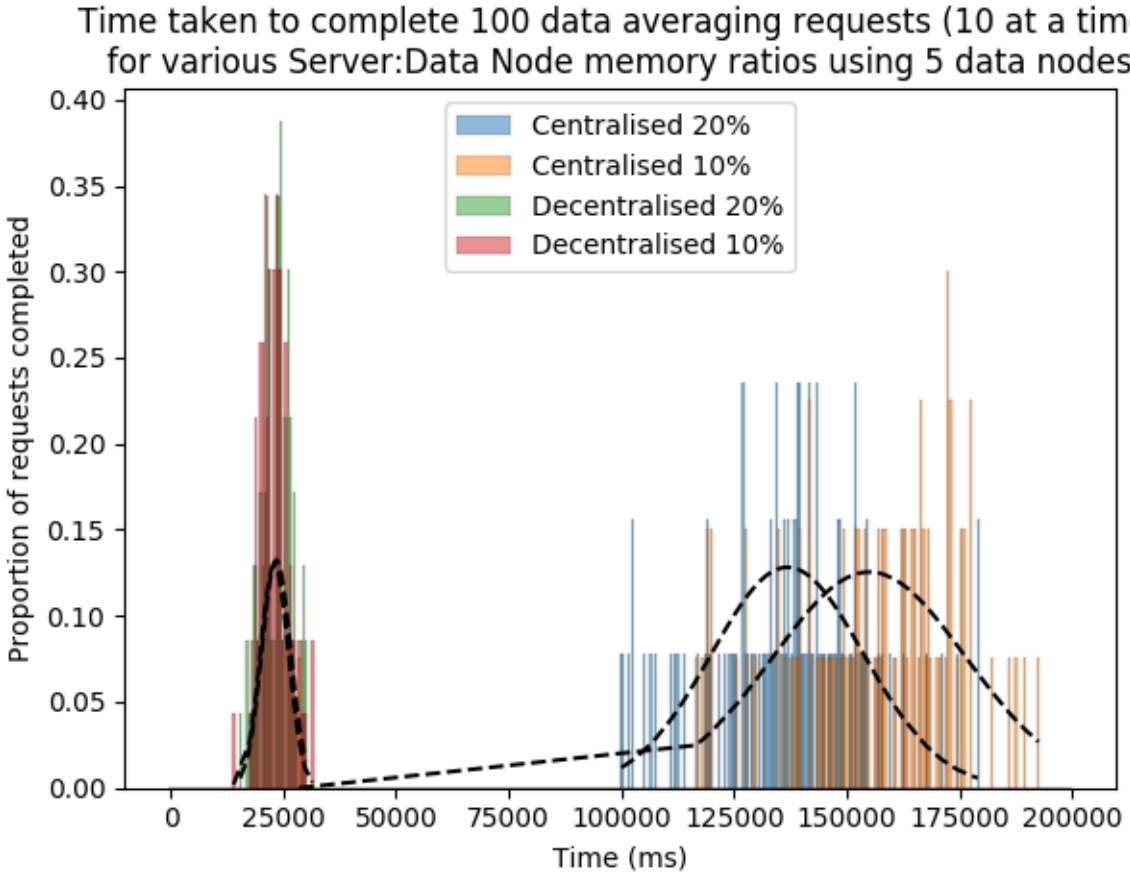


Figure 4.4: Request latencies for 100 requests sent 10 at a time for different Server:Data memory ratios and methods of computation. **TODO: update this figure using bar chart with error bars**

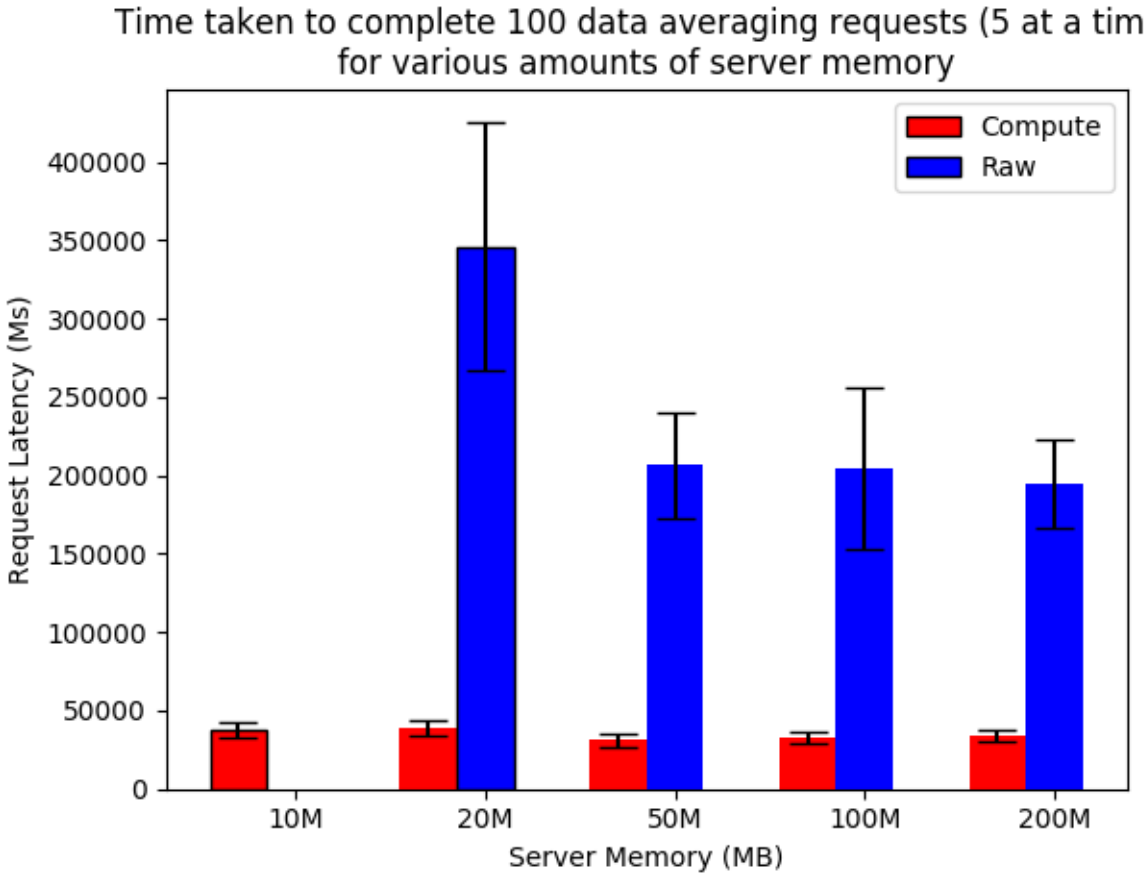


Figure 4.5: Request latencies for 100 requests sent 5 at a time for various amounts of server memory.

there are no spikes but did it still fail when only allocated 4MB of memory[2]. This indicates that by using a decentralised approach it is possible to provision less memory in central clouds. Whilst memory is inexpensive, it can be costly to over-provision in order to cope with high load, using *PAM*'s request policies could help solve this problem by dynamically deciding whether to prefer local or remote computation based on current resource usage. In combination these findings indicate that using *PAM* we can lower the memory requirements of central servers by offloading compuatation to edge devices even when these edge devices have vastly less memory.

4.2 Granular Control Over Data Processing

PAM seeks to provide granular control over not only where data is processed but what data we can process over. This combines the key cocepts of offloading computation and applying complex access control policies, the remainder of this section evaluates how well *PAM* meets each of these goals.

[2]4MB is the minimum amount of memory which can be specified with Docker Swarm.

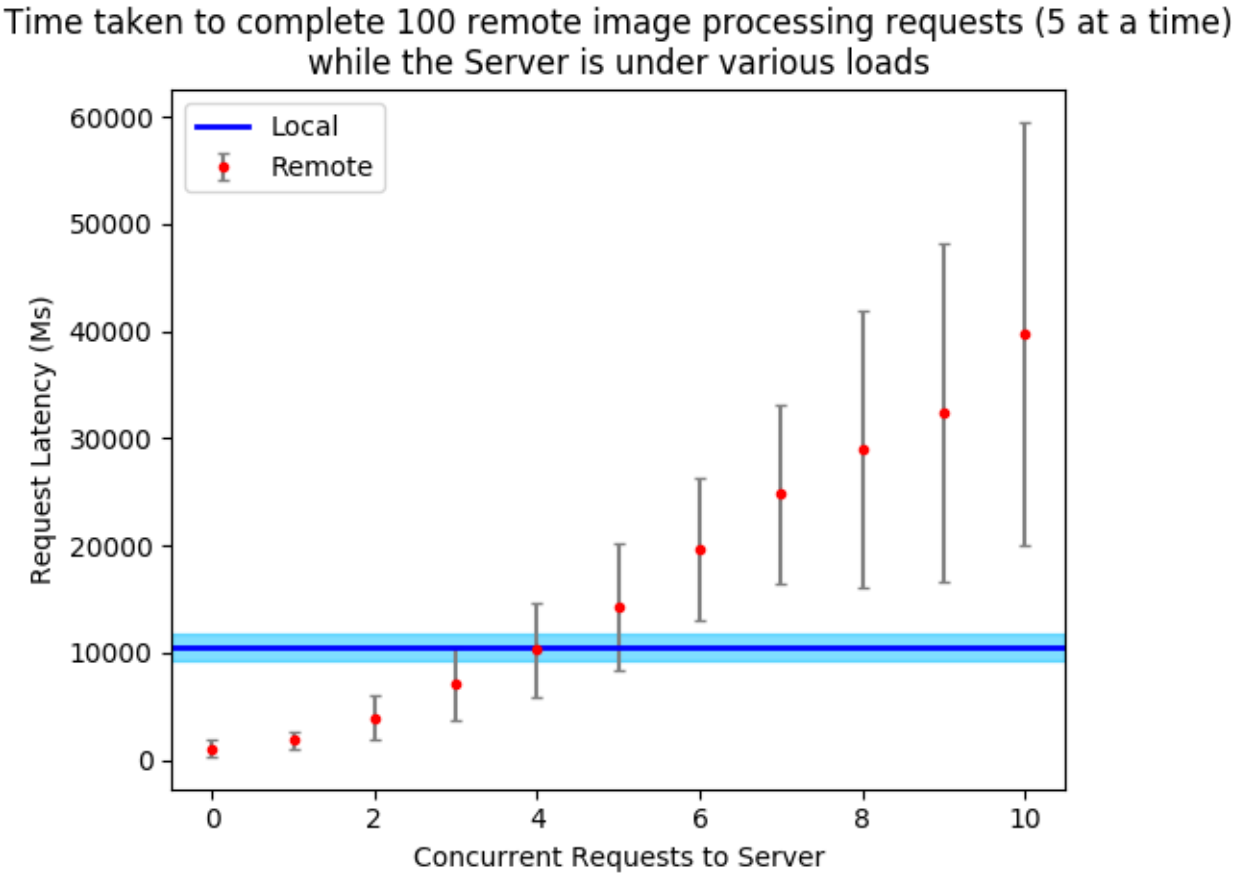


Figure 4.6: Request latencing of remote image processing requests to the server for various server loads compared to doing image processing locally.

4.2.1 Location of Data Processing

The goal of PAM's request policies is to allow the preffered location of data processing to be specified as part of the request. As this is simply a field in the request it is easy for programmers to set it dynamically i.e. based on current load, network conditions or any boolean predicate.

Meeting the second 2 success criteria in Table 4.1 demonstrates how using *PAM* leads to more flexible and robust system designs through granular control of the location of data processing. *EnConsComp* can be used to illustrate this by using a range of computation policies **Add appendix** at the same time with either no handlers registered, only a raw data handler, only a full computation handler ot both. The server can handle either partial or full results and when multiple handlers are registered a choice is made based on the request policy. This essentially allows us to state a preferred mode of operation using the request (i.e. distributed processing) but use the other option as a fallback (i.e. centralised processing). We have seen in the section above the benefit of pushing processing to remote nodes but being able to naturally provide a fallback is increases resilience to failure.

ImPro illustrates that it is not always most efficient to do computation in the cloud. Figure 4.6 shows that we can save time doing computation locally, even on a device with $\frac{1}{5}$ of the memory and CPU power, if the server is under enough load. When evaluating PAM in the *ImPro* scenario it was found that network latency and bandwidth had little effect on compute time (due to the relatively small size of images and relatively low network latencies compared to the time it takes to run the neural net). Similarly, file size had almost no effect as this would only affect the propagation delay when uploading the image to the server which was also drowned out by the computation **add figure**. The main factors which did have an effect were avilable memory and CPU time, whilst a central server is likely to have many more resources than an edge device it is also likely to server may more clients. If some clients could do their processing locally we could reduce load on central servers and so provide lower latencies to those clients who do request server computation. *PAM* provides the flexbility to compute in either location also provides resilience against particularly slow, or low bandwidth networks (or even network failure).

The ability to choose processing location can have an impact of things other than latency. Power is a large concern for mobile edge devices, if clients can a decision regarding where to do computation based on their power level then those with plenty could move computation locally, reduce load on servers, and so reduce latency for those which request central computation. This would mean they need to listen for a response for less time and so may save them even more power.

Future Work

Automating the decison for where to do processing would be an interesting next step. Work on this in the past has profiled local and remote performance as well as network conditions in order to decide dynamically whether to offload indvidual method calls [9]. A simpler extension here would be to work on the granularity ot http requests and use observed request latency and current power levels to make a dynamic decison about the location of processing.

This could be implemented using PAMs dynamic computation policy by only making the local compute method available when we have enough energy to spare. The request polcies can then be used to choose a preferred location based on observed request latency.

4.2.2 What Data Can be Processed

The Data Policies I have implemented allow users to specify transforms which must be applied to database columns before they are used as part of SQL queries. They can do this on a per Privacy Group basis where each Privacy Group contains a list of identifiers for potential requesters. The remainder of this section shows this is expressive enough to implement various privacy mechanisms including k-anonymity, L-diversity and ϵ-differential privacy[3].

[3]These mechanisms are described in §2.5.3

k-anonymity and L-diversity provide protection for anonymised databased against statistical attacks **add refs**. The usual mechanisms for acheiving this are through abstraction (i.e. only giving data at the month granularity as opposed to the day granularity) and suppression of specific records. We also will usually remove columns with unique identifiers such as names, we can achieve this as part of a *PAM* data policy using excluded columns. Many algorithms exist to establish what transforms must be applied and which rows we need to remove (call the set of rows to remove `exclude`). Once this has been done for a table then we can use *PAM* column transforms similar to the one below to implement the generalisations and suppressions.

```
1 func(arg interface{}) (interface{}, bool, error) {
2   if contains(exclude, arg) {
3     return nil, true, nil
4   }
5   return yearGranularity(arg), false, nil
6 }
```

Differential Privacy seeks to maxmimise the information gained from database queries whilst measuring, and minimising, the privacy loss resulting from the query. ϵ-differential is a formalism which ensure the difference between adding or removing a record from a database has bounded effects query results.

There are two common methods for implementing differential privacy. The first, *Randomised Response* [27, 17] stems from the social sciences and acts on the data level and involves randomly deciding between giving a random result and the actual result. This can easily be expressed as a data policy in the form of a column transform such as the one below.

```
1 func(arg interface{}) (interface{}, bool, error) {
2   if rand.Intn(1) > 0 {
3     return arg, false, nil
4   }
5   return randomValueForColumn(), false, nil
6 }
```

This shows how we can implement ϵ-differential privacy as a data policy.

The second method for implementing differential privacy is the Laplace mechanism **??**. This involves adding noise to results of queries in order to ensure ϵ-differential privacy. This is not supported directly by the data policies as the noise which we must add is very specific to the query made, and the data policies aim to provide data protection at the level of SQL tables.

4.3 Applicability to Various Scenarios

Social networks allow users to share information about themselves publically and find information about others. However this requires us to trust the social network provider

with all of our data and even if they allow us to specify complex privacy policies regarding fellow users it does not allow us to hide specific data from them easily. Safebook [10] uses a distributed P2P architecture to help solve this by storing compies of users data on other user's machines only if they are a direct social link, protected with attribute-based encyryption. It's network of servers also allows requests to be offloaded to other machines in the event of high load or failure. A middleware such as *PAM* could make building applications such as this more manageable by expressing the offloading mechanism and computation and request polcies and allowing access based on the requester using data policies. Perhaps if mechanisms such as this were more readily available and easy to implement then data breaches such as **Insert data breach** would be easier to avoid.

LEACH [16] is an energy efficient protocol for wireless sensor communication. It groups sensors into clusters which all report to one clusterhead, this clusterhead may then aggregates data and reports it to a data collection point. As clusterheads are selected randomly it may not have a large amount of battery left, *PAM* or a similar middleware would facilitate implementing a sensor network where either raw data or aggregates can be reported based on both user preference or device power, this could reduce the energy requirements of the network and allow sensors to live for longer and hence collect more data.

4.4 Limitations and Future Work

Plan: This will probably discuss taking measurements periodically to allow the decision for computatin polcies to be made based on i.e. battery life or some other function May say that to run this on IOT or mobile the perhaps implementing in other languages may be required

NOTE: I am condidering removing this section as I discuss the limitations and future work throughout anyway?

Limitations/Future work:

1. May need to implement a mobile friendly version as most of the previous work on offload is mobile centric.
2. Data policies do not support updates if any columns are excluded even if the update is not to that column.
3. Mosco provides a wide range of transforms and it may be easier to specify some privacy preserving mechanisms using these, I could seek to support more specific policies (they could be built on top of the current transforms)
4. I could make it easier to run automatic dynamic offloading decisions, i.e. allowing functions which are run every x seconds to be specified as part of computation policies which can dynamically change the available methods (currently people would have to write a script to do this themselves)
5. Could support any boolen predicate for offloading decisons i.e. rather than speci-

fying Local or Remote we could say Local if it will take you longer than x seconds and Remote otherwise.

6. Although power has been chosen a key motivator for mobile offload in the past, it is a difficult metric to measure and so *PAM*'s effect on it is not explicitly measured. However, some im but the implications for it are discussed at the end of §4.2.1.
7. ...

Chapter 5

Conclusion

As many users become more aware of how companies are acquiring and using their data privacy aware technologies are likely to grow in popularity. *PAM* illustrates not only how we can limit the data we release but but also how we can constrain where it is processed. This benefits the creators of data because they can control what happens to it but also companies who wish to use this data as without controls such as PAM users may simply refuse to give them data as opposed to allowing them reasonable access as they may with PAM.

5.1 Achievements

I have sucessfully implemented a middleware which provides granular control over not only where data is processed, but what data can be processed. I have demostrated its effectiveness and ease of implementation in several key scenarios and shown how state of the art privacy and anonymity mechanisms can be implemented within it.

I have also shown that *PAM* has limited overhead, especialy after the addition of the caching of transformed tables. Furthermore it can deal with transient computation capacities, with an optional dynamic implementation of computation policies.

More generally I have created a piece of software which can be included in any chain of Go middlewares and hence is accessible and extensible.

5.2 Lessons Learned

Building a project of this scale has required greater engineering discipline than any of my previous undertakings. Using deployment tools such as Docker to save time in deployment and isolate applications as well as writing unit tests with good coverage and benchmarking the key parts of applications are all elements of good practice which I have developed. If I were to approrach this project again I may have tried to limit its scope to only solve

the data policy element of it. This would have allowed me more time to explore how more complex transforms could be simply supported and perhaps even to have extended the parser to provide more information about which table a referenced column is from which would allow updates to rows even if some columns are excluded. By the nature of the project being a middleware it is not standalone in nature. It was difficult to choose a small selection of use cases to demonstrate it, especially given each would represent significant implementation time which then could not be spent on the core of the project, the middleware itself. I'm pleased with how I reused code from other sources to speed up the development of examples so that they could inform me on which areas of the middleware to focus on rather than hinder my progress.

Bibliography

[1] Adult Dataset. https://archive.ics.uci.edu/ml/datasets/Adult.

[2] AWS Free Tier. https://aws.amazon.com/free/.

[3] Cloud Computing Law - Christopher Millard - Oxford University Press. https://global.oup.com/ukhe/product/cloud-computing-law-9780199671687?cc=gb&lang=en&.

[4] Corpora of misspellings for download. https://www.dcs.bbk.ac.uk/~ROGER/corpora.html.

[5] Docker. https://www.docker.com/.

[6] Production-Grade Container Orchestration - Kubernetes. https://kubernetes.io/.

[7] What is cloud computing? A beginner's guide — Microsoft Azure. https://azure.microsoft.com/en-gb/overview/what-is-cloud-computing/.

[8] Mehdi Bahrami and Mukesh Singhal. A Light-Weight Permutation Based Method for Data Privacy in Mobile Cloud Computing. In *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pages 189–198, San Francisco, CA, USA, March 2015. IEEE.

[9] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Ch, and Paramvir Bahl. MAUI: Making smartphones last longer with code offload. In *In MobiSys '10: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, pages 49–62, 2010.

[10] Leucio Antonio Cutillo, Refik Molva, Thorsten Strufe, and Tu Darmstadt. *CONSUMER COMMUNICATIONS AND NETWORKING Safebook: A Privacy-Preserving Online Social Network Leveraging on Real-Life Trust.*

[11] Rajneesh Dwevedi, Vinoy Krishna, and Aniket Kumar. Environment and Big Data: Role in Smart Cities of India. *Resources*, 7(4):64, December 2018.

[12] Cynthia Dwork. Differential Privacy. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 1–12. Springer Berlin Heidelberg, 2006.

[13] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric Computing: Vision and Challenges. *ACM SIGCOMM Computer Com-*

munication Review, 45(5):37–42, September 2015.

[14] Petko Georgiev, Nicholas D. Lane, Kiran K. Rachuri, and Cecilia Mascolo. LEO: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking - MobiCom '16*, pages 320–333, New York City, New York, 2016. ACM Press.

[15] Ali Gholami and Erwin Laure. Security and Privacy of Sensitive Data in Cloud Computing: A Survey of Recent Developments. *Computer Science & Information Technology (CS & IT)*, pages 131–150, December 2015.

[16] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. pages 3005–3014, 2000.

[17] Naoise Holohan, Douglas J. Leith, and Oliver Mason. Optimal Differentially Private Mechanisms for Randomised Response. *IEEE Transactions on Information Forensics and Security*, 12(11):2726–2735, November 2017.

[18] Wenlu Hu, Ying Gao, Kiryong Ha, Junjue Wang, Brandon Amos, Zhuo Chen, Padmanabhan Pillai, and Mahadev Satyanarayanan. Quantifying the Impact of Edge Computing on Mobile Applications. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems - APSys '16*, pages 1–8, Hong Kong, Hong Kong, 2016. ACM Press.

[19] Alexei Ledenev. Chaos testing and network emulation tool for Docker.: Alexei-led/pumba, October 2018.

[20] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 24–24, Atlanta, GA, USA, 2006. IEEE.

[21] G. Orsini, D. Bade, and W. Lamersdorf. CloudAware: A Context-Adaptive Middleware for Mobile Edge and Cloud Computing Applications. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 216–221, September 2016.

[22] Syam Kumar Pasupuleti, Subramanian Ramalingam, and Rajkumar Buyya. An efficient and secure privacy-preserving approach for outsourced data of resource constrained mobile devices in cloud computing. *Journal of Network and Computer Applications*, 64:12–22, April 2016.

[23] M. Mazhar Rathore, Awais Ahmad, Anand Paul, and Seungmin Rho. Urban planning and building smart cities based on the Internet of Things using Big Data analytics. *Computer Networks*, 101:63–80, June 2016.

[24] Pierangela Samarati and Latanya Sweeney. Protecting Privacy when Disclosing Information: K-Anonymity and Its Enforcement through Generalization and Suppression. page 19.

[25] Latanya Sweeney. *L. Sweeney, Simple Demographics Often Identify People Uniquely. Carnegie Mellon University, Data Privacy Working Paper 3. Pittsburgh 2000. Simple Demographics Often Identify People Uniquely.*

[26] Dinh Tien Tuan Anh, Milind Ganjoo, Stefano Braghin, and Anwitaman Datta. Mosco: A privacy-aware middleware for mobile social computing. *Journal of Systems and Software*, 92:20–31, June 2014.

[27] S. L. Warner. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309):63–66, March 1965.

[28] Fan Ye, Raghu Ganti, Raheleh Dimaghani, Keith Grueneberg, and Seraphin Calo. MECA: Mobile Edge Capture and Analysis Middleware for Social Sensing Applications. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12 Companion, pages 699–702, New York, NY, USA, 2012. ACM.

[29] J. Zhang, B. Chen, Y. Zhao, X. Cheng, and F. Hu. Data Security and Privacy-Preserving in Edge Computing Paradigm: Survey and Open Issues. *IEEE Access*, 6:18209–18237, 2018.

Appendix A

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Centralised and Decentralised Clouds: managing processing in the context of sensitive data

J. Moxham, St John's College

6 October 2018

Project Supervisor: Dr J. Singh

Director of Studies: Dr R. Mullins

Project Overseers: Prof. A. Pitts & R. Mantiuk

Introduction

Data privacy is a growing issue in Cloud computing [29] [15] as it often requires you to relinquish control of your data. This project aims to create a middleware which supports granular polices for what can be done with your data and who it can be sent to.

The term Cloud Computing refers to the delivery of computing services over the internet. This has become widespread with products such as Amazon Web Services [2] offering a simple and affordable way to do computation in the Cloud without having to manage the individual, real machines which execute your code. The result of a third party managing your data is that you must trust them.

Edge Computing (sometimes referred to as Fog Computing) is a more decentralised form of Cloud computing. It refers to moving some of the processing closer to where the data is generated (at the 'edge' of the network) in order to reduce the centralised computation

requirements and the amount of data sent over the network. When compared to a fully centralised approach this can potentially:

- Alleviate bandwidth requirements by processing some of the data closer to its source and sending aggregates or results to the central Cloud servers
- Allow processing and hence results to occur closer to clients thereby reducing network latency
- Give users greater control of their data by not centralising all of it. This aids privacy by giving edge nodes control of the computation performed on their data and the results which are sent back

In some cases Edge Computing presents a trade-off; computing in a federated manner on lots of nodes may use more resources in total than a fully centralised approach. It may also be that combining results which have had some computation done at the edge may result in responses which contain aggregated data leading to the overall result being of lower quality (although this is not always the case).

The middleware created by this project will have a policy framework which allows data sources to specify what data is allowed to be sent to the cloud and what processing they can perform on data locally. In turn requests (simply for data or for processing of data) will be annotated with a requesting entity and a preferred location for processing (if necessary). The framework will resolve these in order to decide what data to centralise and where to perform processing.

The types of requests to be considered are:

- A central third party requesting data from many nodes. (e.g. aggregating data stored nodes with varying privacy policies)
- Processing that could be performed locally or in the cloud as we already have all the necessary data. (e.g. performing spell check on a document)
- When a request is received requiring data stored on other edge nodes. This is different to the first type as the result is entrusted to an edge node rather than just a central one. (e.g. rating energy consumption relative to others in your area)

Starting point

I have taken second year courses in Networking and Concurrent and Distributed systems which will help me to understand the interaction between a client device and a cloud device. This year I am also taking the Cloud Computing course which will cover virtualisiation as well as other useful concepts for this project.

Load balancing frameworks which distribute blocks of work between devices with different characteristics already exist, a primary example being Kubernetes [6]. However, these do not consider the different privacy concerns of the devices which they run things on. They

have also been aimed at data centres rather than the heterogeneous environment of Edge computing where we would like to manage where computation is done as opposed to always achieving maximum efficiency.

Docker [5] is a state of the art container management tool which allows you to limit the resources allocated to any given container. I will simulate various devices and workloads using containers with different restrictions.

Resources required

For this project I will mainly use my own Dell XPS 15 running Ubuntu Linux. This Laptop has 16GB of RAM, a 250GB SSD and a quad-core 2.60GHz Intel I7-6700HQ CPU.

Backups of the code will be done using git and GitHub whenever I work on the project. I will also use GitHub to back up any documents I create including this proposal, my log book and my Dissertation.

If my laptop were to fail I have money set aside to replace it. In the meantime I could easily migrate to the MCS machines due to the virtualised nature of the computation required to run the planned tests.

An example dataset containing sensitive information which a node may not want to release raw is the Census Income Data Set [1](also known as the "Adult" dataset"). This could potentially be used for the first example and is freely available.

Many datasets of misspellings and text with poor spelling are available [4]. These can be used to test and evaluate the spell check task.

Work to be done

The main piece of software which needs to be developed is a middleware with a policy framework which will manage interactions between centralised clouds and decentralised clouds in the context of homogeneous privacy policies and processing ability. I will then implement example use cases using the middleware to demonstrate its effectiveness.

Nodes with data should be able to specify granular policies for:

- What data can be sent to other (specific) entities
- What processing can be done on the data locally
- If any filtering and preprocessing needs to be done on the data before it can leave (e.g. removing unique identifiers)

Requests for data should be annotated with proof of the identity of the requester and requests for processing should additionally contain the preferred location of processing.

The project breaks down into the following sub-projects:

1. Build a middleware to support the policies listed above
2. Implement the example where a central node performs computation over data stored on many edge nodes. This will allow evaluation of how successful the middleware is at supporting heterogeneous privacy policies whilst still aggregating results.
3. Implement spell check as a request for the middleware. This will allow processing to be performed centrally or locally based on:
   - Where the document is stored
   - Whether the device with the document on it will allow it to leave (and if it requires it to be encrypted)
   - Whether the device has the capacity to do the spell check locally
4. Implement a power consumption check as a request for the middleware. This will request power consumption from other nodes near to the requester and then either:
   - Aggregate the data locally to inform the requester if their energy consumption is too high
   - Give the raw data back to the requester and allow it to do the processing

   This will depend on the policies of the sources of the data. It may be that some of the reported statistics are actually aggregates themselves either due to the preference of the central cloud to push computation closer to data or the requirement of edge nodes to only allow aggregated data to leave. I will randomly generate data on the edge devices to simulate real power consumption data.

Success criteria

The goals of the project are:

1. To develop an edge orientated middleware with a policy framework allowing granular control over data and processing of data
2. Assess its applicability in various scenarios

To evaluate the middleware I will compare the functionality and performance for each of the example tasks to a baseline approach which does all computation centrally.

We compare to the baseline based on:

- The amount of time taken to fulfil the request. This can be measured using just one machine by treating each resource constrained docker container as a single machine running on the network. Solutions already exist for simulating various network conditions locally [19]

- The quality of the results, this is defined for each example as:
  - *Central node aggregating data from many nodes & Power consumption:* the quality of the aggregated data compared to calculating it over all of the raw data
  - *Spell check:* the proportion of incorrect spellings found (this may vary due to the amount of time we can spend spell checking or the complexity of the algorithm we can reasonably run). This is less important than the total time for the request in this use case

The decentralised approaches can be performed with various:

- Combinations of data policies
- Processing power ratios between central and edge devices
- Memory ratios between central and edge devices
- Amounts of local data

This will allow me to establish the conditions in which the policy provides the best performance and privacy characteristics.

The project will be considered a success if:

- The middleware supports all 3 examples with various combinations of policies chosen by nodes with data
- I can establish the conditions under which the policy framework provides the best performance and privacy characteristics
- I can indicate potential applications for a framework similar to the one I build

Possible extensions

If I achieve my main result early I will consider:

1. Extending the type of polices which are supported. This could include:
   - Load balancing
   - Considering audit requirements
   - Dynamically changing privacy and security characteristics
2. Extending the policy framework to support streams of data. In this case we can measure the reduction in required bandwidth and central processing by pushing some of the aggregation of data to the edge of the network
3. Extending the middleware to run on intermediate devices between clients and the cloud server itself. An example where this may be useful is if sensitive data should

never leave a company's own infrastructure. In this case we could send data from personal computers to a company server, processing could be done here and only results ever returned to the cloud

Timetable

Planned starting date is 19/10/2018.

1. **Michaelmas weeks 2–4**
   - Begin writing a simple middleware supporting sending requests and decoding responses
   - Find a dataset and specific use case for the first example
2. **Michaelmas weeks 5–6**
   - Finish the simple middleware functionality
   - Add metrics to the middleware to support testing and measurement of bandwidth and request latency
   - Design the policy framework
3. **Michaelmas weeks 7–8**
   - Finalise the policy framework
   - Begin implementing the framework in the middleware
4. **Michaelmas vacation**
   - Finish the policy framework
   - Complete the example of a central node aggregating data from many edge nodes:
     (a) Implement the example using the middleware
     (b) Test it to see if we get a processing time improvement with different central to edge processing power ratios if:
       – All data is stored centrally so we run aggregation locally
       – The data is split across many nodes some of which perform initial aggregation locally
     (c) Write a suite of tests to generate statistics for analysis
5. **Lent weeks 0–2**
   - Write progress report
   - Use test data to create graphs to compare approaches

6. **Lent weeks 3–5**
    - Implement the spell check example
7. **Lent weeks 6–8**
    - Define a model for generating random power consumption data
    - Implement the power consumption example
8. **Easter vacation:**
    - Generate data to compare approaches for all new examples
    - Write Dissertation main chapters
    - Possible extensions
9. **Easter term 0–2:** Further evaluation and complete dissertation
10. **Easter term 3:** Proof reading and then an early submission so as to concentrate on examination revision

Bibliography

[1] Adult Dataset. https://archive.ics.uci.edu/ml/datasets/Adult.

[2] AWS Free Tier. https://aws.amazon.com/free/.

[3] Cloud Computing Law - Christopher Millard - Oxford University Press. https://global.oup.com/ukhe/product/cloud-computing-law-9780199671687?cc=gb&lang=en&.

[4] Corpora of misspellings for download. https://www.dcs.bbk.ac.uk/~ROGER/corpora.html.

[5] Docker. https://www.docker.com/.

[6] Production-Grade Container Orchestration - Kubernetes. https://kubernetes.io/.

[7] What is cloud computing? A beginner's guide — Microsoft Azure. https://azure.microsoft.com/en-gb/overview/what-is-cloud-computing/.

[8] Mehdi Bahrami and Mukesh Singhal. A Light-Weight Permutation Based Method for Data Privacy in Mobile Cloud Computing. In *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pages 189–198, San Francisco, CA, USA, March 2015. IEEE.

[9] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Ch, and Paramvir Bahl. MAUI: Making smartphones last longer with code offload. In *In MobiSys '10: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, pages 49–62, 2010.

[10] Leucio Antonio Cutillo, Refik Molva, Thorsten Strufe, and Tu Darmstadt. *CONSUMER COMMUNICATIONS AND NETWORKING Safebook: A Privacy-Preserving Online Social Network Leveraging on Real-Life Trust.*

[11] Rajneesh Dwevedi, Vinoy Krishna, and Aniket Kumar. Environment and Big Data: Role in Smart Cities of India. *Resources*, 7(4):64, December 2018.

[12] Cynthia Dwork. Differential Privacy. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 1–12. Springer Berlin Heidelberg, 2006.

[13] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric Computing: Vision and Challenges. *ACM SIGCOMM Computer Com-*

munication Review, 45(5):37–42, September 2015.

[14] Petko Georgiev, Nicholas D. Lane, Kiran K. Rachuri, and Cecilia Mascolo. LEO: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking - MobiCom '16*, pages 320–333, New York City, New York, 2016. ACM Press.

[15] Ali Gholami and Erwin Laure. Security and Privacy of Sensitive Data in Cloud Computing: A Survey of Recent Developments. *Computer Science & Information Technology (CS & IT)*, pages 131–150, December 2015.

[16] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. pages 3005–3014, 2000.

[17] Naoise Holohan, Douglas J. Leith, and Oliver Mason. Optimal Differentially Private Mechanisms for Randomised Response. *IEEE Transactions on Information Forensics and Security*, 12(11):2726–2735, November 2017.

[18] Wenlu Hu, Ying Gao, Kiryong Ha, Junjue Wang, Brandon Amos, Zhuo Chen, Padmanabhan Pillai, and Mahadev Satyanarayanan. Quantifying the Impact of Edge Computing on Mobile Applications. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems - APSys '16*, pages 1–8, Hong Kong, Hong Kong, 2016. ACM Press.

[19] Alexei Ledenev. Chaos testing and network emulation tool for Docker.: Alexei-led/pumba, October 2018.

[20] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 24–24, Atlanta, GA, USA, 2006. IEEE.

[21] G. Orsini, D. Bade, and W. Lamersdorf. CloudAware: A Context-Adaptive Middleware for Mobile Edge and Cloud Computing Applications. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 216–221, September 2016.

[22] Syam Kumar Pasupuleti, Subramanian Ramalingam, and Rajkumar Buyya. An efficient and secure privacy-preserving approach for outsourced data of resource constrained mobile devices in cloud computing. *Journal of Network and Computer Applications*, 64:12–22, April 2016.

[23] M. Mazhar Rathore, Awais Ahmad, Anand Paul, and Seungmin Rho. Urban planning and building smart cities based on the Internet of Things using Big Data analytics. *Computer Networks*, 101:63–80, June 2016.

[24] Pierangela Samarati and Latanya Sweeney. Protecting Privacy when Disclosing Information: K-Anonymity and Its Enforcement through Generalization and Suppression. page 19.

[25] Latanya Sweeney. *L. Sweeney, Simple Demographics Often Identify People Uniquely. Carnegie Mellon University, Data Privacy Working Paper 3. Pittsburgh 2000. Simple Demographics Often Identify People Uniquely.*

[26] Dinh Tien Tuan Anh, Milind Ganjoo, Stefano Braghin, and Anwitaman Datta. Mosco: A privacy-aware middleware for mobile social computing. *Journal of Systems and Software*, 92:20–31, June 2014.

[27] S. L. Warner. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309):63–66, March 1965.

[28] Fan Ye, Raghu Ganti, Raheleh Dimaghani, Keith Grueneberg, and Seraphin Calo. MECA: Mobile Edge Capture and Analysis Middleware for Social Sensing Applications. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12 Companion, pages 699–702, New York, NY, USA, 2012. ACM.

[29] J. Zhang, B. Chen, Y. Zhao, X. Cheng, and F. Hu. Data Security and Privacy-Preserving in Edge Computing Paradigm: Survey and Open Issues. *IEEE Access*, 6:18209–18237, 2018.