LARMAS

Language Resource Management System

Prepared by:

TINOTENDA CHEMVURA

CHMTIN004

Department of Electrical Engineering

University of Cape Town

Prepared for:

DR MOHOHLO S. TŠOEU

Department of Electrical Engineering

University of Cape Town

November 2017

Submitted to the Department of Electrical Engineering at the University of Cape Town in partial fulfilment of the academic requirements for a Bachelor of Science degree in Electrical and Computer Engineering.

Key Words: Client-server, web-application, natural-language-processing, speech-recognition, machine-translation, distributed-system, cloud-computing

DECLARATION

1. **I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.**
2. **I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this final year project report from the work(s) of other people, has been attributed and has been cited and referenced.**
3. **This final year project report is my own work.**
4. **I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as their own work or part thereof**

Name:	**Tinotenda Chemvura**		
Signature:	[illegible]	**Date:**	13 November 2017

Acknowledgements

I would firstly like to thank Jesus for his blessings and guidance through my journey at UCT.

I would like to extend my gratitude to my supervisor Dr Mohohlo S. Tšoeu whose door was always open to assist me throughout the project. His enthusiasm and guidance kept me focused and on the right path, and I will forever be grateful for the advice and work ethic I received from him, habits I will continue to apply throughout my life and career.

I would like to thank my mother and father for everything they have done for me. For sacrificing so much to put me through university and being my cheerleaders throughout the toughest times. My best friend Komborero Mtambirwa helped me proofread my thesis and gave advice on how to conclude the report.

Finally, I would like to thank the developer community at the StackExchange fora, namely StackOverflow, Stats, Math, OpenData and AskUbuntu, the forums on GitHub project repositories that I used and the various online developer forums I may have forgotten. These communities provided a great deal of support especially during the software implementation stage of the report where I was using several new technologies with a limited amount of time I had.

Abstract

Natural Language Processing (NLP) is a major area of artificial intelligence that is concerned with the ability of a computer to recognise, translate and understand human speech. Research in NLP over the past 70 years has improved the algorithms used in NLP, moving from the simple matching of words and rules in machine translation to using statistical models and neural networks. Modern methods of machine translation and automatic speech recognition (major fields of study in NLP) use machine learning algorithms and neural networks, which require a large amount of speech and text training data to produce accurate results. Therefore, there is a need for a way to efficiently collect, store and access language resources for NLP.

LARMAS (Language Resource Management System) is a client-server web application that has been designed to efficiently collect language resources for NLP and manage easy access to these resources for third-party NLP engines and projects. The project was broken down into three phases. The first phase lasted the first 6 weeks. This phase was characterised by research into machine translation and automatic speech recognition to determine what data needed to be collected. After determining the data that needed to be collected, research was done on the different literature, tools and web technologies needed to implement, comparing them to aid with the overall design of the system. LARMAS will store prompts which will be distributed to users to record and annotate before sending their contributions to the system. Users will also be able to translate these prompts into any of the languages they speak.

A relational database schema was designed to facilitate managing users and authentication, storing prompts, annotations and translations, and creating relations to allow for parallel text to be extracted. Several application architectural designs were investigated and Model-View-Controller layered architecture was chosen as the main application architecture for LARMAS. However, this architecture borrowed some concepts from the other architectures that were investigated, namely, the space-based and microkernel architectures. Object storage will be used for storing the media files uploaded to LARMAS by the contributors. A REST API was created as the interface that will be used by third party applications to interact with LARMAS. The endpoints were well documented and put on the home page of LARMAS. A browsable API was also created to enable developers to interact with the API in a browser without having to write code to test its functionality. Different server architectures that could be used to deploy LARMAS were investigated and the recommended architecture is highly scalable, both vertically and horizontally.

The second phase of the project lasted 4 weeks and this is when a prototype for LARMAS was built in Django using Python 3.5. The development process was modelled after the Scrum framework where work was split into small “actions” which last 3 to 6 days (instead of 2-3 weeks in Scrum). The development of LARMAS was rapid and test driven, using TravisCI for Continuous Integration and Git for version control.

The last two weeks of the project is when tests and experiments on LARMAS were carried out. These tests tested both the functional and non-functional requirements of LARMAS and they included comparing databases, measuring response times and scalability and comparing load balancing algorithms. After discussions of the test results, conclusions were made, addressing the project objectives that were laid out in the introduction.

Contents

List of Figures

List of Tables

Glossary

ACID	Atomicity, Consistency, Isolation, Durability
AI	Artificial Intelligence
API	Application Programming Interface
ASP	Active Server Pages
ASR	Automatic Speech Recognition
AWS	Amazon Web Services
BSD	Berkeley Software Distribution
CAT	Central Africa Time (UTC +02:00)
CDN	Content Distribution Network
CoC	Convention over Configuration
CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
DB	Database
DBMS	Database Management System
EC2	Elastic Cloud Computing
GNU	"GNU's not Unix" *(recursive acronym)*
GNU AGPLv3	GNU Affero General Public License version 3
GraphQL	Graph Query Language
HMM	Hidden Markov Models
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
iSCSI	Internet Small Computer Systems Interface
JSON	JavaScript Object Notation
LARMAS	Language Resource Management System
LM	Language Model
LTS	Long Term Support
MT	Machine Translation
MVC	Model View Controller
NFR	Non-Functional Requirements
NLP	Natural Language Processing
NoSQL	Not Only SQL
POJO	Plain Java Object
POSIX	Portable Operating System Interface
PPL	Perplexity
RAID	Redundant Array of Independent Disks
RBMT	Rule-Based Machine Translation
RDBMS	Relational Database Management System
RDS	Relational Database Service
RTT	Round Trip Time
REST	Representational State Transfer
SMT	Statistical Machine Translation
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
URI	Universal Resource Identifier
URL	Universal Resource Locator
UTC	Coordinated Universal Time
WER	Word Error Rate
XML	eXtensible Mark-up Language

1. Introduction

1.1. Abstract

This chapter gives a motivation for conducting research work in this thesis. It briefly explains how computers are used in natural language processing (NLP) and the challenges currently faced with collection data for NLP. A solution is then proposed along with the main objectives and followed by the scope and limitations of the thesis.

1.2. Natural Language Processing (NLP)

NLP is a major area of artificial intelligence that is concerned with the ability of a computer to automatically recognise, translate and understand human speech as it is spoken. Machine Translation (MT) was the first focus of NLP. It began in the late 1940s with one of the most significant milestones in the world of MT being the publication of Warren Weaver's memorandum titled "Translation" in July 1949 [1]. This memorandum started off with a brief overview of how computers had already been used to translate foreign text with Richens and Booth's use of punch cards to produce word-for-word translations of scientific abstracts. Weaver put in four proposals in his memorandum. These were [1]:

1. The problem of words that have multiple meanings can be tackled by the examination of text in the immediate context.
2. Weaver hypothesised that translation could be addressed as a problem of formal logic. This assumed that there are logical elements in human languages.
3. Cryptographic methods could be applied to translation. This would treat the foreign text as encrypted code which can then be decrypted using methods developed in cryptography.
4. "Linguistic Universals", which are underlying properties that are possessed by all human languages, could be exploited to make translation straightforward.

Early methods of MT tackled translation as word-for-word dictionary-based processing with ambiguity being resolved using local context. These early approaches to NLP were very inaccurate. Scientists soon realised that human language was much more complex than code breaking and computers at that time simply could not handle the complexity. These days, computers are much more powerful and scientists have developed computer systems that can "learn" about data that is fed to them. In fact, computers are now able to automatically translate between some of the popular languages, if they are fed the right data.

1.3. Machine Translation (MT)

MT is one of the research areas of NLP that studies the use of software to automatically translate text or speech from one language to another [2]. This is done using machine learning, corpus statistical and neural techniques which are in continuous development to produce better translations. MT is classified as *AI-Complete* [3], meaning that is it one of the most difficult problems that can be solved using AI since it requires all the different types of knowledge that humans possess (grammar, facts about the world, semantics, etc.)

Modern MT techniques like Gaussian Mixture Model and Hidden Markov Model require a large amount of data to be effective [4]. This data is in the form of "parallel corpora", which are large collections of text in one language that is aligned and placed alongside its translations in other languages. Popular western languages like English, French, etc. have very large and easily accessible parallel corpora like OPUS. However, these datasets often do not have nearly enough data on South

African (and African) languages to create reliable, parallel text with an English corpus or corpora for other languages.

1.4. Speech Recognition

Also known as Automatic Speech Recognition (ASR), this is another research area of NLP that is concerned with the ability of computers to recognise words and phrases of spoken language and converting them to text (or a format that can be used by the machine itself). Because this requires a wide range of areas of human knowledge, ASR's difficulty is classified as *AI-Complete* [3].

There are several approaches used in ASR. A few examples of these are Dynamic Spectral Warping, Statistical Dynamic Time Warping and Hidden Markov Models just to mention a few. All these methods require large amounts of data to be efficient [5]. More data creates more accurate acoustic models which are better for training speech-recognition systems. Acoustic Models are used to statistically represent phonemes in a language. The model is created by taking a speech corpus and using training algorithms to create statistical representations for each phoneme in a language. The larger a speech corpus is, the more accurate the acoustic model will be. Fewer errors are encountered when predicting words based on the words that come before it for large speech corpora [5].

1.5. Data Collection for NLP

The types of data needed for NLP take the form of text, audio and video recordings and signals from sensors used in transcribing sign-language. Most researchers in the NLP will refer to one or more of the various open-source datasets. Some of these datasets include:

- **WikiText** [6]: Language modelling text corpus, from Wikipedia articles, that are curated by Salesforce MetaMind.
- **OPUS** [7]: An open source collection of translated texts from the internet.
- **Broadcast News** [8]: text dataset used for next-word-prediction. Has a license restricting usage for only research purposes.
- **Google Books Ngrams** [9]: open-source text dataset built from Google Books which offers a simple method to explore when a word first entered wide usage.
- **VoxForge** [10]: An open-source English speech dataset in which contributors contribute by recording short phrases via a web application. It also provides HTK, Julius and Sphinx acoustic models.
- **TED-LIUM Corpus** [11]: An open-source corpus made from audio talks and transcriptions of 1,495 TED talks.

Some projects that have special requirements for the data they need will pay companies like Appen [12] and Globalme [13] which specialise in data collection that will be specially prepared according to noise, language, accent and management requirements.

Attempts have been made to create effective methods to collect open data. Open Speech Recording is an open-source project by AIY team that focuses on collecting single spoken English words and then release the data under an open-source license. Data is collected via a web application which prompts contributors to say out 135 words that appear on the screen [14]. *Lwazi*, a South African government-funded project that ran from 2006 to 2009. It collected voice recordings to create corpora for the 11 official spoken languages in South African. Data was collected through telephone calls where contributors would call the service and follow the voice prompts to record their voices for the project [15]. Using this method, Lwazi managed to collect 76 hours of audio in 4 years.

1.6. Problem Statement

Human language processing systems require large amounts of speech, text and sign-language data to produce accurate results. Therefore, there is a need for a way to efficiently collect, store and access language resources for natural language processing.

1.7. Problems with current approaches

The problems faced with current approaches to collection and management of speech and text resources are:

- **Not enough data.** Acoustic Models and Language Models are used in ASR to statistically represent phonemes (for the former) and word sequences (for the latter). These are built using the language's speech corpus. The larger this corpus is, the more accurate the ASR engine will be for speech-to-text translations. Unfortunately, open-source speech corpora are not large enough to create accurate acoustic and language models. This is especially true for African languages which have very small datasets compared to some western languages like English, French and German just to mention a few [15].
- **Restricted access to language resources**. Most ASR and other NLP projects, including open-source ones, have very limited access to the language resources they use. In most cases, this is because the resources would have been licensed to them by third-party suppliers who put restrictions on direct access to these resources.
- **Current data collection methods are not scalable.** Current data collection methods have a simple architecture that can be described as:
    - Clients upload the raw recordings to a server.
    - Researchers manually process these recordings to make sure they are valid for their needs.
    - The recordings are formatted and stored in a database.

  These approaches work for small-scale data collection for research purposes but it is very slow and inefficient for large-scale data collection. There are bottlenecks at the receiving end where researchers manually process the raw files and at the storage of the actual data. They cannot handle large-scale collection of data (i.e. if thousands of researchers had participated) and a lot of labour is also wasted when the researchers go through raw data that is not valid for storage.
- **Current data collection methods are thin-client based**. When collecting data for a speech corpora (or video in the case of sign language), every recording must be processed to make sure that it is in the right format, has the right Signal-to-Noise ratio, and that it is transcribed properly (words are aligned to the recording). Current data collections methods do not allow any of these processes to happen on the client-side before transmitting the data over to the server for processing and storage. In turn, a lot of the collected data is invalid and internet bandwidth is wasted. Historically, client-side devices (voice recorders and mobile phones) were not powerful enough to perform some of the processing required on the raw data.

1.8. Proposed Solution

The aim of this project is to develop an open-source client-server system for language resource management; The **Language Resource Management System** (LARMAS). This system will be used to efficiently collect language resources (corpora, sign-language and text) and will allow easy access to the resources for NLP engines and front-end clients (mobile, desktop and web applications).

1.9. Objectives

This project will have the following objectives:

1. **Build a system to collect language resources for NLT**. The system must allow for efficient collection of language resources to be used for NLT. These resources could be text, audio, video or other formats.
2. **Highly scalable**. To allow for large-scale collection of data, the system should be able to handle the extra load when multiple users are using the system and as the workload grows over time.
3. **Access to the language resources**. Users of this system must have access to the resources it manages. Users will include NLP researchers and NLP tools and engines that need access to language resources to function.
4. **Must be a smart-client and/or thick-client based system.** The system must allow the clients to be able to do some of the processing required on the raw data before it is transmitted to the storage server.
5. **Further development**. The project must provide enough documentation and support to allow for further development.
6. **Open-source.** The project must be open-source. All the tools and resources used in this project should be open-source and the project itself should be licensed accordingly to allow for further development by any interested persons.

1.10. Scope and Limitations

1.10.1. Scope

This project covers the research, design and implementation of a prototype of a client-server system that will be used to collect and manage language resources for NLP. Implementation will be done up to the alpha phase of the software development lifecycle. It will not contain all the features that the final product is envisioned to have and testing phase will only go up to and including black-box testing. The system should specify how resources should be uploaded onto it. Language resources will also be made available to third-party systems and applications while protecting this data from unwanted changes modification. The system will also be designed to be highly scalable to allow for large-scale data collection.

1.10.2. Limitations

There are only 12 weeks allocated to this project, therefore, certain features will not be optimised to allow for completion of the project. Actual data collection will not take place during this project. For testing purposes, language resources from other open-source databases will be used instead. There is a strict budget for this project, therefore, some features may not be fully tested or implemented. Only an alpha version of this system will be produced because of the lack of capital and time to produce a beta or production version.

1.11. Plan of Development

The first 6 weeks of the project will be used for research and designing the system. The research will include research into the different methods of MT and ASR to determine the type of data that needs to be collected. After this, there will be a literature review on the different technologies and solutions that will be used to design and implement a prototype for LARMAS. The software implementation of LARMAS will begin in the 7th week and 4 weeks will be spent on developing the software. The last 2 weeks will be spent carrying out tests and experiments on the system to verify its functionality and any hypotheses that may arise during the project. The results will be discussed and conclusions will also be made in the final 2 weeks of the project. Figure 1 is a Gantt chart of the project.

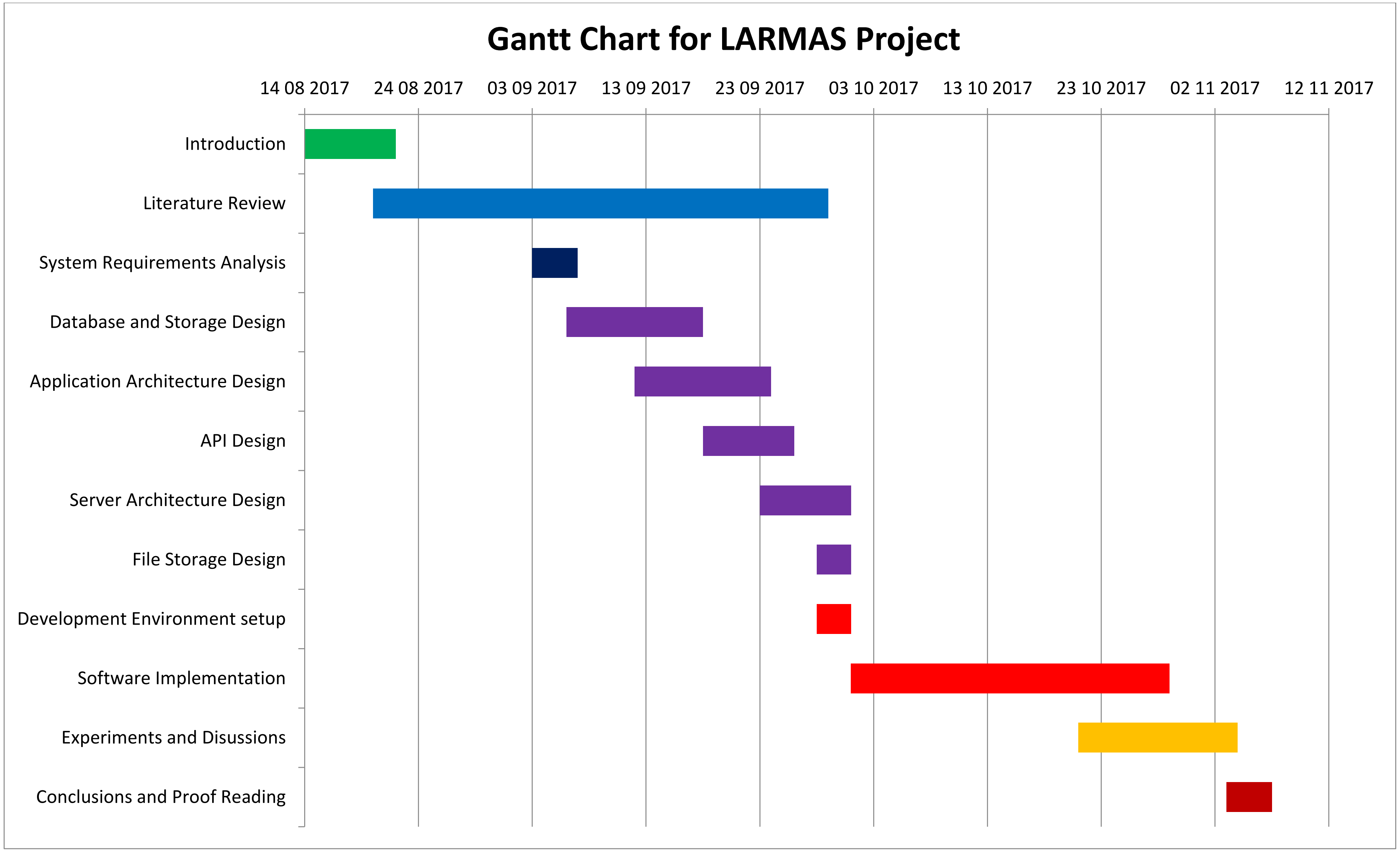


FIGURE 1: GANTT CHART FOR LARMAS PROJECT

1.12. Report Outline

1. Introduction

This chapter gives a motivation for conducting research work in this thesis. The motivation briefly explains how computers are used in natural language processing and challenges currently faced with collection data in NLP. A solution is then proposed along with the main objectives and followed by the scope and limitations of the thesis.

2. Literature Review

This chapter investigates the different types of data required for MT and ASR, the two major types fields of NLP that will be covered in this project and determine the data that will need to be collected and stored on the LARMAS system. The chapter then goes introduces the reader to the concept of web applications and defines what they are and used for. This is then followed by the literature for the different technologies and concepts that will determine the structure, functionality and performance of the LARMAS web application.

3. Research Design

This chapter focuses on the design and implementation of LARMAS. It starts off by defining the functional and non-functional requirements of the system. This is followed by a section describing the development environment, where all the tools used to develop LARMAS are summarised and briefly explaining how they were used. The chapter then goes on to explain how the database, architecture, API and storage systems were designed to aid the management and distribution of prompts and the collection of annotations and translations. The last section explains how LARMAS was designed to meet the non-functional requirements that were defined at the beginning of the chapter.

4. Tests and Experiments

This chapter describes the test and experiments that were carried out on LARMAS. It starts off by describing the testing environment and the tools used for the tests. It then describes in detail, the four groups of tests that were carried out.

5. Results and Discussions

This chapter discusses the results of the tests and experiments carried out in Chapter 4. Results from each experiment are presented in the form of graphs, screenshots and tables, each with comments explaining what they represent. Discussions and conclusions about each experiment.

6. Conclusions

This chapter concludes the report on LARMAS. It reviews each of the objectives set out in the introductions and explains how each objective was achieved.

7. Recommendations

This chapter discusses the results of the tests and experiments carried out in Chapter 4. Results from each experiment are presented in the form of graphs, screenshots and tables, each with comments explaining what they represent. Discussions and conclusions about each experiment.

2. Literature Review

2.1. Abstract

This chapter investigates the different types of data required for MT and ASR, the two major types fields of NLP that will be covered in this project and determine the data that will need to be collected and stored on the LARMAS system. The chapter then introduces the reader to the concept of web applications and defines what they are and used for. This is then followed by the literature for the different technologies and ideas that will be used to design and determine the structure, functionality and performance of the LARMAS web application.

2.2. Translation

Two of the most common approaches to machine translation will be reviewed for this project. These approaches will be studied to determine what kind of data they would need access to in order to perform the necessary operations for a successful translation.

2.2.1. Rule-Based Machine Translation (RBMT)

RBMT uses linguistic information, semantics, syntactic and morphological properties, from both the source and target languages to build linguistic rules that will be used for the translation. In RBMT, there are three stages of machine translation [16]. These are **analysis,** which extracts linguistic information from the source text, **transfer**, which transfers the source text into the syntactic and semantic structure of the target language, and **synthesis**, which replaces the constituents in the source text with the target language equivalents.

Figure 2 is a Bernard Vauquois'Triangle that shows the different approaches to RBMT:

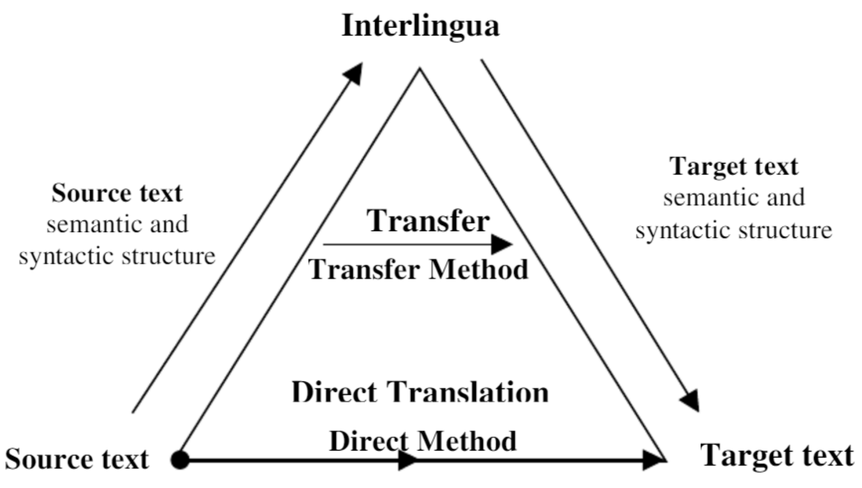


FIGURE 2: BERNARD VAUQUOIS' TRIANGLE SHOWING THE DIFFERENT METHODS OF RULE-BASED TRANSLATION

With the **Direct Approach**, words and phrases in the source language are translated directly (using direct mapping) to the target language without going through an intermediary representation. **Interlingua approach** aims to approach the height of the Bernard Vauquois' Triangle, an intermediary, interlingua-based system which is an abstract representation of the text. An example of an interlingua is Universal Networking Language proposed by the United Nations University in 1996. Finally, the

Transfer Approach does not reach the top of the Bernard Vauquois' Triangle but reaches an appropriate level of transfer before translating to the target text [16].

The data used in RBMT comes from the following sources:

- Language dictionaries for both the source and target languages for the morphological analysis.
- Bilingual dictionaries to convert source language words into the target language.
- Grammar dictionaries/libraries for the syntactical and semantic analysis.

2.2.2. Statistical Machine Translation (SMT)

SMT performs translation by using statistical translation models whose parameters are generated from analysis of parallel corpora. The basic idea of SMT can be defined using the following mathematical relationship [17]:

Most probable translation $\check{T}$ for a given source sentence S and target language T is given by:

$$\check{T} = argmax_T P(T|S)$$
$$Using\ Bayes\ Theorem\ we\ get$$
$$P(T|S) = \frac{1}{P(S)} \times P(S|T)P(T)$$

Where $P(S|T)$ is the **translation model** which represents the probability that the source string is the translation of the target string, and $P(T)$ is the **language model** which represents the probability of seeing that target language string. $\check{T}$ is given by picking the translation that give the highest probability therefore:

$$\check{T} = argmax_T P(T|S) = argmax_T P(S|T)P(T)$$

The first translation models relied on the translation probabilities of individual words and did not consider the idiosyncratic expressions. More recent models now use entire phrases instead. Translation models are calculated using different statistical methods like Hidden Markov Models and IBM Alignment Models. These models use unsupervised learning, meaning that the system is not given any examples of what the output is supposed to be but tries to find which word alignments give best explain the observed system [4]. By splitting a source and target sentences into phrases, the translation model can be expressed as:

$$P(T|S) = \prod_{i=1}^{N} \emptyset(s_i|t_i) \cdot d(a_i - b_{i-1})$$

Where $\emptyset(s_i|t_i)$ is the phrase probability and $d(a_i - b_{i-1})$ is the distortion probability (distortion is the relative distance between the phrase positions in the two languages). Figure 3 shows how phrases between a phrase in French (S) and ones in Norwegian are mapped to each other.

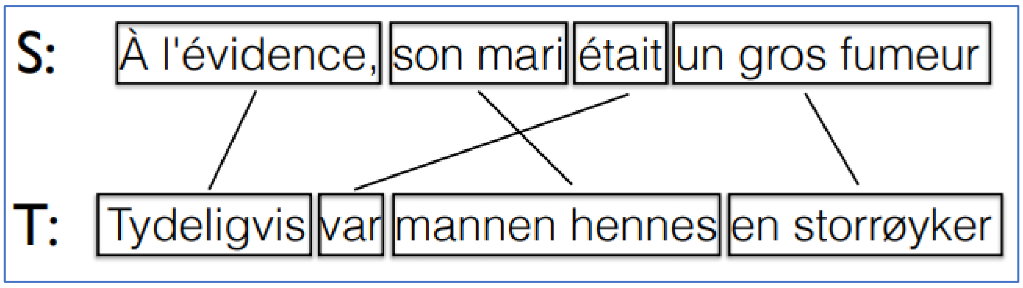


FIGURE 3: TRANSLATION BETWEEN FRENCH AND NORWEGIAN SENTENCES. THE SENTENCES READ "APPARENTLY, HER HUSBAND WAS A HEAVY SMOKER." [17]

The language model encodes the fluency of the target language i.e. how the translated words and phrases will be ordered in the target language. For a word sequence of {w_1, w_2, w_3, …, w_N} in the target language, the language model is expressed as N-grams where:

$$P(w_1^N) = \prod_{k=1}^{N} P(w_k | w_{k-N+1}^{k-1})$$

As seen from the expression above, the probability of each word occurring depends on the words occurring before it. The N-gram probabilities can be estimated from a large amount of monolingual data using mostly bigrams or trigrams. Once the translation and language models are obtained, a **decoder** is used to construct all the possible translations and then searches in this space of possible translations for the most probable one.

To achieve the best possible translations using SMT, the following data will be needed:

- Large amounts of parallel text to be used as training data. Unsupervised learning requires a lot of training data to create accurate translation models.
- Large amounts of monolingual text to create language models that create translations with high fluency.

2.3. Automatic Speech Recognition (ASR)

The most common approach to ASR today is the probabilistic approach where a speech signal corresponds to a word or phrase with a certain probability. This probability is calculated using acoustic properties of speech, knowledge about linguistic structures and pronunciations.

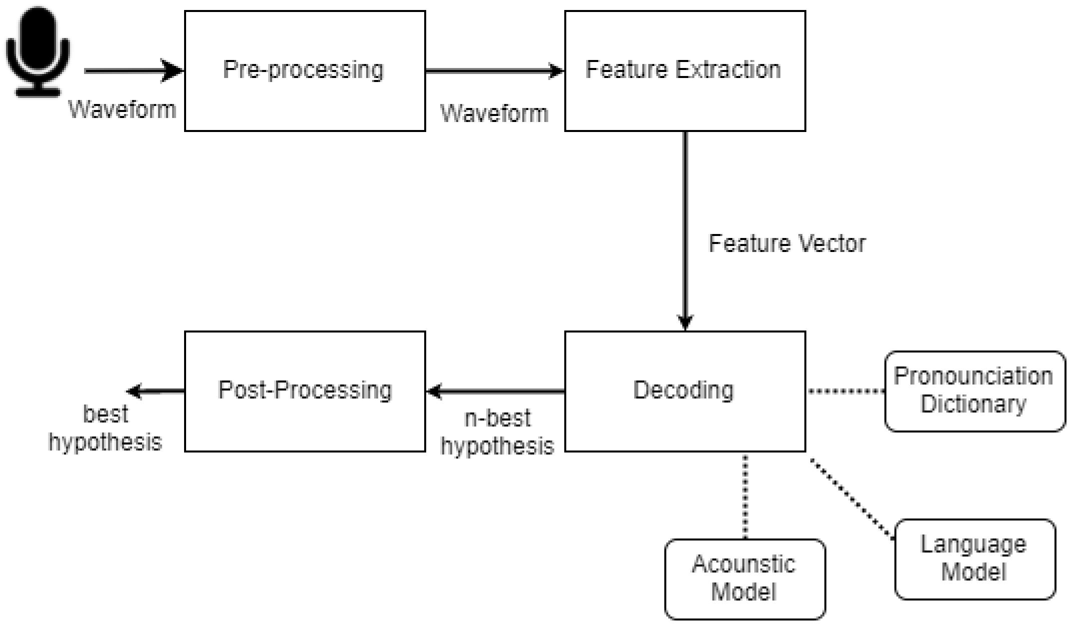


FIGURE 4: OUTLINE OF A TYPICAL ASR SYSTEM

Figure 4 shows an outline of a speech recognition system. The pre-processing block filters the incoming speech signal, reducing noise and normalising it. From this block, the features of the waveform are extracted in the feature extraction block. These features are temporal and spectral features which include energy, amplitude, pitch, spectral roll off, etc. These features are sent as a feature vector to the decoding module which will use one or more of the various methods of decoding to predict the words using acoustic and language models as well as a dictionary.

A language model (LM) estimates the probability $P(W)$ for a sequence of words W in a vocabulary V (whose size can be hundreds of thousands of words). The word sequence is broken down to individual words or smaller segments like utterances which will be referred to as w_i to have $W = \{w_1, w_2, w_3, \dots, w_n\}$. Using the assumption that the probability of a segment w_1 only depends on previous words [5]:

$$P(W) = \prod_{i=1}^{n} P(w_i|w_1, w_2, \dots, w_{i-1})$$

Perplexity (PPL) is the measure of how likely a given LM will accurately predict the test data. It is used to measure the quality of a LM using the entropy of the underlying source. PPL is given by [18]:

$$PPL(LM) = 2^{-\frac{1}{N}\sum_{i=1}^{n} log_2 p(s_i)}$$

$* \ where \ N \ is \ the \ number \ of \ words \ in \ the \ corpus.$
$* \ m \ is \ the \ number \ of \ sentences$
$* \ p(s_i) \ is \ the \ probability \ given \ to \ each \ word \ in \ the \ LM$

The smaller PPL is, the better the language model is. The size of the language model also has a significant impact on the word error rate (WER). Figure 5 shows results of an experiment carried out by YouTube where people were asked to speak a set of random Google.com queries. It can be seen that PPL and WER decrease as the size of the LM increases. Therefore, large language models are needed to create accurate speech recognition systems.

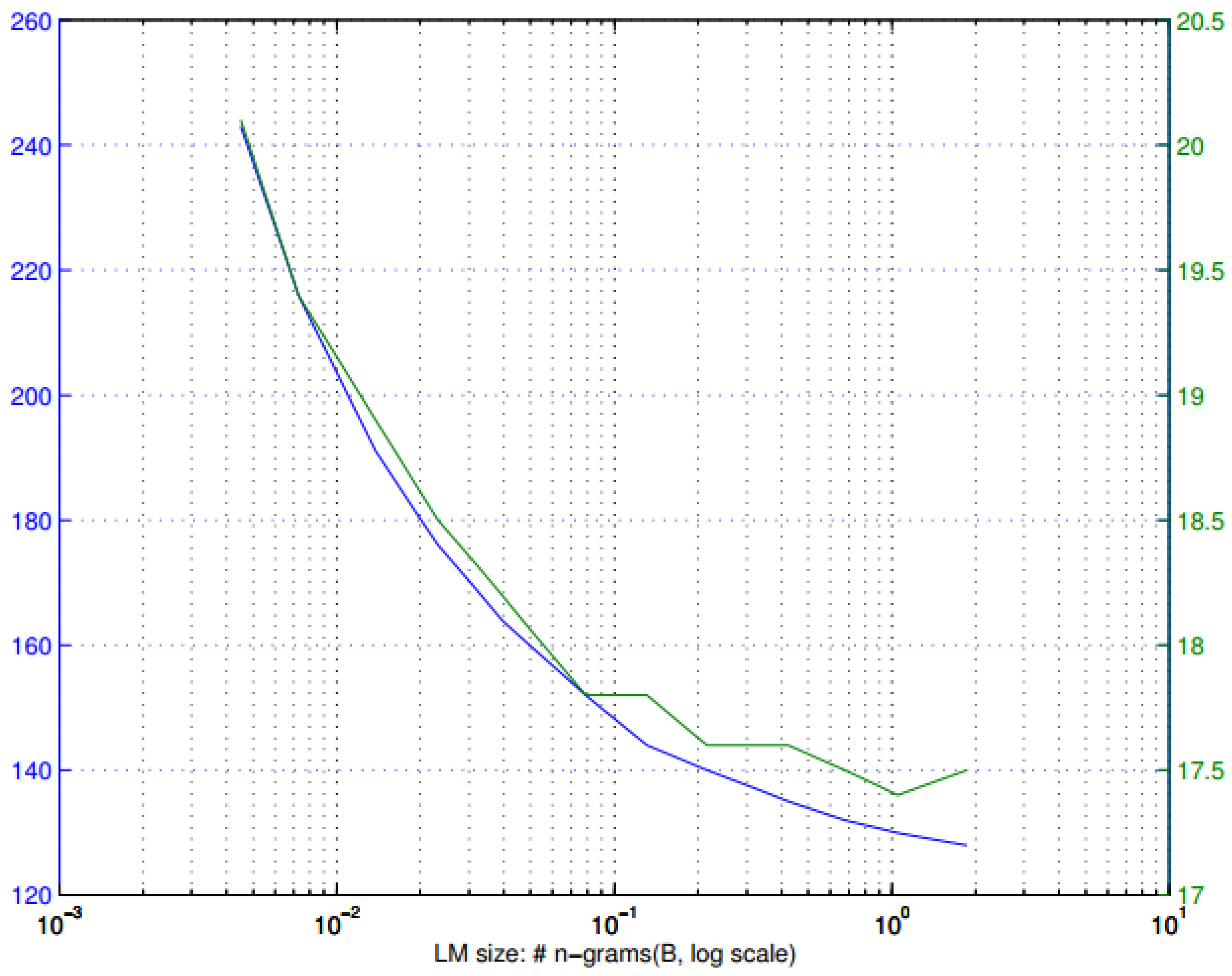


FIGURE 5: PERPLEXITY (LEFT) AND WER AS A FUNCTION OF THE SIZE OF LANGUAGE MODEL [5].

An acoustic model contains statistical representations of each of the distinct sounds (phenoms) that make up a word. It is created by taking a speech corpus and using special algorithms to create Hidden Markov Models for each phoneme. To create accurate acoustic models, a large speech corpus with a rich composition of all phenoms is needed hence the need to have an efficient way of collecting this data.

2.4. Web Applications

A web application is a piece of software that can be accessed via a browser. Web applications and web clients form a client-server environment where the "client" refers to the browser (or web client) and the "server" is the web application. Web applications have a variety of different uses including social networks, communication, data storage, media streaming just to mention a few.

2.4.1. HTTP

There are many different communication protocols used by web clients and web applications to communicate with each other, however, the most popular one is the HTTP protocol. The HTTP protocol is a request/response protocol that specifies how web client requests for data and how the server (web application) responds to requests from the clients [19]. HTTP request methods indicate the type of action to be performed by the web application. The two most common HTTP methods are GET and POST. The GET methods requests for a specific resource from the server and optional GET parameters can be used along with the request. The following is an example of a URL for a GET request to Google.co.za to return search results for "University of Cape Town":

https://www.google.co.za/search?q=university+of+cape+town

POST methods are used to submit data to the server. This data can range from simple strings like usernames and passwords, to pictures and videos. Table 1 summarises some of the HTTP request methods [19].

TABLE 1: A DESCRIPTION OF SOME OF THE HTTP REQUESTS METHODS.

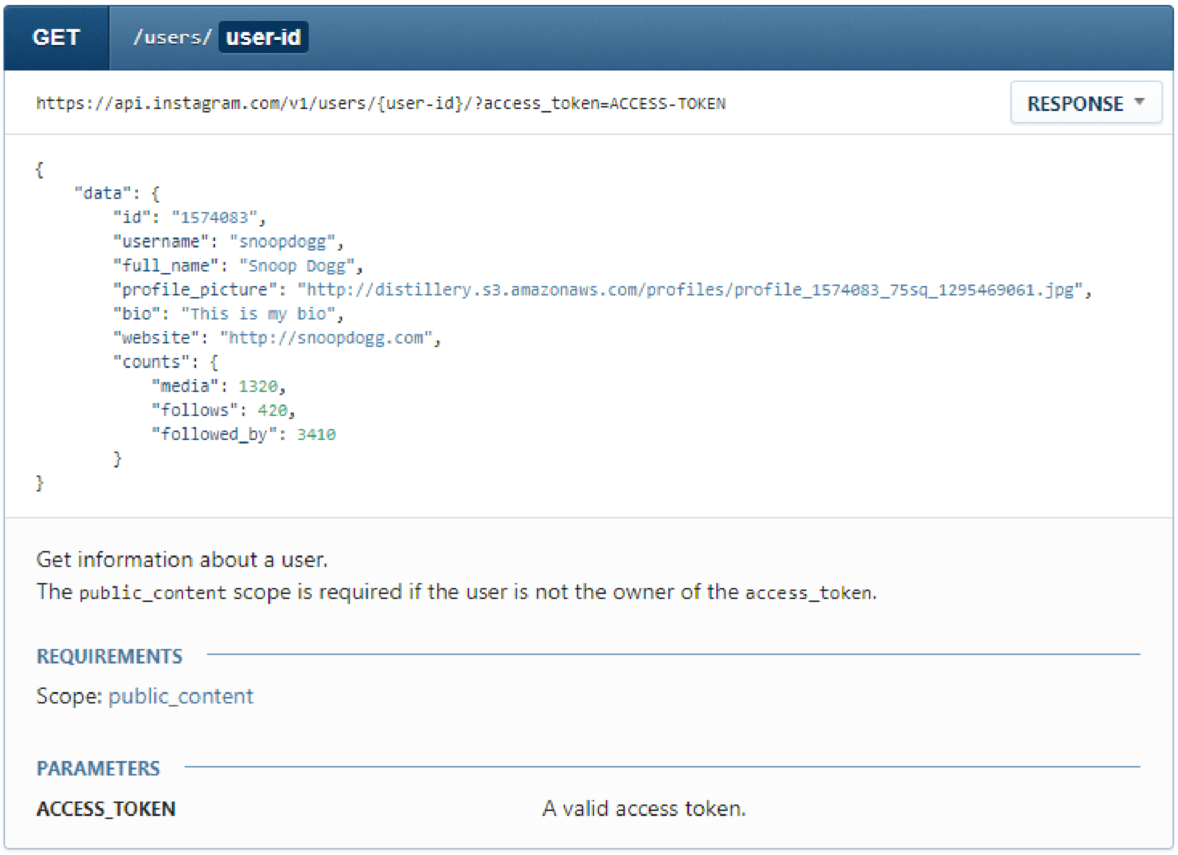
Method	Description
GET	Request for resource from server
POST	Submit data to the server
HEAD	Same as GET but does not return the body
PUT	The data within the request must be stored at the URL supplied, replacing any existing data.
DELETE	Delete a resource
OPTIONS	Return the HTTP methods supported by the server
CONNECT	Client requests the HTTP proxy to forward a TCP connection to some destination. Used to create a TCP/IP tunnel for secure connections using HTTP proxies.

2.5. Web API

A Web API is an Application Programming Interface (API) that specifies the set of rules and specifications that web applications and client-side applications must follow to communicate with each other over the HTTP Protocol. Endpoints specify where the resources lie that can be accessed by third-party software using HTTP URIs. Examples of popular Web APIs include Google Maps API which enables developers to integrate Google Maps features (search for places, navigation, traffic, etc.) into their application, and Flickr API which enables third-party developers to access Flickr services and resources (upload, browse pictures, etc).

2.5.1. Representational State Transfer (REST)

REST is an architectural concept that allows client-side applications to access and manipulate web resources on a web application using a predefined and static set of stateless operations. Web resources come in the form of images, audio, documents, etc and each resource is identified by a unique URL. Web servers that provide REST API will respond to requests in the form of JSON, XML or other predefined data format that can be parsed on the client-side application. REST can use HTTP methods to manipulate data where the GET method is used for lookups and the PUT, POST and DELETE methods are used for mutation, creation and deletion of the web resource respectively [20]. Figure 6 shows an example of an Instagram API endpoint to retrieve the profile details of a user on Instagram. The client-side application must send an HTTP GET request to the URL ***https://api.instagram.com/v1/users/{user-id}/*** where the "**user-id**" parameter will be the ID of the Instagram user in question. The Instagram API will respond with the details of the user in JSON format.


GET /users/ user-id

https://api.instagram.com/v1/users/{user-id}/?access_token=ACCESS-TOKEN

RESPONSE

```
{
    "data": {
        "id": "1574083",
        "username": "snoopdogg",
        "full_name": "Snoop Dogg",
        "profile_picture": "http://distillery.s3.amazonaws.com/profiles/profile_1574083_75sq_1295469061.jpg",
        "bio": "This is my bio",
        "website": "http://snoopdogg.com",
        "counts": {
            "media": 1320,
            "follows": 420,
            "followed_by": 3410
        }
}
```

Get information about a user.
The `public_content` scope is required if the user is not the owner of the `access_token`.

REQUIREMENTS

Scope: public_content

PARAMETERS

ACCESS_TOKEN	A valid access token.

FIGURE 6: A DESCRIPTION OF AN INSTAGRAM API ENDPOINT FOR RETRIEVING DETAILS OF A USER [21].

Advantages of using REST in the design of an API are:

- Rest provides complete separation between the client and the server and this allows for the development of a public API and also makes the API highly scalable.
- It has low resource usage since message contents can be sent as JSON or any custom datatype that uses fewer resources.
- REST operations are stateless therefore it is well suited for Create, Update and Delete (CRUD) operations.

A major drawback of using a RESTful API (an API designed using REST) is that the client-side application cannot define exactly what sort of data it needs from the server. This often leads to the application downloading a lot of extra data that it does not need just to get the parts that it needs. An example is using the Instagram API to get profile pictures of a group of users. The only way this is possible is to retrieve the profiles of all the users and then extract the URLs of the pictures from the responses which means that resources are wasted retrieving the extra data that will not be used.

2.5.2. Simple Object Access Protocol (SOAP)

SOAP is an XML-based messaging protocol that is used for exchanging information among web applications. It is platform independent and it is often used over the HTTP protocol. SOAP specifies how HTTP headers should be encoded and provides an XML file that can be used to exchange information. SOAP was designed to enable client-side applications to easily make Remote Procedure Calls (RPC) on the server despite what platform they were running on [22]. A SOAP message is an XML document that is made up of an envelope that defines the start and end of the message, a header that contains attributes of the message that are used to process the message, the body which contains the contents of the message and a fault element which has information about errors that occur while processing a message. Figure 7 shows an example of a SOAP message sent to ***http://www.xyz.org/Quotation*** as an HTTP POST request to get a quotation named Microsoft.

```
POST /Quotation HTTP/1.0
Host: www.xyz.org
Content-Type: text/xml; charset = utf-8
Content-Length: nnn

<?xml version = "1.0"?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV = "http://www.w3.org/2001/12/soap-envelope"
   SOAP-ENV:encodingStyle = "http://www.w3.org/2001/12/soap-encoding">

   <SOAP-ENV:Body xmlns:m = "http://www.xyz.org/quotations">
      <m:GetQuotation>
         <m:QuotationsName>MiscroSoft</m:QuotationsName>
      </m:GetQuotation>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

FIGURE 7: EXAMPLE OF A SOAP MESSAGE TO RETRIEVE A QUOTATION NAMED MICROSOFT [23].

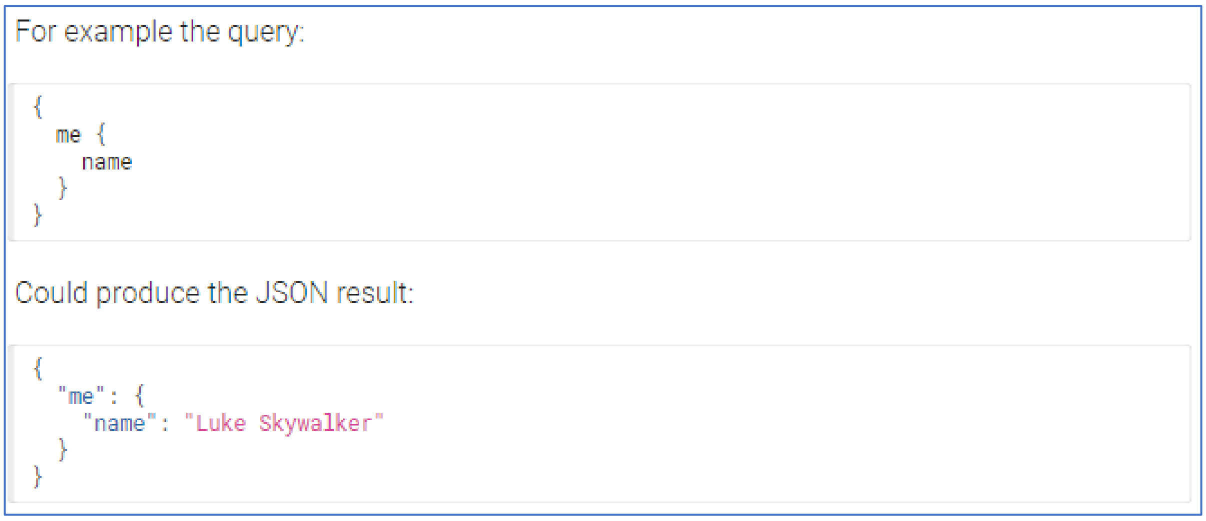
Some advantages of using SOAP include [22]:

- SOAP operations are *stateful*, which means that it can support applications that need conversational state management and contextual information, with no additional development to implement this functionality.
- If applications using SOAP must agree on the exchange format, SOAP provides specifications for the interaction.
- SOAP supports WS-Security, a security standard with enterprise security features and can be implemented together with SSL.

Some major drawbacks of using SOAP include its use of XML, a verbose messaging format which leads to slow processing and requires larger bandwidth. Many languages also do not offer support for SOAP development. SOAP services are also more difficult to develop and more so when working with more complex clients which require maintaining a *stateful* connection.

2.5.3. GraphQL

GraphQL is a query language for APIs developed by Facebook and released to the public in 2012 [24]. It was developed to address the limitations presented by REST APIs and web service architectures. Its runtime is strongly typed allowing client-side applications to define the data they need from the web API. GraphQL queries will be in a JSON-like format which specifies the data to return from the server. The response from the server will be a JSON object with results in the same structure as that specified in the query [24]. Figure 8 shows an example of a query that will return the name of an object stored on the server.


For example the query:

```
{
  me {
    name
  }
}
```

Could produce the JSON result:

```
{
  "me": {
    "name": "Luke Skywalker"
  }
}
```

FIGURE 8: AN EXAMPLE OF A GRAPHQL QUERY AND RESPONSE [25].

GraphQL is a relatively new technology and, so far, has been implemented several popular languages including Java, Python, JavaScript, Ruby, PHP, etc. The biggest advantage of GraphQL is the ability for client-side applications to describe their data and specify the data they want from the server and get predictable results. It also allows retrieval of more resources in a single request. The major disadvantage of GraphQL is that because it is still a new technology, the support community is still small, making the development of complex applications more difficult. Servers that implement GraphQL need to do more processing to verify and parse queries compared to other API implementations.

2.6. Web Application Frameworks

Web application frameworks, or web frameworks, are software framework that is designed to aid and speed up the development of web applications. They do this by simplifying and abstracting common overhead and activities associated with web development [26]. There are many different types of web frameworks which focus on either the frontend or backend part of the web application. Web application frontend refers to the components of the web application that the user sees and interact with, like the look of the web page, styling (CSS) and client-side scripting (JavaScript). Web application backend refers to the “behind the hood” operations of the web application, communication between the database and the server, processing of requests from the clients, etc. The following are the benefits of using backend web frameworks [27]:

- **Abstraction**: Web frameworks abstract the handling of HTTP requests/responses and database access. This means that the developer does not need to fully understand how these components work to develop the application.
- **Faster development**: Because the developer does not need to waste time implementing the abstracted functions mentioned above, more time can be spent developing the core features of the application.
- **Security**. Most common web frameworks have been thoroughly penetration tested and any security loopholes are quickly patched when discovered. This relieves some pressure on the developer to ensure that their application is secure and safe from attackers.
- **Scalability**: Most web frameworks are designed to make them easy to scale.

A few examples of common web frameworks were investigated.

2.6.1. Django

Django is a free and open-source (3-clause BSD license) Python, server-side web framework developed by the Django Software Foundation. It follows a "batteries included" approach where all the necessary components to develop a web application are available in the basic Django package. This allows the developer to focus their time and efforts on developing the application rather than waste time on configurations and setting up the working environment. Django follows the Model-View-Controller (MVC) architecture, a type of layered architecture explained further under 2.7.1. The following are some advantages of using Django [27].

- **Python**: Django is based on Python, a very popular and flexible programming language that is easy to learn and allows for rapid software development.
- **Support**: Due to the popularity of python and the Django framework, there is a great deal of support for these in the developer community.
- **Fully loaded**: Django comes with everything needed to set up and run a basic web application. This includes a DBMS, authentication, sitemaps, RSS feeds, a development server, etc.
- **Scalable**: Django framework is highly scalable. It supports majors web applications like Instagram [28] and Disqus [29].

2.6.2. Ruby on Rails

Ruby on Rails (sometimes called **Rails**) is a free and open-source, server-side Ruby web framework developed by David Heinemeier Hanson and first released in 2005. It follows "convention-over-configuration" (CoC), a concept introduced by David Hanson in which frameworks attempt to decrease the decisions a developer using the framework has to make without losing flexibility. CoC also means that the developer can spend more time on developing the application [30]. Some advantages of using Rails are:

- Ruby is a highly readable programming language designed for rapid application development. Most of its libraries are also open-source.
- It encourages test-driven development and has good built-in testing frameworks.
- It is easy and fast to set up a Rails project.
- The Rails framework has three different development environments; testing, development and production.

2.6.3. ASP.NET

ASP.NET is a free and open-source server-side web framework developed by Microsoft and first released in 2002. It was designed for enterprise, interactive and data-driven web-applications. ASP.NET has several different versions which follow different architectural patterns, like ASP.NET MVC which follows the MVC pattern and ASP.NET Core, which is a more modular framework, similar to the microservices architecture. Some benefits of using ASP.NET are:

- It primarily uses C#, a high performance and compiled programming language but also supports other languages namely. The most popular alternatives are Visual Basic, J# and JScript.
- It is managed and regularly updated by Microsoft.
- Ensures high reliability and security with the built-in Windows authentication system.

2.7. Application Architectural Patterns

Application Architectural patterns, also called architectural styles, characterize a family of systems that are related by shared structural and semantics. They are not to be confused with "*Design Patterns"* which are a general reusable solution for common problems; architectural patterns are applied on a high and coarse-grained level. Benefits of using architectural patterns include [31]:

- **Reuse** (both design and code): Well understood solutions are applied to new problems and certain implementations of patterns can be shared across multiple systems which use the same architectural patterns.
- **Communication**: Because architectural patterns are well researched and documented, they provide a common platform that improves communications among developers.
- **Options**: They give developers a wide range of tried and tested solutions to work with, drastically cutting down on the time they would otherwise spend trying to design an architecture.
- **Expert Knowledge**: Using them give developers access to a pool experts in those fields.

The major drawbacks of using architectural patterns is that they can be deceptively simple and implementation and integration into a software development process can be a time-consuming activity. Five architectural patterns will be investigated. These are Layered (n-tier), Event-Driven, Microkernel, Microservices and Space-Based.

2.7.1. Layered (n-tier) Architecture

This is the most common type of architecture which is used in several of the large and popular web frameworks like Java EE, Drupal and Express. Components of a layered architecture are organised in horizontal layers (or tiers) with each layer performing a specific role within the application [32]. The term “n-tier” refers to the number of layers/tiers “n” that are used in the system. In software systems, the lowest layer (or first tier) is usually the database layer which manages reading from, writing to and access to the databases. Figure 9 shows the interactions between the layers in a 4-tier architecture:

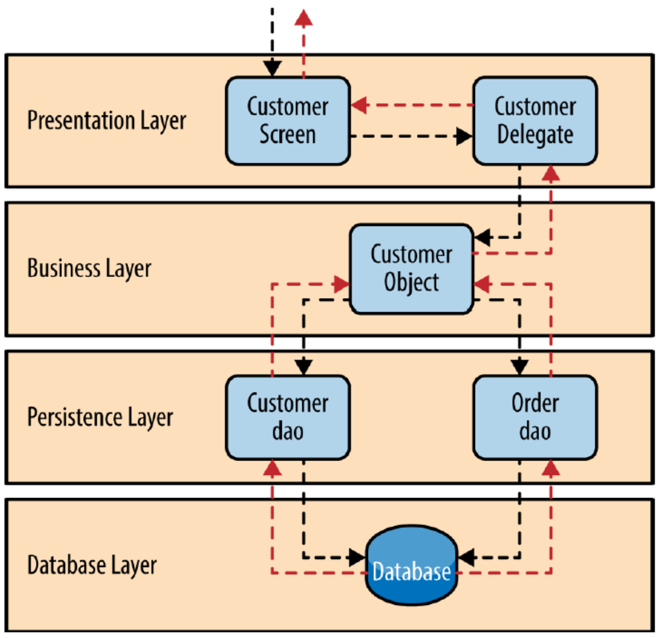


FIGURE 9: DIAGRAM SHOWING INTERACTIONS INSIDE A 4 TIER ARCHITECTURE [32]

Benefits of using the layered architecture are:

- Separation of functionality because each layer is supposed to perform a single function.
- It allows fast development of new applications.
- For use in applications which need to mirror processes (in businesses or enterprise software)
- Suitable for applications with strict maintainability and testability standards.

The drawbacks of using a layered architecture include:

- Risk of messy code. If layers do not have clearly defined functions, the source code will quickly and easily turn into a "big ball of mud", unorganised and unstructured making it hard to debug and reuse [33].
- Systems can slow down due to the "sinkhole anti-pattern" which is when most of the requests pass through the layers without being modified or processed in each layer.
- Layer isolation can also make it hard for new developers to understand the architecture without needing to understand all the modules in the architecture.
- Any new changes to the system will often require full deployment of the whole system, no matter how small the changes are.
- Scaling this architecture can be done by duplicating the system into multiple nodes but this would be expensive because of the broad granularity.

2.7.2. Event-Driven Architecture

This architecture is popular in distributed asynchronous systems and is used to produce highly scalable applications of any size. It is characterized by a central unit that accepts all data (or events) and passes it to the module that handles that data/event [32]. There are two main topologies that make up the event-driven architecture, the Mediator Topology and the Broker Topology.

2.7.2.1. Mediator Topology

This topology is suitable for systems in which the events have multiple steps which require some level of orchestration to process the event. The Figure 10 below is a depiction of the mediator topology.

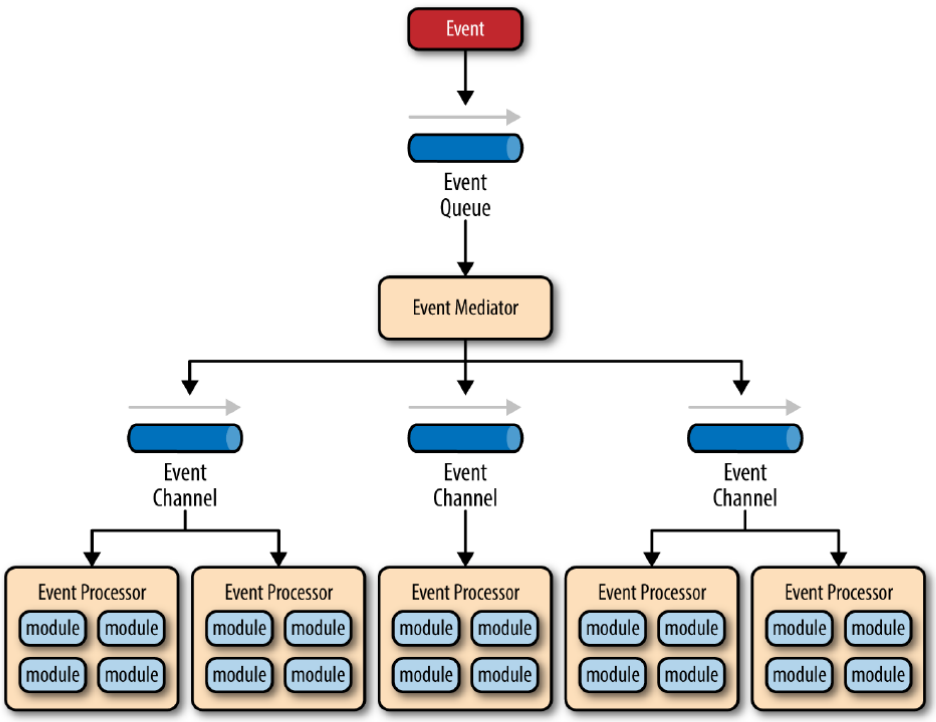


FIGURE 10: DIAGRAM SHOWING THE MEDIATOR TOPOLOGY [32]

The mediator topology works in the following way:

- The client sends an event to the **event queue** where it is queued before being passed onto the **event mediator.** Event mediators can be implemented in different ways using open-source integrations such as Spring Integration or Apache Camel.

- After receiving the event, the event mediator orchestrates the events by creating and asynchronously sending additional events to the **event channels**.
- Event processors listen for events from the event mediation via the event channel and execute each step of the event. They are self-contained, independent and highly decoupled components.

2.7.2.2. Broker Topology

The broker topology is useful for events which are simple and do not require event orchestration. Events flow in a chain-like manner through a message broker across the different event processors unlike in the mediator topology where there was one central event mediator. This is shown in Figure 11 below:

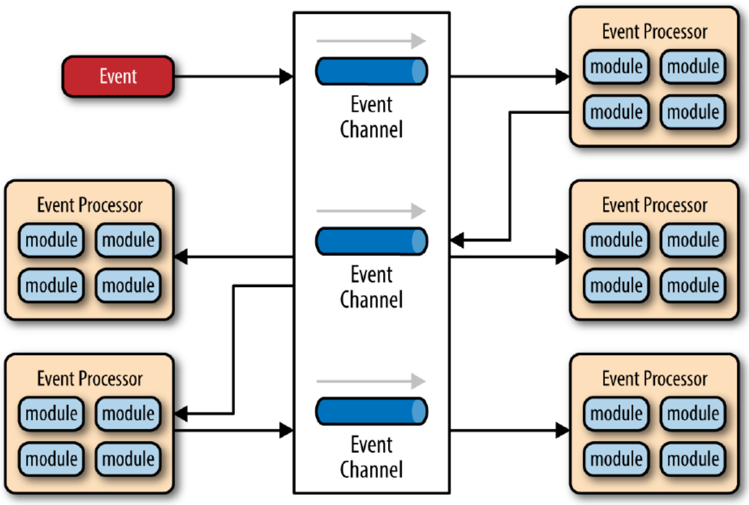


FIGURE 11: DIAGRAM DEPICTING THE BROKER TOPOLOGY OF THE EVENT-DRIVEN ARCHITECTURE [32]

Event-driven architectures are suitable for:

- User interfaces.
- Applications with modules are not dependent on very few or no other modules.
- Asynchronous systems with asynchronous data flow.

Advantages of an event-driven architecture are:

- Fine-grained scalability can be easily achieved by scaling each event processor separately since they are independent and decoupled.
- Can easily adapt to complex and changing environments.
- Can easily be extended by adding new events.

Disadvantages include:

- Testing is complex when modules affect each other because interactions between modules can only be tested in a fully functional system
- Error handling is difficult to structure for events that are processed by different modules due to the independence and decoupling of the event processors.
- Events cannot fully be processed if any one or more of the modules fail.
- The system can slow down dramatically due to messaging overhead if the event messages arrive in bursts.

2.7.3. Microkernel Architecture

This architecture is mostly used in applications that are going to be packaged and made available for download. The microkernel architecture allows developers to add plugins to support or add functionality to the core application.

The microkernel pattern has two main modules, the core system and the plug-in module. The core system will be developed separately and contain minimal functionality to make the system functional. Plug-in modules are independent standalone modules that have extra features and functions that are meant to add functionality to the core system. The relationship between the core-system and plugin modules is shown in Figure 12:

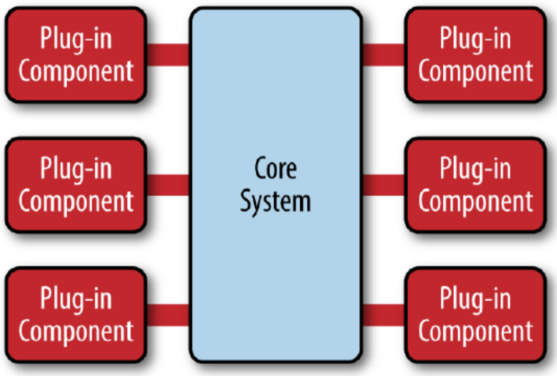


FIGURE 12: DIAGRAM SHOWING HOW THE MICROKERNEL ARCHITECTURE IS SET UP. [32]

A popular software that uses this architectural pattern is the Eclipse IDE, which allows developers to create plugins to make the IDE capable of supporting any programming language and/or platform.

Microkernel architecture is most suitable for:

- Operating systems.
- Software and tools that are available for download.
- Software and tools that are used by a wide variety of people for different applications. (different plugins can be created for each domain of usage.)
- Applications with a fixed set of functions but a dynamic set of rules for the functionality.

Advantages of the microkernel architecture include:

- Can be embedded and used as part of (or embedded in) another architectural pattern.
- Good for evolutionary design and incremental development.
- Easy to test and deploy

Disadvantages include:

- Product based and suitable for small systems, therefore, it is not scalable.
- Development process is very complex

2.7.4. Microservices Architecture

This architecture is gaining popularity as an alternative to monolithic and service-oriented systems. A microservice architecture is made up of independently deployable, small, modular services that run on the server as unique processes and communicate with each other through services like REST API or Google Protobuf [34]. Figure 13 shows a basic model of a microservices architectural pattern.

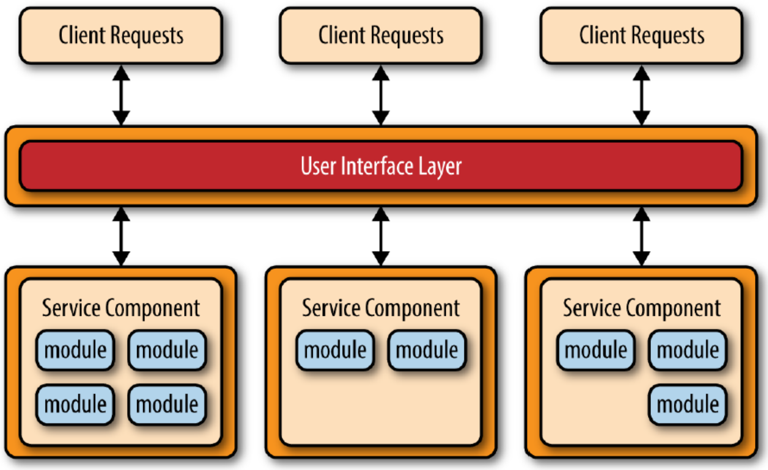


FIGURE 13: BASIC MICROSERVICES ARCHITECTURE PATTERN [32]

Each service has a certain function it performs on the overall system and can have various levels of granularity. This architecture pattern is most suitable for:

- Websites with small components.
- Data centres with well-defined boundaries.
- Development teams which are spread out in different locations.
- Rapidly developing businesses and web applications.

Advantages of a microservices architecture are:

- Highly scalable because each microservice is independently deployable.
- Very easy to deploy
- Very easy to test because of the modularity of the architecture.
- Each microservice can be developed using different languages.

Disadvantages of a microservices architecture are:

- Low performance especially with services that communicate with each other via the network.
- Some application tasks cannot be split into separate independent services.
- The server/cloud can easily become imbalanced if services are not fully dependent.

2.7.5. Space-Based (cloud) Architecture

This architectural pattern is designed to address scalability and concurrency issues faced by web applications. Because databases are hard and expensive to scale, the database layer of a web application will often not be able to keep up with the increasing load on the server despite scaling attempts on other parts of the application.

To avoid collapsing under heavy load, high scalability in space-based architecture patterns is achieved by sharing both the processing and the storage across multiple servers which is where the alternative term “cloud” architecture is derived from. There are two primary components, the *processing unit* and *virtualized middleware*. Figure 14 shows a space-based architecture.

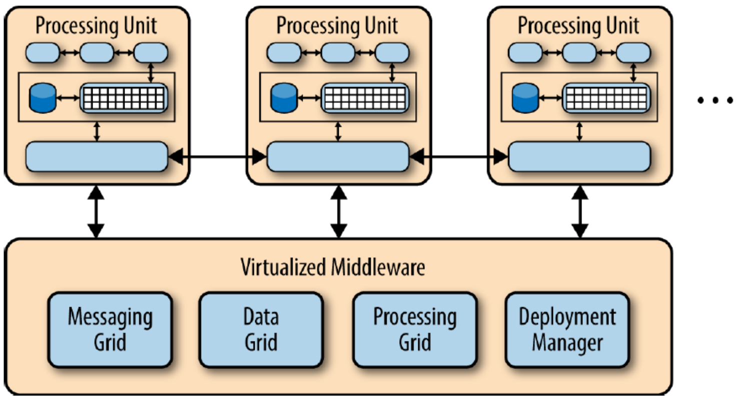


FIGURE 14: DIAGRAM SHOWING A SPACE-BASED ARCHITECTURE [32]

The processing unit contains the components of the application that performs the business logic (or bulk of the processing of data). The virtualized middleware acts as the controller in this architecture and handles components used by the processing units like handling communication between processing units, synchronisation and handling requests from clients. There are four key components of the virtualized middleware. These are:

- **Messaging grid**: Manages client sessions and input requests. It determines which processing unit should process the request and forwards the request to that unit.
- **Data Grid**: Manages data replication and synchronization between processing units which must have the exact same data inside their memory.
- **Processing Grid**: an optional component that mediates and orchestrates processing of requests when there are several processing units working on the same request.
- **Deployment Manager**: This is the unit that achieves the scalability of the architecture. It continuously monitors the load on the server and dynamically manages the shutdown and start-up of processing units depending on the load.

The space-based architectural pattern is suitable for:

- Web services with large load variations (like social networks)
- Services with data that can occasionally be lost without severe consequences.

Advantages

- Highly responsive and scalable architecture
- Easy to deploy.
- High performance due to its cloud-based processing components.

Disadvantages

- Very hard to develop due to the sophisticated data caching.
- Hard to create a test environment with high user loads.

2.8. Database Management System (DBMS)

A **database** is an organised collection of related data and a **database management system** is a server that manages access and storage of data stored in a database. Web applications use DBMS for several uses including user account data, passwords, session tokens, etc. A DBMS needs to be able to efficiently search, store, delete and organise data inside databases and these requirements have led to a variety of types of database [35].

2.8.1. ACID

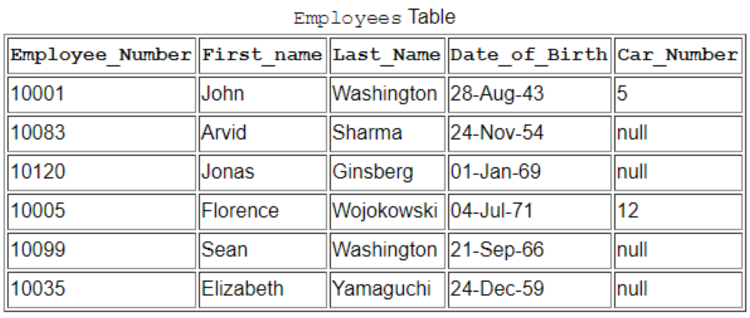
ACID is an acronym for Atomicity, Consistency, Isolation, Durability. It is a set of properties that guarantee validity of data inside a database after each transaction (a query to a database) in the case of any errors or system malfunctions. The four characteristics can be briefly described as follows [36]:

- **Atomicity**: Each database transaction should not change the state of a database if any part of the transaction fails.
- **Consistency**: Each transaction should leave the database in a valid state.
- **Isolation**: Concurrent transactions should produce the same results if they were to be executed sequentially.
- **Durability**: Any committed transaction should remain committed even in the event of a system crash or power failure.

A DBMS that guarantees ACID properties is regarded as reliable because it maintains the integrity of applications. However, adherence to ACID properties can have a negative effect on performance since there must be ACID checks after each database transaction.

2.8.2. Relational Model Databases

These are the most popular type of databases in web applications. A relational database is a database that stores data in tables with rows and columns. A table in a relational database can also be called a "relation", referring to the way a collection of similar data is stored as rows as shown in Figure 15 .



Employees Table

Employee_Number	First_name	Last_Name	Date_of_Birth	Car_Number
10001	John	Washington	28-Aug-43	5
10083	Arvid	Sharma	24-Nov-54	null
10120	Jonas	Ginsberg	01-Jan-69	null
10005	Florence	Wojokowski	04-Jul-71	12
10099	Sean	Washington	21-Sep-66	null
10035	Elizabeth	Yamaguchi	24-Dec-59	null

FIGURE 15: AN EXAMPLE OF A TABLE (RELATION) STORING DATA ABOUT EMPLOYEES IN A COMPANY. [35]

A DBMS for relational databases is simply called a Relational Database Base Management System (RDBMS) [37]. Structured Query Language (SQL) is a standard query language that is used to define and manipulate data in databases. There have been many SQL DBMSs over the years, each with their advantages and disadvantages. A few of the popular ones are described below.

2.8.2.1. MySQL

MySQL is an open-source, ACID compliant [38] RDBMS originally developed by MySQL AB which is now owned by Oracle Corporation. Because of its popularity, MySQL has become an industry standard and thus is supported in almost all major software systems from operating systems to web frameworks. It also provides error messages in multiple languages besides just English. However, since it was acquired by Oracle in April 2009, the developer community has since been worried that MySQL no longer falls under Free and open-source since members of the public can no longer fix bugs and create patches [39]. Besides these worries, MySQL still offers a lot of advantages over other DBMSs, these include [40]:

- Can be used when there is no network available
- The library can be used for standalone applications to have an embedded database.
- Has a separate program for server and client networked environment.
- Has a flexible privilege and password system and all password traffic is encrypted.
- Can support very large databases that contain more than 50 Million records and can support up to 64 indexes per table.

2.8.2.2. SQLite

SQLite is an open-source self-contained, serverless and ACID compliant RDBMS developed by The SQLite Consortium. Unlike most DBMSs, SQLite does not operate like a client-server system, instead, it is embedded within the program which will be using it. Any program that uses SQLite links the SQLite libraries and SQL queries to the database are made via function calls which greatly reduces latency as compared to conventional RDMSs in which communication to the database would have to be between processes possibly on different machines. SQLite requires no setup and configuration and stores its data inside a cross-platform single file which makes it very portable. Because of its lightweight architecture, SQLite is good for the following uses [41]:

- Early stages of software development where access to a fully functional database is needed but not yet set up and configured.
- Internal databases. Used for server-side only or client-side only databases to store persistent data related to that application.
- Web applications with low to medium traffic (fewer than 100,000 database hits per day)

2.8.2.3. PostgreSQL

PostgreSQL is an open-source, ACID compliant RDBMS developed by PostgreSQL Global Development Group. It is designed to securely store data and manage systems of any size with multiple concurrent users. Postgres uses a multiple row data storage called Multi-Version Concurrency Control (MVCC) to achieve concurrency in high volume environments. MVCC uses the first-committer-wins-rule which works by taking a snapshot of the current data before access to the data and compares it to the original data before updates to the data are saved [42]. Advantages of using PostgreSQL include [43]:

- Designed to support high volume concurrent data streams
- It's cross-platform
- Fully open-source with good third-party support.

2.8.2.4. VoltDB

VoltDB is an open-source in-memory RDBMS based on NewSQL. NewSQL is a new class of DBMS that delivers the same performance as NoSQL DBMS (described later in this chapter) while retaining support for queries and ACID to improve performance for heavy workloads [44]. In-memory means that data queries all happen in memory rather than on permanent storage devices like hard disk and solid-state drives. This leads to extremely fast performance compared to the more traditional RDBMS. The durability property of ACID is achieved by taking continuous snapshots and asynchronous command logging for crash recovery. Advantages of using VoltDB include [45]:

- Cheaper and easier horizontal scaling.
- Shorter transaction latencies and higher overall performance compared to other RDBMSs.

- No multiprocessor overhead
- Optimised for complex operational workloads.

2.8.3. NoSQL

NoSQL (Not Only SQL) is a term to refer to DBMS which do not use the relational model. NoSQL was developed to address the limitations faced by relational databases. These limitations are [46]:

- **Concurrency**: As the size of relational databases increase, their complexity also increases and these lead to concurrency problems such as deadlocks, causing the read/write speeds to be greatly reduced.
- **Big Data:** Large amounts of data are difficult to process using relational databases.
- **Scalability and Availability:** Currently, the best way to scale relational databases is to use vertical scaling, by more memory and/or CPU modules on a single node, which is costly and limited by the capacity of the machine(s) on that node.

These limitations are addresses by NoSQL, which have the following advantages over relational databases:

- **No Schema**: NoSQL databases use storage methods that do not require fixed table structures (XML, Maps, etc.) or prior establishments. They can also store custom data types at any time.
- **Horizontally Scalable**: NoSQL databases can be scaled horizontally by adding more hardware/software entities so that they work together as a single logical unit i.e. adding more processing nodes to work together as opposed to vertical scaling where you add more resources to a single node.
- **Lower Cost**: NoSQL databases can run on multiple low-cost nodes which is cheaper than running relational databases which traditionally must be on a single node.

However, NoSQL comes with their own disadvantages which include:

- **Complexity**: NoSQL databases do not have their own query language (like SQL) which makes complex databases queries difficult to program especially for large databases.
- **Reliability**: Unlike most relational databases, NoSQL DBMSs do not support ACID properties which means that more (complex) programming is needed to achieve ACID properties.
- **Consistency**: Because NoSQL does not support ACID properties without extra programming, there is no consistency across the different NoSQL DBMSs.
- **Bleeding Edge Technology**: Most businesses either have not heard of NoSQL or are not interested in implementing it and replacing their relational databases with NoSQL. Because of this, there is a smaller developer community for programming and support when developing systems that use NoSQL.

Some of the popular NoSQL DBMSs are briefly investigated below.

2.8.3.1. MongoDB

MongoDB is an open-source NoSQL DBMS that is developed by MongoDB Inc. It stores data in documents, which are key-value pairs structured in a JSON format. A set of related or similar documents is called a collection. However, documents within the same collection have a dynamic schema, which means that they can have different fields within them [47]. Table 2 shows the relationships of terminologies between the relational model and MongoDB.

Table 2: Table showing the relationships of terminologies between relational model and MongoDB

MongoDB	Relational Model
Document	Row
Collection	Table
Field	Column

Advantages of using MongoDB are:

1. **Document Object Oriented**. Data is stored in the form of JSON documents which offers high flexibility. Documents can do most things you can do with RDBMS, because of how similar they are, and more thanks to the dynamic schemas.
2. **Replacement for RDBMS**: Can be used to replace an RDBMS by just converting RDBMS schemas to documents and collections.
3. **High performance, scaling and flexibility**: Because MongoDB is a NoSQL DBMS, all the features offered by NoSQL discussed above in 2.8.3.

2.8.3.2. Redis

Redis is an open-source, in-memory, NoSQL DBMS developed by Salvatore Sanfilippo primarily for POSIX systems. It uses key-value store with optional durability (the "Durability" property of ACID) and is sometimes referred to as a data structure store because keys can contain different data types (sets, hashes, lists, etc.). Queries to the Redis database can also be atomic, despite the type of query or data in question (list, string, map, etc.). Because it is an in-memory database, persistence can be achieved by dumping the database onto the storage drive or by logging commands to a log onto the disk [48].

Advantages of using Redis include:

- All operations are atomic.
- Can support different data types
- It can also be used as a cache or a message broker thanks to its ability to support different rich data types.
- Because it can be set to not be persistent, Redis can be very fast with the ability to perform 110,000 SET requests per second [49].

2.8.3.3. Apache Cassandra

Apache Cassandra, commonly referred to as just Cassandra, is an open-source NoSQL DBMS developed by Apache Software Foundation. It supports replication across multiple data centres and distributed systems where all nodes will be decentralized and capable of processing any request [50].

Unlike other NoSQL databases which use some form of key-value pairs to store data, Cassandra uses a combination of key-value pairs and tables. In Cassandra, tables are part of the keyspace container and tables have partitions with unique partition keys and each partition contains rows with a similar structure, which may optionally have unique clustering keys [50].

Advantages of using Apache Cassandra are:

- Designed for distributed systems.
- It has a rigid architecture with no single point of failure. Cassandra does not have a master node when it is being used in a distributed system or data centre eliminating the single point of failure at the master node. All nodes can process any request.
- Has high horizontal scalability.
- Can support rich data types like lists, sets, etc.

2.9. File Storage

Effective storage is needed for the audio and video data that will be collected. Developers have come up with many ways of storing large files on web servers which will be investigated below.

2.9.1. Filesystem Storage

Filesystem storage involves storing data in server's filesystem. A filesystem is the method and data-structures used by an operating system to store, catalogue and retrieve files. Advantages of using the filesystem to store media files are:

- Coding is much simpler when storing files in the filesystem because of the familiar interface and the large support from the programming community.
- Migrating the data will be very easy as it will involve copying from one location and pasting in another location while having the freedom to dictate read and write permissions on individual files. Migrating to CDNs or more secure and streamlines systems like Amazon S3 will be easier in the future.
- File storage space is usually the cheapest option when it comes to data storage.

The biggest problem with using a filesystem is that the operations will not follow the ACID properties. When using a relational database, relational mappings to the files will not be guaranteed.

2.9.2. Database storage

This would mean storing uploaded media inside the database along with the rest of the data will be stored for the web application. The advantages of this approach are:

- It is more secure than storing on the server's filesystem.
- Database backups will also include the media files.
- An ACID database will include a rollback of an update, something that would be complicated if the files are stored outside the database
- Uploaded files will be well synchronised with the rest of the contents of the database. This will make it much easier to track database transactions.

The size of the database is the root of most problems associated with storing files in a database. The size of the database would be several times larger than it would have been if it was storing references to the files instead. Because relational databases cache some, or sometimes, ALL the data to the server's memory before performing transactions [51], storing files inside the database will lead to ineffective memory usage and therefore slow database transactions. As databases get larger, backups will also become less frequent if storage space is expensive.

Besides the disadvantages stated above, storing files in a database can be a good option if there is only a small number of files stored in the database.

2.9.3. Block Storage

Block storage is a type of data storage system that stores data in fixed-size volumes called blocks which are often made up of a single drive or partition on a drive. The blocks are controlled by the operating system and communicate via Ethernet using Fibre Channel over Ethernet or iSCSI protocols. Block storage offer blocks of any size which could be treated like a normal disk (can format any filesystem on it and use it like a normal hard drive). They can be attached to a virtual machine and used for database, file-systems and RAID. Some advantages of block storage include:

- Blocks can be resized at any point making them easy to scale.

- They provide a familiar programming interface as they can be used the same way a filesystem can be used.
- Blocks can be easily attached/detached and moved between virtual devices.
- Block storage is well supported on many platforms and programming languages.

Some disadvantages of using block storage are that blocks are tied to one server at a time therefore in the case of a system failure, data stored on that node can be completely lost. Blocks face the same problems faced by filesystems in the sense that they can only store very little metadata about the files they store. Any additional information will have to be managed by a database, which adds complexity to designing and maintaining such a system.

2.9.4. Object Storage

Also referred to as *object-based storage*, object storage is a type of data storage system that stores and manages data as objects instead of storing them in a file hierarchy as in a filesystem or object storage. Each object will be made up of the actual data, metadata and a global unique identifier [52]. The metadata contains any information about the data with no restrictions about what metadata is possible. The global unique identifier is an address given to the object for it to be located in a distributed system. Advantages of object storage include [52]:

- Highly horizontally scalable architecture. The object storage system can be scaled by both by adding additional nodes and adding resources to each node.
- Object storage uses HTTP to access it, therefore, can be used by any platform or programming language.
- Customisable metadata. The metadata for objects does not need to be the same and can be customised to the needs of any application. It can include security details about the data, the owner, properties of the file.
- High reliability and availability. Objects are replicated at each node in the object storage system, therefore, there are several copies of the same files are available at any point and if one node fails, the system can still operate normally with the remaining nodes.

The disadvantages that come with using object storage are that using it is not as intuitive as using traditional databases, filesystems or block storage since communication is only via a communication protocol like HTTP. Any object that must be modified must be completely rewritten (you can't modify it or make incremental changes) [52].

TABLE 3: TABLE COMPARING THE DIFFERENT FILE STORAGE SYSTEMS

Storage System	Ease of use	Development complexity	Reliability	ACID properties	Scalability
Filesystem	High	Low	Low	Low	Low
Database	Medium	Medium	High	High	Medium
Object	Low	High	High	Low	High
Block	Medium	Medium	Medium	Low	High

2.10. Load Balancing

Cloud computing is the delivery of computing services over the internet. These services include computing servers, storage and database management. One of the major advantages of cloud computing is they can be used to offer infrastructure as a service. This can be used to build distributed systems where a system can have several processing nodes instead of one large server. Just like inside a single server where the operating system manages the distribution of the workload among the cores of the CPU, a distributed system needs a way to manage and distribute the workload among the processing nodes. This work is done by a load balancer. A load balancer is a server that manages the distribution of the workload in a distributed system across the processing nodes. There are different methods (or algorithms) of load balancing. The following are some examples of load balancing algorithms:

- **Round Robin**: This algorithm forward requests equally and sequentially to the backend application nodes. Each server can be weighted such that it receives more requests than others. This algorithm is simple to implement and is suitable for loads which have requests that are similar in complexities. It does not perform very well with loads that have requests with varying complexities.
- **Least Number of Connections**: The load balancer keeps track of the number of connections each application node has and forwards requests to the node with the least number of active connections. This algorithm is more complex than round robin and will require more resources.
- **Shortest response time**: The load balancer keeps track of the response times of the nodes, that is, the Time to First Byte (time between forwarding the request and receiving the first byte from the server). Requests are forwarded to the server with the least number of active connections and the low average response time. This algorithm is effective in systems with nodes located in the same geographical area (thus low and equal latencies)
- **IP Hash**: The IP Address of the incoming request determines which server it will be processed on. This algorithm is suitable for services which require users to be connected to the same server for all their sessions.

2.11. Software Development Process

Scrum is a process framework managing software development. Scrum is designed as an alternative to the waterfall model to increase productivity and efficiency in the development of software that has requirements that may change over time [53]. Scrum is designed for small teams which break down the work to be done into the “actions” that will be completed in short cycles, often 2 weeks long. On a daily basis, the team will have stand-up meetings, short meetings where each team member gives a brief summary of what they did the previous day and what they plan on achieving on that day. This keeps everyone in the team updated and on the same level of understanding in the project [53].

Because this project is carried out by one person with limited time and resources, the software development process for LARMAS will be different from the one described above although it will borrow many concepts from it. The first half of the project involves literature review and design of the system. During this period, there will be weekly meetings a week with the project supervisor Dr Tšoeu for updates and feedback on the project’s progress, and a few meetings with other students working on related projects, Nikar Ramchunder and Zabelo Mabuza who will be developing a client-side mobile applications that will use LARMAS for their backend services (to store the data they collect).

The second half of the project will be for implementing a prototype for LARMAS. The work will be divided into “actions” similar to the scrum process. However, these actions will have flexible 3 to 6-

day cycles instead of the 2-week cycles in scrum and some will overlap with each other. These actions will be different features that are needed in LARMAS, starting with the most important core features to the less critical features.

2.11.1. Test Driven Development

Development of LARMAS will be test driven. Test-driven development is a software development practice in which a developer writes very specific tests for the features they are about to implement based on the requirements before implementing the features. The developer then produces the code to implement the features until all the tests pass. Tests can be created using testing frameworks that may be independent or be part of the programming language or framework. Examples of testing frameworks include JUnit for Java and PyTest for Python. Test-driven development is iterative and the cycle is summarised below [54].

1. Add new tests
2. Run tests to verify that they fail.
3. Write code to implement the feature or fix bugs.
4. Run tests.
    - If test fails, go back to step 3
    - If tests pass, continue to step 5
5. Refactor the code. This step cleans up the code, moving it to relevant packages, removing any duplicates, renaming variables and classes, adding comments and enforcing any programming paradigms (like Object-oriented programming for example).
6. Run tests.
    - If tests fail, go back to step 3.
    - If tests pass, the cycle can be repeated for a new feature. (back to step 1)

2.11.2. Version control

Also known as revision control or source control, version control is the management of changes to source code over the period of the development. A version control system (VCS) is a software tool used for version control. It manages and keeps track of all changes made to the source code [55]. When a developer makes additions of changes to source code, he/she adds (*commits*) the code to the VCS repository. More advanced VCS allows multiple developers to work on the same code base in a *distributed version control* and will attempt to merge the changes from all commits from the different developers.

In version control, a branch is a copy of the main repository that is used to make changes to the source code without changing the source code in the main repository. Branches are used to implement and test new features and run experiments. When changes made in the branches are approved, they can be merged to the main repository. The main branch of a VCS is called the master branch. The branch on which the latest developments are made is called the *develop* branch and the branches on which features are implemented are called *feature* branches. When features are fully implemented and tested, their branches are merged into the develop branch and the feature branch is closed off. Major releases are added to the *master* branch [56].

2.11.3. Continuous Integration

Continuous integration (CI) will also be used in the development of LARMAS. CI is a software development practice where a developer commits merges their additions and/or changes into the main repository several times a day to avoid integration problems [57]. Each time these changes to the main repository are made, tests are run to make sure that the changes to not break the system. If the tests fail, the developer will be notified immediately with information on which tests failed and which commit caused the tests to fail.

2.11.4. Summary of Software Development Approach in LARMAS

The first stage of the implementation stage of the project in LARMAS will be setting up of the software development environment. This is where the chosen operating system will be installed and all the necessary development and testing tools and frameworks will be installed and configured.

After setting up the environment, the work that needs to be done and features to be implemented will be broken down into actions. Each action during the development of LARMAS will have the following steps:

1. Create a new branch in the version control with a name of the feature to be implemented to work to be done.
2. Write integration tests and unit tests.
3. Write code using the methodology described in section 2.11.1 committing changes as each test passes (practising CI).
4. After all tests pass and code has been refactored, commit and merge changes with the *develop* branch in the main repository.
5. If integration tests fail, debug the code until the tests pass.
6. If all tests (unit tests and integration tests) pass, merge the *feature* with the *develop* branch.

When all the actions have been completed, experiments will be designed to test the performance of LARMAS in a production environment and test and compare different configurations. After the experiments, there will be discussions on the results of the experiments and conclusions of the project will be made. Figure 16 is a Gantt chart of the actions and activities during the software development phase of the project.

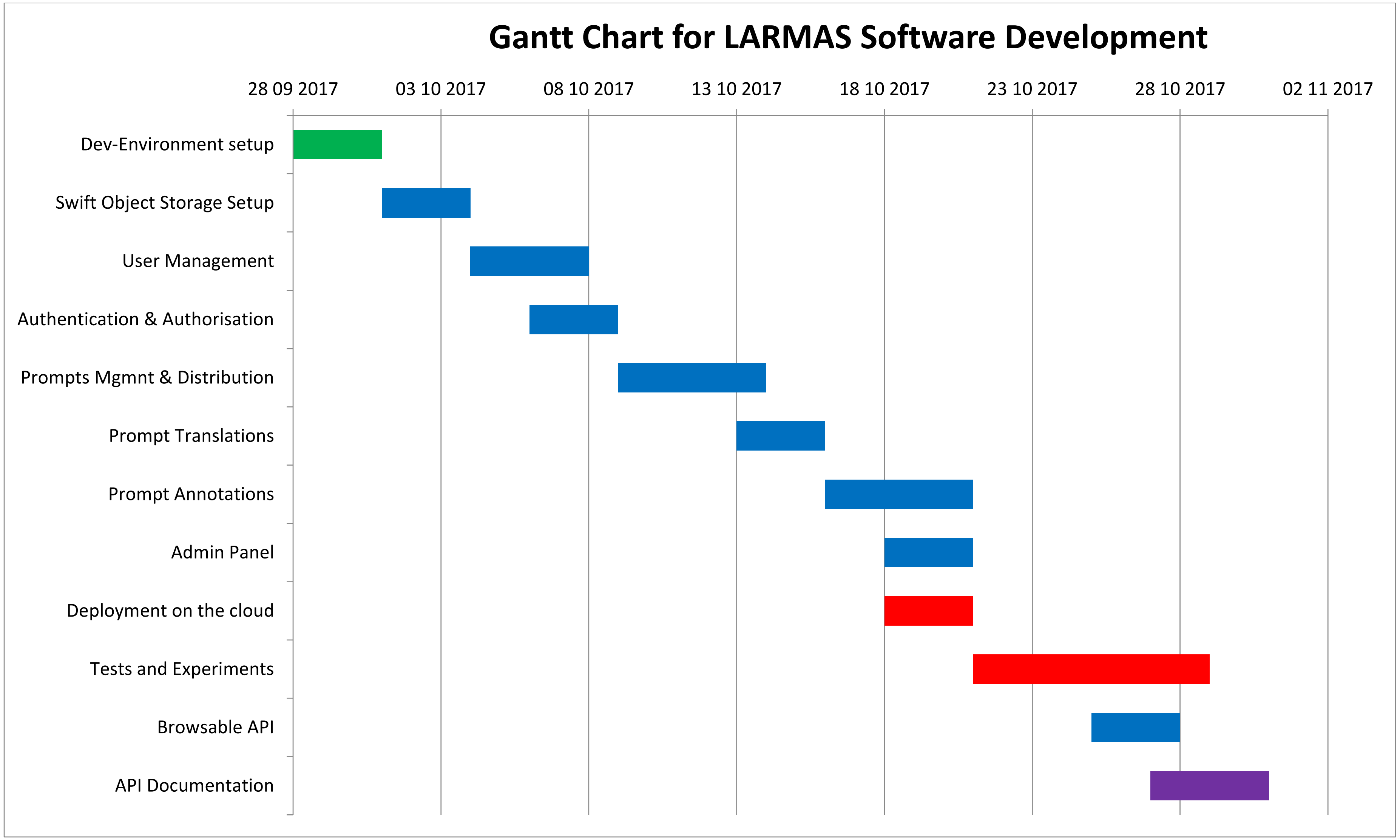


FIGURE 16: GANTT CHART OF THE SOFTWARE DEVELOPMENT OF LARMAS

3. Research Design

3.1. Abstract

This chapter focuses on the design and implementation of LARMAS. It starts off by defining the functional and non-functional requirements of the system. This is followed by a section describing the development environment, where all the tools used to develop LARMAS are summarised and briefly explaining how they were used. The chapter then goes on to explain how the database, architecture, API and storage systems were designed to aid the management and distribution of prompts and the collection of annotations and translations. The last section explains the server architecture options for deploying LARMAS and how LARMAS should be configured and deployed on the cloud.

3.2. System Requirements

The system requirements for LARMAS will be split into functional and non-functional requirements. Functional requirements describe what the system should do. Non-Functional Requirements (NFR) look at the criteria used to judge the performance of the system. [58]

3.2.1. Functional Requirements

a) **Users**: Contributors to the project must be able to register as users to capture important data like languages spoken, dialect, age, etc. Collected data will be mapped to the users that contributed the data.
b) **User Management**: Uses must be able to add and/or update their details (names, contact details, languages, etc.)
c) **Prompt Generation**: Users with administrative privileges must be able to add prompts for any of the supported languages to the LARMAS database.
d) **Prompt Distribution**: The system must evenly distribute prompts to registered users to keep a balanced corpus.
e) **Prompt Rejection**: In case users come across a phrase they are not able to say, or if they do not want to contribute to any specific phrase for any reason, they must be able to reject any of the prompts they received and request for a new one.
f) **Annotation Collection**: Contributing users must be able to record prompts, annotate the recording and upload it to LARMAS.
g) **Annotation Management**: Administrative users must have access to all the collected annotations along with the raw data. They must also be able to process the raw data further (doing manual validation checks).
h) **Translation Collection**: Users must be able to translate prompts and upload the translation to LARMAS for storage.
i) **Parallel text:** An administrator must be able to download parallel text from the contributions made by other users.

3.2.2. Non-Functional Requirements

3.2.2.1. Response Time

The response time is the time it takes for a server to process a request from the moment it receives the request to the moment it returns a response. The general advice on response times according to Jakon Nielsen are [59]:

- **0.1 seconds** is the maximum response time which causes the user to feel like the system is responding *instantaneously*.
- **1 second** is the maximum response time for which the user's concentration and flow of thought will stay uninterrupted.
- **10 seconds** is the maximum response time at which user's attention will stay focused on the dialogue. If response times are longer, users will want to preoccupy themselves with other tasks while they wait for a response from the server.

LARMAS API endpoints perform a wide variety of operations, some requiring multiple databases queries while some may only perform one database query or none at all. The performance of the database will largely depend on the DBMS that is used and the infrastructure and resources supplied. Experiments by Google have shown that increasing their web search latency from 100-400ms reduces the number of searches by a user by 0.2 to 0.6%. They also found that the total number of searches by users decreases as the latency increases and for longer response times, the number of searches by a user does not return to normal even after the response times go back to their previous values [60]. Table 4 shows results from the experiments conducted by Google:

TABLE 4: EXPERIMENT IMPACT ON DAILY SEARCHES PER USER [60]

Type of Delay	Magnitude (ms)	Duration (weeks)	Impact (%)
Pre-Header	50	4	N/A
Pre-Header	100	4	-0.20
Post-Header	200	6	-0.29
Post-Header	300	6	-0.59
Post-Ads	400	4	-0.30

Google recommends that server response times be under 200ms, excluding the network latency [61]. The requirement for LARMAS, therefore, is that 95% of all requests to LARMAS must have a response time of less than 200ms for the beta version and final release candidates. This response time is 20% that of the maximum response time for which a user's concentration and flow of thought will stay uninterrupted. This means that the client-side application will have 800ms to process the response, including the network latency.

3.2.2.2. Workload Modelling

The workload is the distribution of the load across all usage scenarios of a web application. A workload model describes how an application will be used in the production environment. An application must be tested against a workload model that is similar to the environment the application will be in during production. A good workload model will aid in the preparation of test data and lead to more accurate and reliable performance test results.

TYPES OF USERS

TABLE 5: THE DIFFERENT TYPES OF LARMAS USERS.

Group Identifier	Description
A	Dedicated contributors who will be continuously contributing to LARMAS on a regular basis.
B	One time contributors in the case where a researcher will go around asking people to contribute via their phone, desktop application or web application.

C	Administrative users who will oversee verification of data submitted by contributors.
D	Administrative users who will be updating or adding data to the system (prompts, languages, profiles, etc.)
E	NLP researchers who will be pulling data from LARMAS to use for their own research (ASR, MT, etc.)

KEY SCENARIO IDENTIFICATION

Key scenarios are the scenarios with actions which will have the greatest effect on the performance of LARMAS. These include the most frequently visited, time-dependent, critical and resource intensive scenarios. Table 6 summaries the key scenarios in LARMAS:

NB: Column Heading Decryptions of Table 6

- **Actions**: The actions that will be taken during the scenario.
- **Input Data**: Data that will be provided to LARMAS to carry out the scenario.
- **Output Data**: Data that will be provided to the user during the scenario.
- **Think Time**: The time it takes for the whole scenario to play out.

Scenario	User Group	Description	Actions	Input Data	Output Data	Think Time (s)
S1	A, B	Record and annotate a prompt.	▪ Login ▪ Request for prompt(s) ▪ Upload the recording and annotation.	▪ Authentication details ▪ Audio Recording ▪ Annotations	N/A	60
S2	A, B	Translate a prompt.	▪ Login ▪ Request for prompt(s) ▪ Upload translation	▪ Authentication details ▪ Translation	N/A	40
S3	C, D	Add a new prompt to the LARMAS.	▪ Login ▪ Create new prompt	▪ Authentication details ▪ Prompt details	N/A	10
S4	C, E	Retrieve parallel text	▪ Login ▪ Get parallel text	▪ Authentication details ▪ Language code	▪ Parallel Text	10
S5	C, E	Retrieve annotations and recordings.	▪ Login ▪ Get annotations and recordings	▪ Username and Password ▪ Or Auth Token ▪ Translation	▪ Annotations ▪ Recordings	10

TABLE 6: SUMMARY OF THE KEY SCENARIOS FOR LARMAS

RELATIVE LOAD DISTRIBUTION

Some scenarios will be executed more frequently than others and the relative load distribution shows the percentage that each scenario's load to the total load of the system. The load is determined not only by the number of requests that scenario uses but by the processing that is done for each of the requests. Accurate figures for the distribution of the load can only be determined after a beta version is released. However, for testing purposes, predictions will be made based on talks with the project supervisor Dr Tšoeu where it was concluded that the project will focus on data collecting for the first 3 to 5 years of usage and then only then can it have enough data for NLP researchers to start pulling data from LARMAS. The relative load distribution is shown in Table 7.

TABLE 7: THE RELATIVE LOAD DISTRIBUTION FOR EACH KEY SCENARIO

Scenario	Percent of Total Load Distribution	
	First 5 years	After 5 years
S1	35	10
S2	30	10
S3	15	5
S4	10	40
S5	10	35
Total	**100**	**100**

TARGET LOAD LEVELS

The last step in modelling the workload involves stating the target load levels. This will be represented by the number of requests LARMAS should expect and handle per unit time. To get an estimate of the total number of users LARMAS will have, we must first estimate the number of users and contributors it will have. LARMAS API can support desktop, mobile and web-applications. The target market is, therefore, university students who have access to these devices. There is a total of 958,212 university students in South Africa as of 2015 [62]. 60% of adults in South Africa have smartphones, 18% have personal computers and 7% have a tablet device [63]. It is safe to assume that all adults with personal computers or a tablet device will also have a smartphone. 59% of smartphone users access the internet every day and these are the users who would be in the best position to contribute to LARMAS [63].

Not everyone will be willing (or able) to contribute to LARMAS. Market penetration is the measure of the adoption of a product or service compared to the total theoretical market size. Marlene Jensen suggests multiplying the theoretical market size by a percentage between 10 and 40 percent to get an estimate of the normal market penetration for a business product [64]. Using a market penetration of 40%, the total number of users of LARMAS will be:

$$Number\ of\ Users = 958\ 212 \times 60\% \times 40\% \times 59\% = 135\ 683\ \approx 136\ 000$$

This number is still an initial estimate and will be revised over time during the project's lifetime.

TABLE 8: THE TARGET LOAD LEVELS FOR THE DIFFERENT SCENARIOS

Key Scenario	Normal Load Requests (per hour)	Peak Load Requests (per hour)	Peak Load Duration (per day)
S1	5,250	10,500	1
S2	4,500	9,000	1
S3	2,250	4,500	1
S4	1,500	3,000	1
S5	1,500	3,000	1
Total	**15,000**	**30,000**	**1**

3.2.2.3. Scalability

Scalability is the ability of a system to grow in capacity to meet its increasing performance demands [65]. LARMAS will start with a low number of users and is expected to grow rapidly as more client-side applications are developed. It should, therefore, be designed in such a way that will make it easy to scale **up** and **out** without affecting performance, availability and reliability as the number of users increases.

Scaling up is increasing the system's resources to increase its performance and capacity to handle the increasing load [65]. This involves adding more RAM, network adapters or CPUs to the existing setup. These extra resources will enable the server to launch more threads and thus handle more requests. However, there is a limit to which you can scale up a server before one started getting diminishing returns.

Scaling out is adding more servers to meet the increasing workload. Because scaling out introduces more nodes to the system, it improves reliability and availability since there more servers to handle the request as opposed to having one server.

3.3. Development Environment

- The web application will be developed in **Ubuntu 16.04.3 LTS** (Xenial Xerus).
- **Python 3** will be programming language of choice and the **Django Web Framework** will be used for the web application. The reasons for choosing Python and Django are:
    - Python has a small learning curve which means it is easy to learn therefore less time will be spent trying to learn the language and more time will be devoted to the project.
    - Python has high readability. This is a highly desirable property since this project will be shared and used with other researchers and developers, most of whom are not primarily software developers, but major in other fields like engineering, language processing, etc.
    - Python is one of the most used programming languages which means it has a large supporting community and a wide variety of open-source libraries.
    - Django is a very popular python based web framework which has a large support community.
    - Django is thoroughly tested and is used for some of the largest web applications including:
        - Instagram (https://instagram.com): A popular image sharing platform [28].
        - Pinterest: A popular image sharing platform.
        - Bitbucket (https://bitbucket.org): A web hosting service for version control systems like Git and Mercurial [66].
        - Disqus (https://disqus.com): A popular online blog commenting hosting service [29].
        - Mozilla Firefox (https://www.mozilla.org/): The website for Mozilla (includes support and extensions pages) for the popular web browser, Firefox.
        - The Washington Post (https://www.washingtonpost.com): a popular and influential newspaper. [67]
- **SQLite** will be used as the database engine during the early stages of development. A suitable DBMS will be thoroughly investigated once the core functionality of the system has been implemented.
- **Git** will be used for version control and the source code will be hosted on **GitHub** [68].

- **Travis-CI** will be used as the continuous integration service to build and test the web application [69].
- **HAProxy** will be the load balancer for LARMAS.
- **OpenStack Swift**: The object storage technology used for file storage.
- **Apache HTTP Server**: The server that was used to server LARMAS in tests and experiments

3.4. Database Design

The web application will be used to collect different types of data specified under sections **2.2** for MT and **2.3** for ASR. Users will be created via the API end-points through any suitable client (web, mobile or desktop). These users will then be able to upload data to the LARMAS in different ways depending on the data as specified below.

3.4.1. Annotations

Prompts in the context of this project are a collection of short phrases that are specially designed to cover all the important linguistic structures (grammar and style). LARMAS will store a large collection of these prompts for each language, possibly hundreds of thousands. The prompts will be evenly distributed to registered users who will then record and upload their recordings together with annotations and metadata which will be used in ASP processing. The database schema for the collection and storage of prompts and annotations is shown in Figure 17.

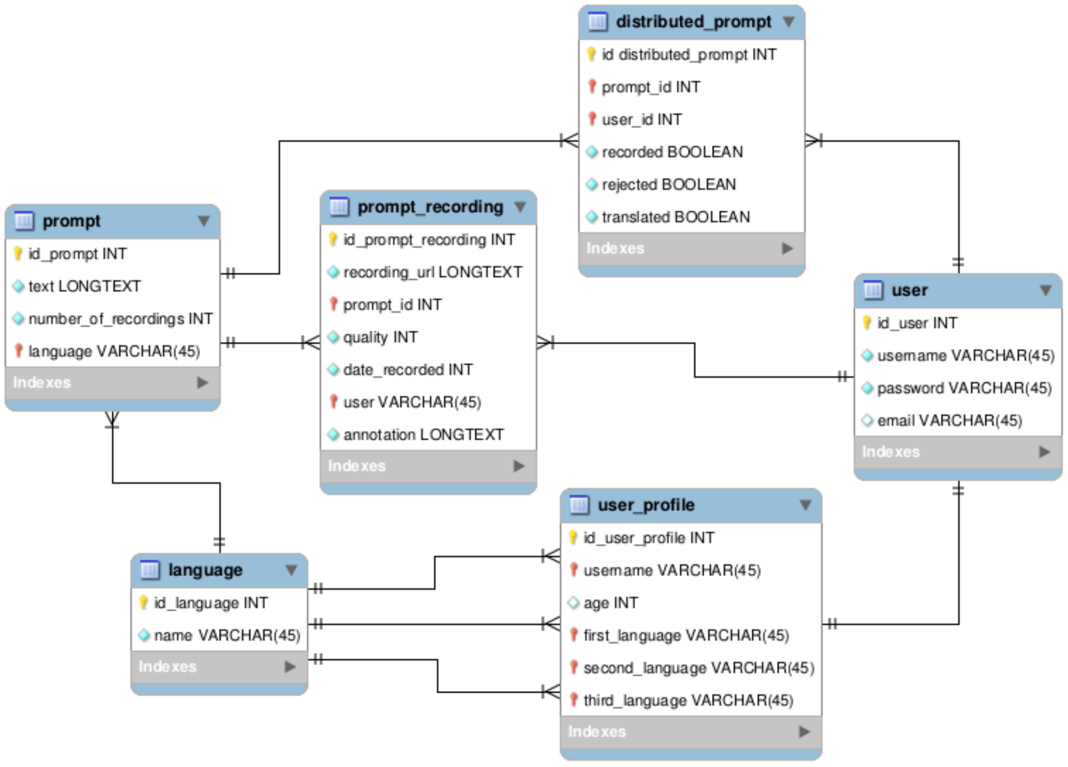


FIGURE 17: DATABASE SCHEMA FOR THE COLLECTION OF ANNOTATIONS

The "Prompt" table will store the prompts and the number of recordings collected for each of the prompts. Each request to LARMAS for prompts will return a list of 50 prompts. Users will also be allowed to reject a prompt. To avoid sending the same prompt or a rejected prompt to a user, the server will first check if that prompt has already been sent to the user in the "Distributed Prompt" table, a table which will keep track of all prompts that have been sent to the users and whether they have been recorded or rejected.

The “Prompt Recording” table keeps records of all the prompts that have been recorded and uploaded to the server. Each record in this table has information on which user uploaded the prompt, the id of the prompt recorded, the quality rating of the recording and the annotation of the recording. When a user uploads a prompt, the server will first verify the authenticity and integrity of the audio file. After verifying the rest of the metadata, the server stores the recording and responds to the client with a success message.

Prompts must be evenly distributed to contributing users and there are several ways of doing this.

1. Using the Law of Large Numbers. The Law of Large numbers in probability theory states that as the number of experiments increases, the ratio of outcomes will converge to the theoretical or expected ratio of outcomes [70]. If a prompt is chosen at random by generating a random **prompt_id** for distributing prompts, the probability of choosing a prompt (D) using the classical theory of probability [71] is given by:
$$P(D) = \frac{1}{n}$$
$$where\ \boldsymbol{n}\ is\ the\ total\ number\ of\ prompts.$$
P(D) is constant therefore as the number of requests for prompts increases, the distribution of prompts will be normal. However, this is only true if n is constant. If n varies, P(D) will also vary and this will lead to uneven distribution of prompts (the first prompts added to the system to have more recordings than newer ones)
2. Sorting or querying the table for each request to get the prompts with the lowest number of recordings. This option will only be realistic if there are only a few hundred or a few thousand prompts. However, because this system will store possibly millions of prompts from all the different languages, processing each request for prompts will take a long time as the DBMS sorts through the records.
3. Asynchronously and periodically querying the database for the prompts with the lowest number of recordings and storing them in a cache to be distributed later. This allows for requests to be processed much faster than the option above since only a few prompts will be searched through. The problem arises when trying to determine how many prompts will be stored in the cache. As the number of users grows, so will the size of the cache and this will eventually slow down the server as the cache gets too large.
4. Keeping a sorted table. Requests to distribute prompts will be processed much faster since they will be selected from the top of the prompts table which will be stored in ascending order of a number of recordings. The drawback of this option is that the database will have to be sorted every time a user uploads a recording to the system which will lead to long server response times.

Option 1 is the most efficient way to distribute prompts, provided that the number of prompts stays constant. However, the number of prompts will change as more prompts are generated and added to the system by the administrators throughout the project’s lifetime and this will skew the observed distribution of the number of recordings for the prompts. The central limit theory states that as the sample size increases to infinity, the distribution of the sample means approaches a normal distribution [72]. The sample size at which a distribution can be assumed to be normal is 50 (or 30 in some cases) as a rule of thumb [73]. LARMAS will store hundreds to thousands of prompts for each language, therefore the distribution of prompts will be normal according to the central limit theory.

3.4.2. Building Parallel Corpora

Figure 18 shows the database schema for the storage of prompt translations. Users will be able to translate the prompts they receive into a different language specified in their profile. This will create a parallel text corpus using the prompts collected by the system and the translations can also be used as prompts for other languages supported by the system. The table will also store alignment data which will be used in MT. This alignment can be performed on the client side and be uploaded to the server along with the translated text or alignment data can be added to the table at a later stage either using software that would do it automatically or manually.

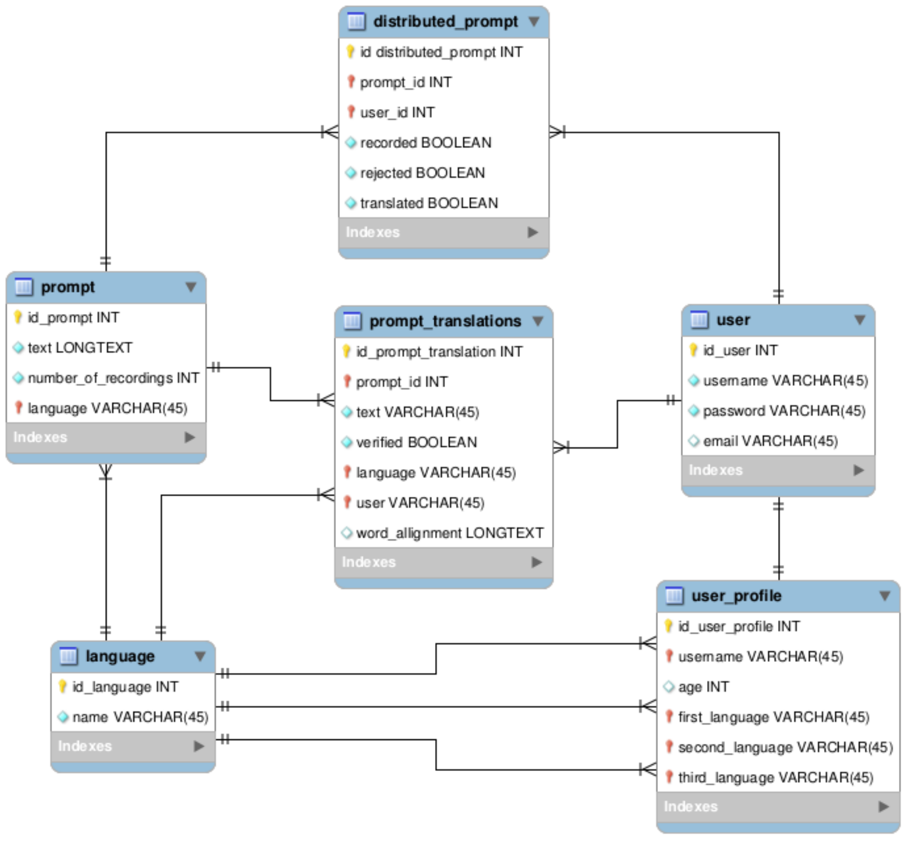


FIGURE 18: DATABASE SCHEMA FOR THE COLLECTION OF PROMPT TRANSLATIONS

Because only a single translation is needed per prompt per language, there will be no need to design another distribution system for prompts for translations. Users can translate the same prompts they receive.

3.4.3. One-time Contributions

LARMAS is designed primarily for client-side applications that will be used by users who want to contribute on a long-term basis, hence the need for users to register to store their details (age, language, region, etc.) so that they do not have to enter that information any time they contribute to the system in the future. However, not everyone will be willing to be a long-term contributor. LARMAS should also allow for researchers who want to organise events for data collection where people will be free to contribute just a few recordings or translations. Making a user register just to contribute once or twice would not be intuitive and may discourage some people.

LARMAS will have a feature to support one-time contributors. This will allow users to contribute without having to register. The same API endpoints will be used to retrieve prompts and upload annotations and recordings but users will have to provide the extra information such as translation language, age, first language, etc. On the server side, a dumb *user* and *user profile* will be created to store this information. However, the user object will be marked as inactive to prevent them from logging in using the automatically generated username.

3.5. Application Architecture

Three architectural patterns will be investigated for the LARMAS web application architecture. These are *Space-Based* architecture, *Layered* architecture and *Event-Driven* architecture.

3.5.1. Space-Based Architecture

A brief overview of this architecture is under section 2.7.5. Figure 19 shows how the space-based architecture for LARMAS would look like.

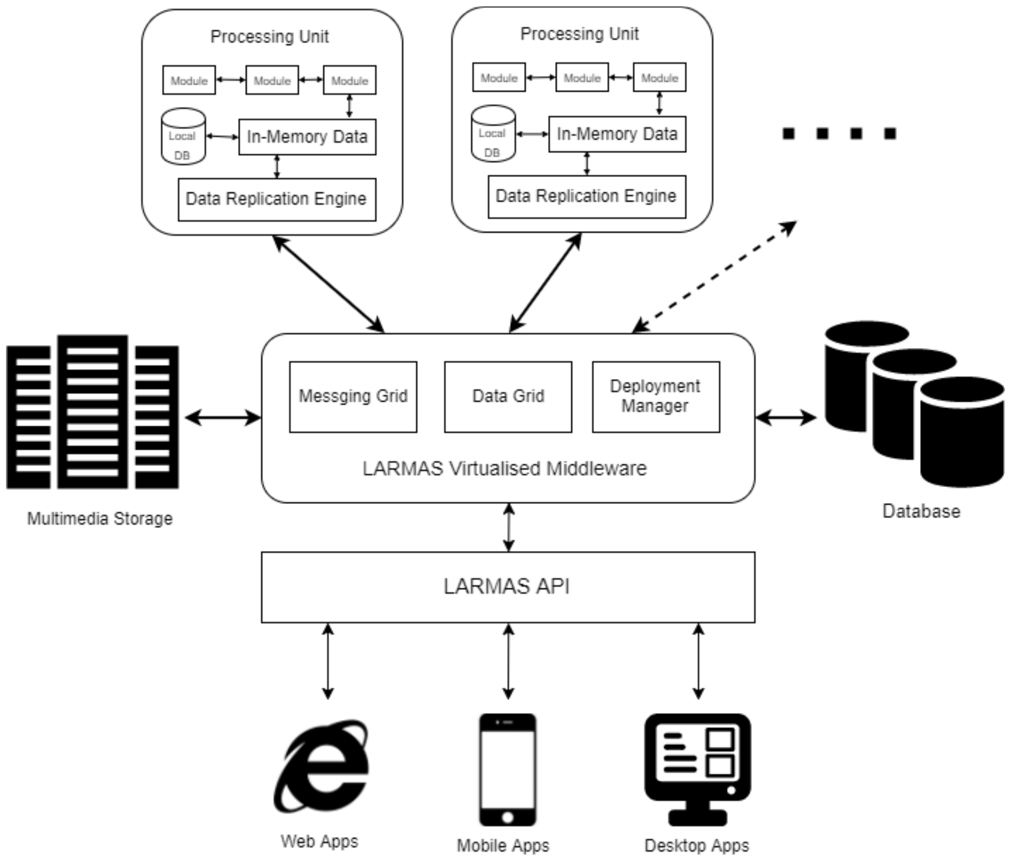


FIGURE 19: LARMAS SPACE BASED ARCHITECTURE

The biggest advantage of this architecture is how easy it is to scale it. It can easily be scaled by adding more processing unit nodes to handle the extra load. Because of this, processing units have to be self-sufficient. In this case, the processing units will contain all the necessary functions to process all the different types of requests from the users, from collecting annotations to managing prompts. The virtualised middleware will manage synchronisation of data, communication and deployment of the processing units.

The Processing Unit

The processing unit is a self-contained unit of scalability. It is normally built using Plain-Old-Java-Object (POJO) which is provided by web frameworks such as Spring Framework. This is processing of data will take place. When building parallel corpora, the units will organise the translations and store them in the databases. The processing units may also have the added functionality to perform word alignments using machine learning algorithms or manually by an authorised administrator. When collecting annotations, the processing unit is responsible for organising the annotations and storing them in the database. The raw media that is transferred along with the annotations (audio for speech annotations and video or sensor signals for sign language annotation) is also processed in the processing unit to ensure that it is valid for usage.

Each processing unit will have a Data Replication Engine which will be used by the virtualised middleware to replicate and synchronise data changes made by one processing unit to other active processing units.

Virtualised Middleware

Besides managing authentication and session information, the messaging grid receives incoming requests from the users and forwards them to the active processing units. The messaging grid will also manage communication between services in the middleware and across the processing units. The data grid component manages access to the data in the distributed memory with the possibility of synchronising it with a database. This data will include user details, prompts, prompt recordings and translations. The processing grid will not be implemented in this design because no parallel processing will be done in LARMAS. However, future development may find a need to implement a processing grid for processing the raw data collected by the system.

3.5.2. Microkernel Architecture

This architecture is described in detail in section 2.7.3. LARMAS is a system that is still in the early stages of development. Because of this, many features are missing and there is a possibility that more features would need to be added to the system. Currently, LARMAS is being designed to perform two types of data collection, annotations and translations. These two functions can be thought of as add-ons to the larger LARMAS system. Visualising them this way allows us to create a microkernel architecture. The microkernel architecture was designed primarily for operating systems but the same concepts can be applied to LARMAS. Figure 20 a microkernel architecture design for LARMAS.

Core System

Just like the core system of an operating system which implements the low-level functionality of the operating system, the core system of LARMAS will have the basic features that are universally found in the whole system and across all the modules. Such features include security, user management, session management and inter-communication across the different services and plug-ins.

Plug-in Components

The plug-in components will each implement a separate feature of the system. This allows for more features to be developed in the future and only need to be installed on the core system. The plug-ins for LARMAS would be ***annotations***, to manage a collection of annotations, ***translations***, to manage collections of translations, ***prompts***, to manage the distribution of prompts and pre-processing which will process the collected raw data to verify its authenticity and measure its quality. Other possible plug-ins that can be developed in the future include one for collecting grammar dictionaries and a data warehouse or plug-ins that will have access to the rest of the stored data to carry out natural language processing like MT and ASR.

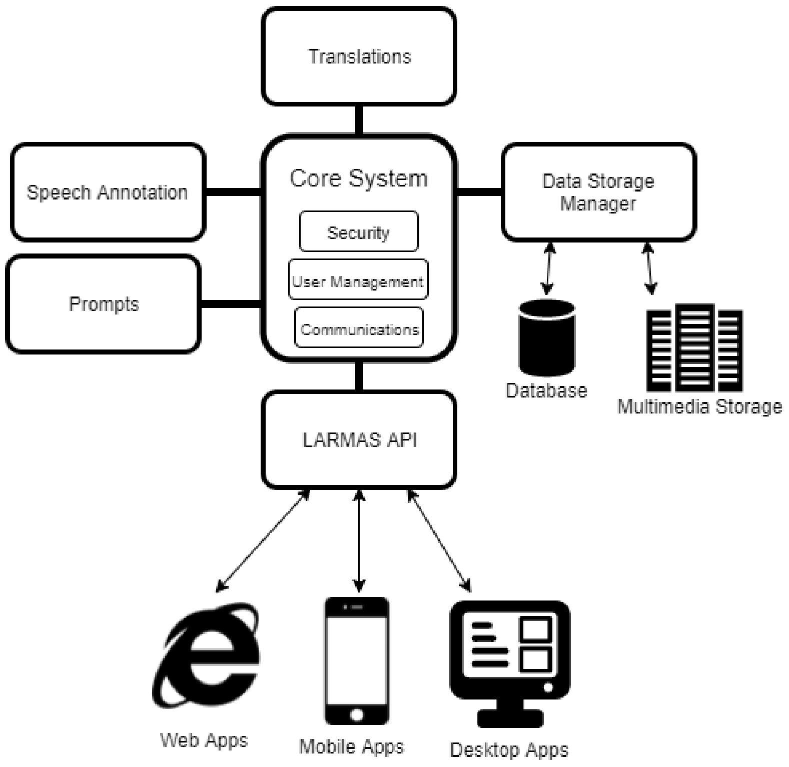


FIGURE 20: LARMAS MICROKERNEL ARCHITECTURE

3.1.1. Layered Architecture

A brief overview of this architecture is found under section 2.7.1. Figure 21 shows a layered architecture design for LARMAS.

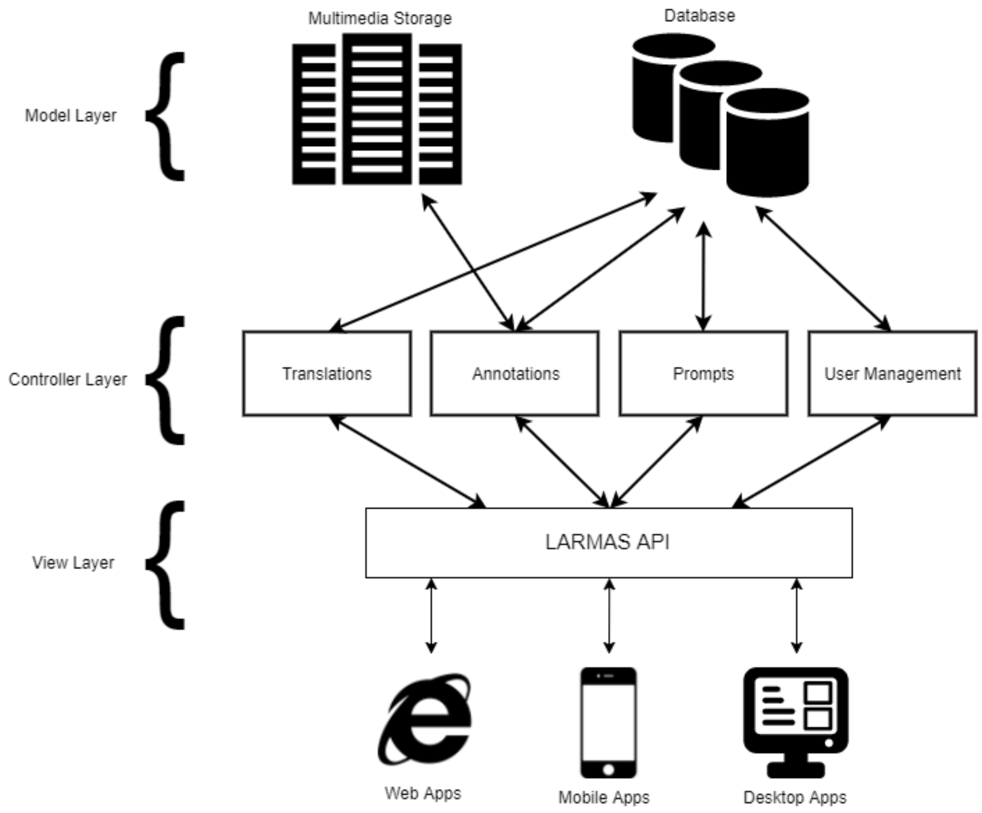


FIGURE 21: LARMAS LAYERED ARCHITECTURE

The primary purposes of LARMAS are to collect, process and store data which will later be presented to third-party applications and systems for natural language processing. Therefore, there is need to separate these core functionalities. For these reasons, the Layered Architecture will be the most suitable architecture (described in section 2.7.1) to separate these functionalities. LARMAS would be made up of three layers to create an architecture called Model-View-Controller (MVC).

Model

The Model layer represents the data stored in the system; namely, the database and file storage. The database will store data about users, authentication, session management, prompts, annotations metadata and translations. This data is in the form of text and numbers which is small enough in size to store efficiently in a database. The raw files collected by LARMAS are often large files which can be several megabytes each. These will be stored outside the database in some form of file storage system like the server's filesystem or object storage.

Controller

The processing of the data and handling requests from clients will be done in the Controller layer. Some of the functions found in the controller include user management, session management, collection of annotations and translations and processing raw data.

View

Output data from the processing that takes place in the controller will be presented in the View layer. The early stages of development for LARMAS will only present data through the API, therefore, it would need further rendering on the receiving client to make it more understandable to the user.

3.1.2. LARMAS Architecture

The three architectures described above each have their own advantages and disadvantages and to create the optimum architecture that satisfies both functional and non-functional requirements, features from all three will be integrated into the final design. Because LARMAS is a new application that will be developed rapidly, the layered architecture will be the overall architecture for the system.

The Django Web Framework is designed for 3-tier architectures like the MVC. The Controller layer of LARMAS will be designed using a microkernel approach. Django has a feature which allows the developer to separate the functionality in the controller layer into what the framework calls "*apps*". LARMAS has the following Django-apps (similar functionality to "plug-ins" in a microkernel architecture):

- **user**: manages the users on the system. This *app* controls access to the *Users* and *User Profile* models where the login details and other important details like names, age, languages and contact information about the users are stored. Please note that any identifying information of the users is optional and only requested for when necessary. Users will only need to enter the languages they speak, region, a username and password to use LARMAS.
- **prompts**: manages the distribution of prompts to the users and authorised system administrators will be able to modify and add more prompts to the database.
- **annotations**: manages collection of annotations along with their raw data. The raw data will be stored in the system's file storage system. Authorised system administrators will be able to modify annotation data (to verify and delete it if not usable).
- **translations**: manages collection of translations of prompts. Authorised administrators will be able to verify the validity of the data and modify or delete when necessary.

Database tables in Django are created using *Models*. A *Model* is a python class that inherits from ***django.db.models.Model*** and each field in a model corresponds to a column in the database table. Django abstracts the database API using these *Models* by creating an API based on the configured DBMS, that allows the user to create, update, retrieve and delete entries in the database (referred to as objects). Each *app* can define its own models and these *models* can be accessed by other apps when necessary. Django provides the following database backend implementations [74]:

- PostgreSQL
- SQLite
- MySQL
- Oracle-DB

SQLite will be used for development because it is lightweight and quick to set up for tests and simulations. Once the core functionalities of the system are implemented, different DBMSs will be tested to determine which one will be most suitable for LARMAS.

The LARMAS API will be implemented in the *Views*, classes meant to encapsulate the processing of requests and generating responses to users. Views have been separated into the different Django-apps to keep the functionalities separate.

Figure 22 shows the final architecture used for LARMAS.

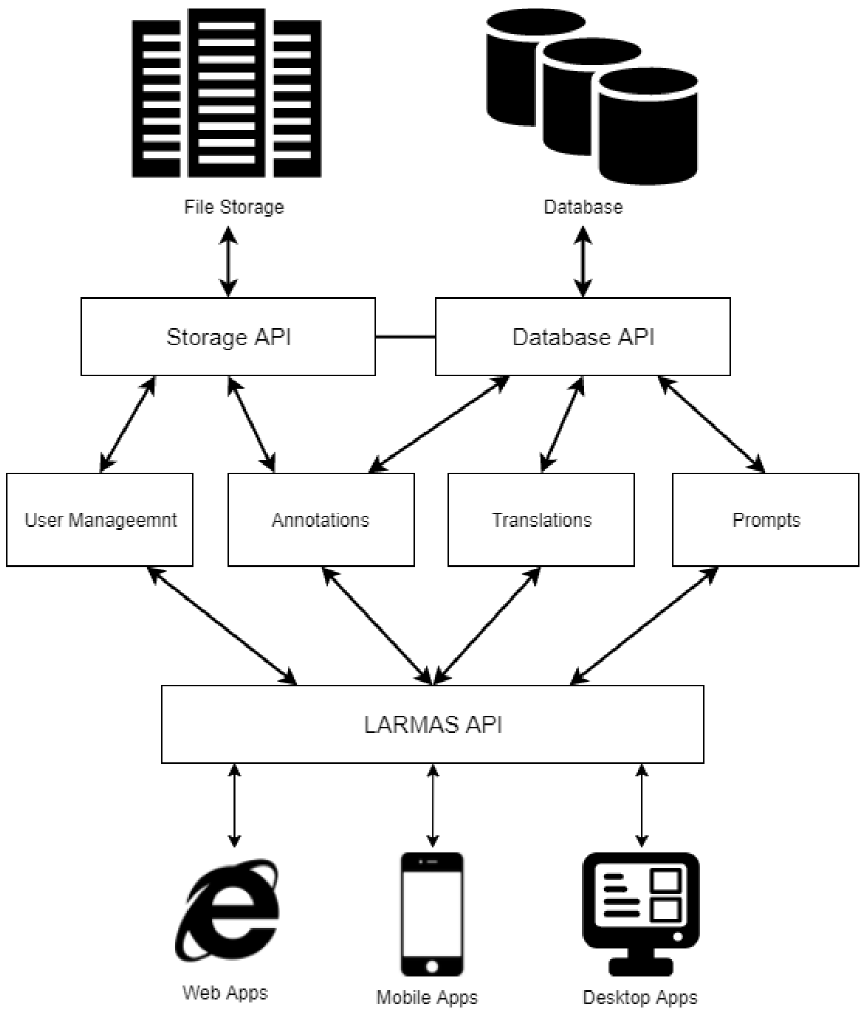


FIGURE 22: ARCHITECTURE USED FOR LARMAS

3.2. LARMAS API

REST was used to implement LARMAS API. It was chosen because it is easy to adapt and be used by developers who will be developing client-side applications making it possible to quickly develop LARMAS and the mobile applications that will be using it. Because of its platform independence and separation of client and server, REST provides good performance and high scalability. LARMAS is being designed to manage language resources; which means it will have to allow resources to be manipulated and deleted and REST is designed to make CRUD operations easy for client-side applications.

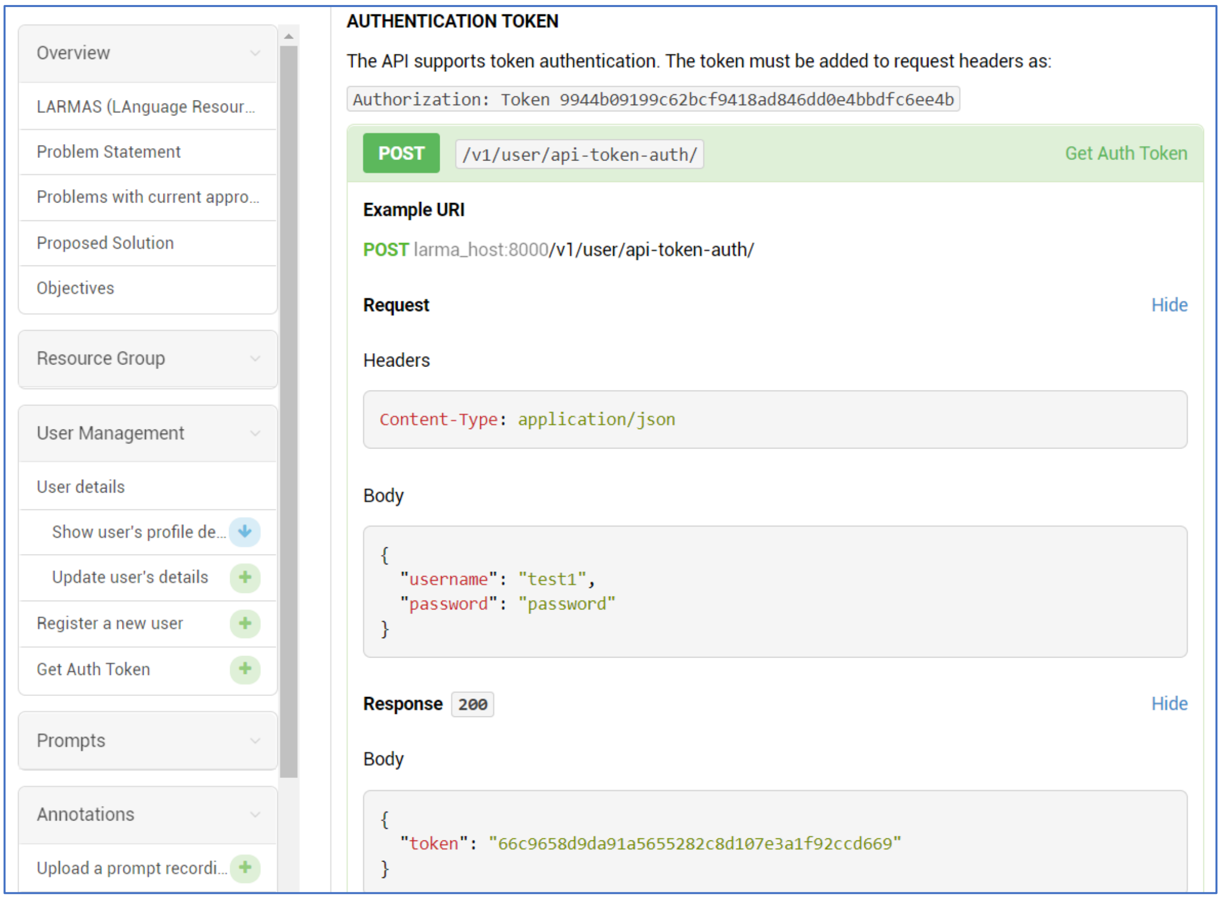
TABLE 9: DESCRIPTION OF ALL THE ENDPOINTS SUPPORTED BY LARMAS API

HTTP Method	URL	Description
GET	/v1/user/	Get the details of the user that is logged in.
GET	/v1/user/{user_id}	Get the details of a user by ID. **Admin Only**
POST	/v1/user/	Update details of a user.
POST	/v1/user/register/	Create a new user account.
POST	/v1/user/api-token-auth/	Get Auth-Token of a user.
GET	/v1/prompts/	Get a paginated response with a list of all prompts on LARMAS.
GET	/v1/prompts/{id}/	Get a prompt by ID
GET	/v1/prompts/retrieve/	For registered users and one-time contributors to retrieve prompts to record and annotate.
PUT	/v1/prompts/reject/{id}/	For registered users to reject a prompt.
GET	/v1/annotations/	Get a paginated response of all the annotations and links to their raw recording.
GET	/v1/annotations/{language}/	Get a paginated response of all annotations of a specific language.
GET	/v1/annotations/{id}/	Get an annotation by ID with a link to the raw recording.
POST	/v1/annotations/upload/	Upload an annotation along with its recording. For registered users and one-time contributors
GET	/v1/prompt_translations/	Get a paginated response of all the translations on LARMAS
GET	/v1/prompt_translations/{id}/	Get a translation by ID
POST	/v1/prompt_translations/upload/	For both registered users and one-time contributors to upload a translation.
GET	/v1/prompt_translations/parallel/{1st}/{2nd}	Get a paginated response of parallel text between two languages.

3.2.1. Usability of LARMAS API

On its homepage, LARMAS has complete documentation of its API with directions and examples of how to use it. LARMAS also has a browsable API that allows third-party developers to interact with the LARMAS API in a browser. They will be able to test it out, make requests and use it to get to grips with the API before integrating it into their programs. To access this browsable API, developers can simply enter the endpoint URL's in a browser and the LARMAS API will respond with HTML pages of the browsable API instead of responding with JSON or XML data. An example of a page on the browsable API is shown in Figure 24.

Figure 25 is a screenshot of the administrator control panel. This panel is only visible to users with administrator privileges. On this panel, administrator users can perform CRUD operations on any data stored on LARMAS. This data includes users and user profiles, authentication tokens, annotations, prompts and translations. It can be used to perform review uploaded content and manage users and the data stored on the system.



AUTHENTICATION TOKEN

The API supports token authentication. The token must be added to request headers as:

```
Authorization: Token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b
```

POST `/v1/user/api-token-auth/` Get Auth Token

Example URI

POST larma_host:8000/v1/user/api-token-auth/

Request Hide

Headers

```
Content-Type: application/json
```

Body

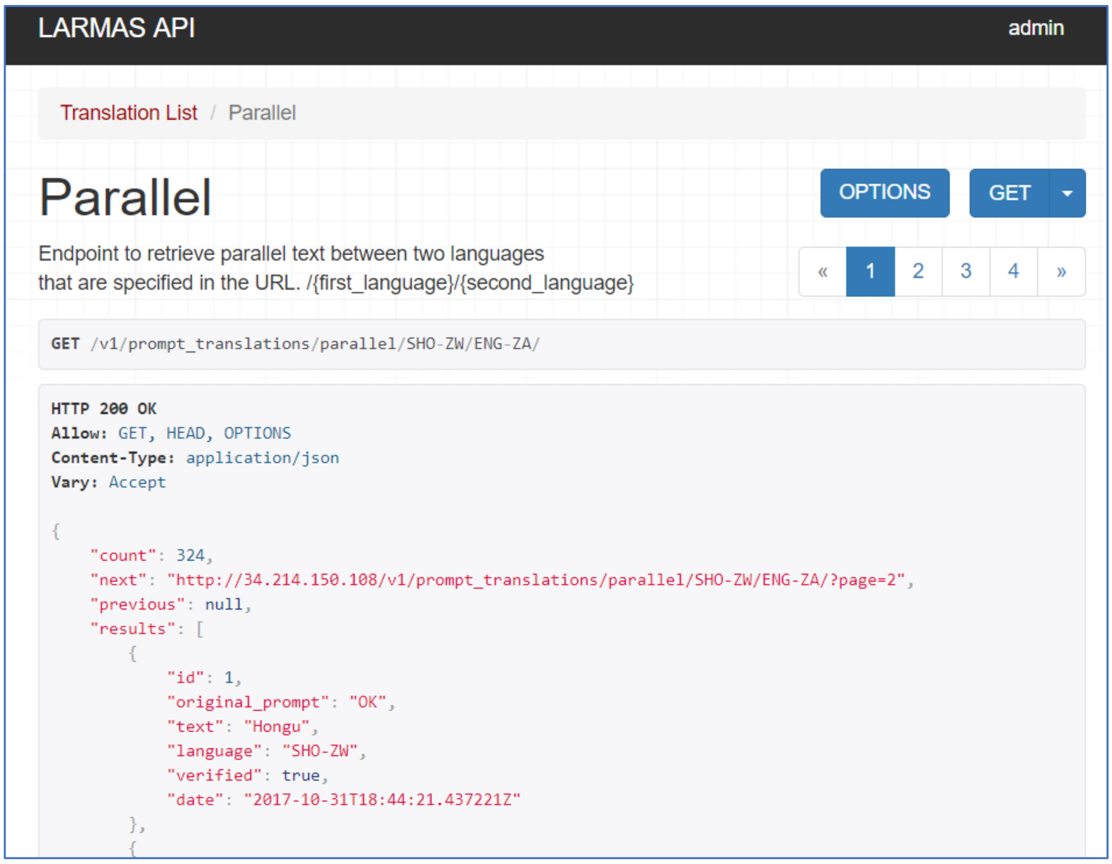
```
{
  "username": "test1",
  "password": "password"
}
```

Response 200 Hide

Body

```
{
  "token": "66c9658d9da91a5655282c8d107e3a1f92ccd669"
}
```

FIGURE 23: A SCREENSHOT OF AN API ENDPOINT TO RETRIEVE AN AUTHENTICATION TOKEN



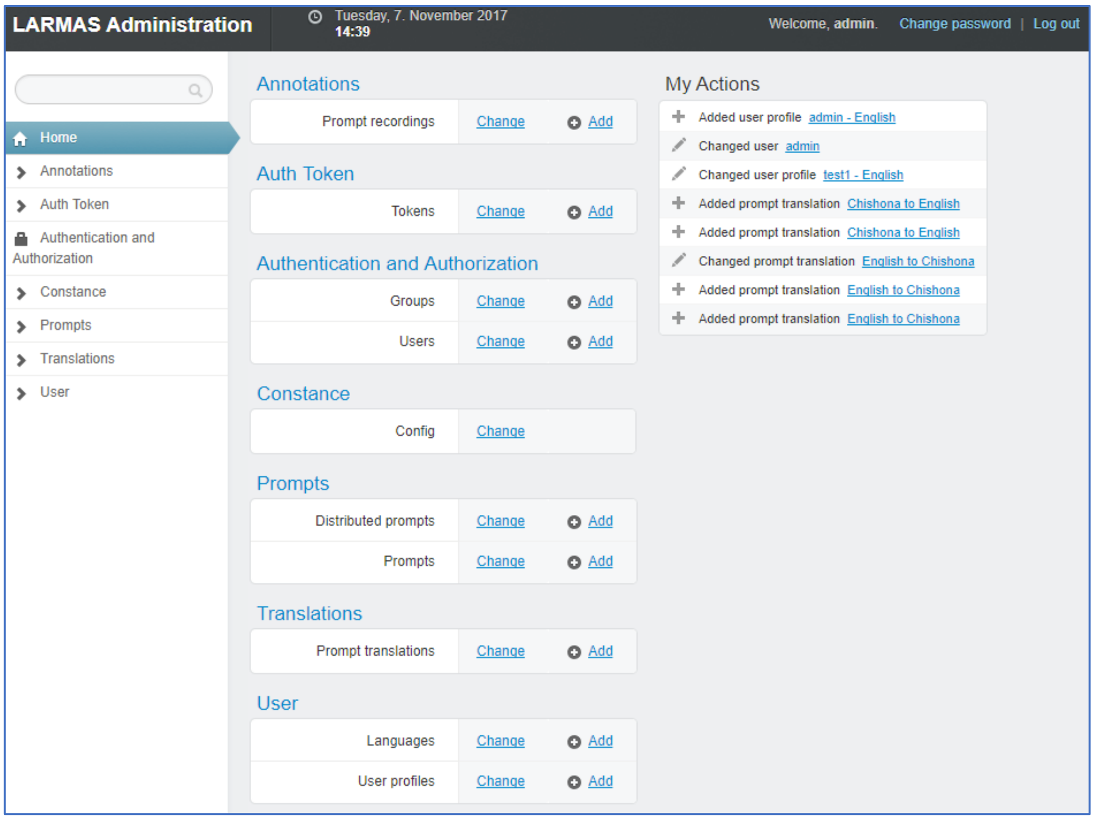
Endpoint to retrieve parallel text between two languages that are specified in the URL. /{first_language}/{second_language}

```
GET /v1/prompt_translations/parallel/SHO-ZW/ENG-ZA/
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "count": 324,
    "next": "http://34.214.150.108/v1/prompt_translations/parallel/SHO-ZW/ENG-ZA/?page=2",
    "previous": null,
    "results": [
        {
            "id": 1,
            "original_prompt": "OK",
            "text": "Hongu",
            "language": "SHO-ZW",
            "verified": true,
            "date": "2017-10-31T18:44:21.437221Z"
        },
        {
```

FIGURE 24: SCREENSHOT OF THE BROWSABLE API.



Annotations

Prompt recordings	Change	Add

Auth Token

Tokens	Change	Add

Authentication and Authorization

Groups	Change	Add
Users	Change	Add

Constance

Config	Change

Prompts

Distributed prompts	Change	Add
Prompts	Change	Add

Translations

Prompt translations	Change	Add

User

Languages	Change	Add
User profiles	Change	Add

My Actions

- Added user profile admin - English
- Changed user admin
- Changed user profile test1 - English
- Added prompt translation Chishona to English
- Added prompt translation Chishona to English
- Changed prompt translation English to Chishona
- Added prompt translation English to Chishona
- Added prompt translation English to Chishona

FIGURE 25: SCREENSHOT OF THE LARMAS ADMINISTRATION PANEL

3.3. File Storage

Object storage will be the file storage solution for LARMAS. Object Storage is a highly scalable and reliable form of file storage thanks to its distributed architecture. Because the resources stored in LARMAS will be provided over the internet, object storage will be used for its access to the files via HTTP, a protocol that can be used on any device that has access to the internet. Because of its architecture, object storage does not need any backing up or recovery plans. Swift Object Storage, the object storage service used in LARMAS, is described below.

3.3.1. Swift Object Storage

Swift Object storage (also known as OpenStack Object Storage, OpenStack Swift or just Swift) will be used to store the multimedia data and other files that will be collected by LARMAS. Swift is a distributed object storage which offers high availability, concurrency and durability; designed for storing unstructured such as audio, video, documents and images along with their metadata. This is consistent with the goals of LARMAS which aims to collect a wide variety of data for language processing. Data is organised in a hierarchy consisting of three levels. These are:

- **Account**: The top level which defines the namespace of the containers. Containers can belong to different accounts.
- **Containers**: The second level which defines the namespace for objects. Containers manage storage and access control to the objects. Objects with the same name but under different containers are represented as different objects.
- **Object**: This level stores the actual data in any format along with the metadata.

Swift has a REST based API [75]. The API reference describes operations supported by the storage API for each of the three levels. Each object has its own unique URL which can be used to access the object. The URL of an object is made up of the account name, container name and the object name, as shown below:

https://cloud.com/api/{account_name}/{container_name}/picture.jpg

Swift Architecture Overview

PROXY SERVER

The proxy server is a front-end layer that is responsible for implementing the Swift API and brings together the rest of architecture. It accepts requests from the clients and locates the account, container and/or the relevant object inside The Ring (described below). The proxy server also encodes and decodes object data using *erasure code,* a type of forward error correction which tolerates the loss of some of the encoded fragments [76].

THE RING

The ring maps names of objects on the disk to their physical locations. There are separate rings for accounts and containers and one object ring per storage policy. Rings are used by the proxy server and other swift processes to locate and interact with accounts, containers and objects within the storage cluster. The Ring uses a modified MD5 hashing algorithm to map the object to a partition.

STORAGE NODES

Storage nodes are responsible for the actual storage of data and respond to the proxy server. A minimum of 3 storage nodes is required to run Swift. To write data into Swift, an HTTP PUT request for the object is sent to the Proxy Server. The Proxy Server communicates with the appropriate storage nodes, via the Ring, and if most of the storage nodes $\left(\frac{2}{3}\right)$ successfully store the data, the Proxy server will report the data as successfully stored. In the case of a failure, the other storage nodes will ensure that data is replicated onto another storage node to keep the same number of copies of data.

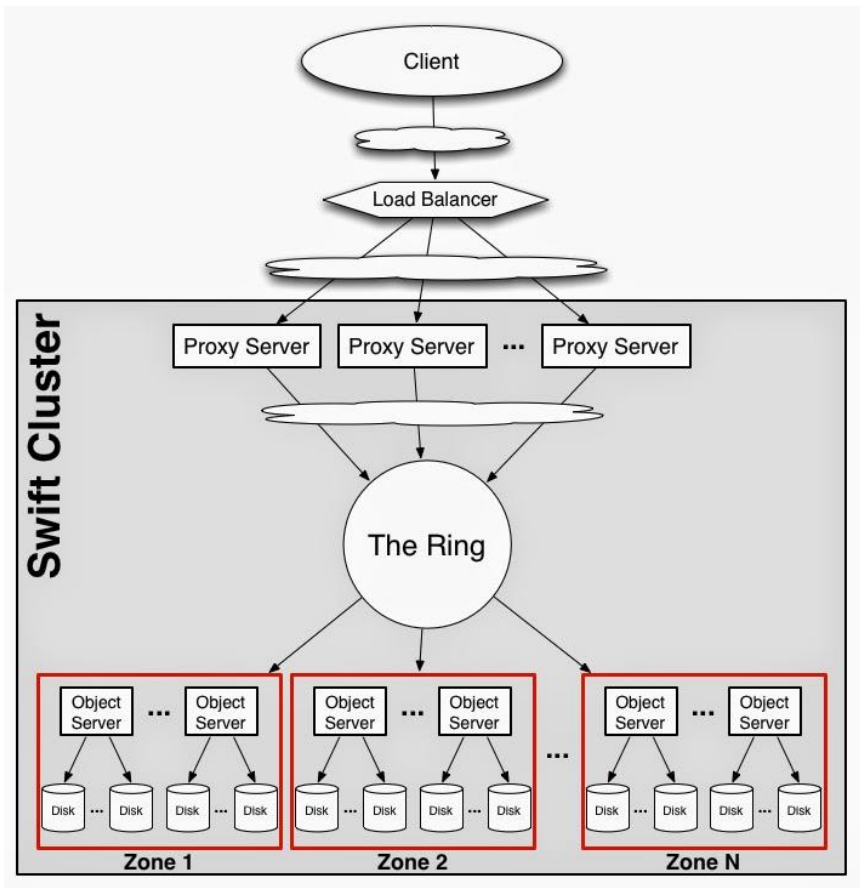


FIGURE 26: DIAGRAM SHOWING THE ARCHITECTURE OF A SWIFT OBJECT STORAGE CLUSTER. "THE RING" IN THIS DIAGRAM IS USED TO REFER TO ALL THE DIFFERENT RINGS USED TO ACCESS THE DIFFERENT COMPONENTS IN THE CLUSTER [77].

REPLICATION

Replication is used to keep the system in a consistent state despite hardware failures. It does this by comparing the data with remote copies to make sure that it remains consistent throughout operation and also verifies it using MD5 Checksum. The replicator also ensures that data is deleted from the system. When objects, containers or accounts are deleted, a tombstone is set to indicate the deletion and this tombstone will be detected by the replicator which will make sure that the data is removed from the system.

AUTHENTICATION

Swift supports several authentication systems with the following common characteristics;

- Can be part of an external system or a subsystem running with Swift as WSGI (Web Server Gateway Interface) middleware.
- Swift user passes an authentication token with each request
- Swift validates each token with the authentication system (external or subsystem) and caches the result.
- The token will eventually expire but does not change between requests.

The authentication token will be sent in a request header using *X-Auth-Token* or *X-Storage-Token* headers. Swift will make calls to the authentication system, giving the token to be validated and the authentication responds with an expiration time. Swift natively supports two authentication systems [78].

- **TempAuth**: A lightweight authentication system used for testing and development environment. The middleware is responsible for creating tokens and which are given to the users when they provide a username and password.
- **Keystone Auth**: OpenStack's Keystone Identity Services is middleware for creating and validating tokens. It is robust and designed for use in production servers.

Swift also allows third-party authentication systems to be used with Swift. A good example is SWAuth, an authentication service installed in the WSGI pipeline that uses Swift itself as a backing sore [78].

Setting up Swift for LARMAS Development.

The detailed setup for Swift in LARMAS is shown under Appendix B. Swift requires at least 4 servers to operate the way it was designed to do so in a production environment. One server will be used to setup the Proxy server, the other 3 will be used as storage nodes. Ideally, these nodes will be located in different geographical locations in case of failure in a specific location. Block storage can actually be used to implement the storage node and this setup will also incorporate advantages and flexibility of block storage especially when there is need to increase the size of the storage nodes. OpenStack recommends the following minimum requirements for the proxy server and storage nodes [79]:

Specifications	Storage Nodes	Proxy Server
Processor	Dual-Core CPU	4+ CPUs
Memory	2 GB RAM	8+ GB RAM
Network	1 Network Card	2 Network Cards
Storage Space	100+ GB but varies with system requirements.	100+ GB

TABLE 10: TABLE SHOWING THE MINIMUM REQUIREMENTS FOR SETTING UP SWIFT OBJECT STORAGE

As seen from Table 10, the proxy server needs 2 network cards while he storage nodes only need one. This is because Swift operates with 2 networks, a public one which will be used by clients to send requests to the proxy server and a private network which will be used for communication among the storage nodes and the proxy server. These two networks are physically (or sometimes virtually) to prevent direct public access to the storage nodes.

Due to the complexities of setting up Swift object storage, normal filesystem storage will be used during development. However, Swift will be used in the Integration testing. Instead of having 4 different servers to run Swift, the three storage nodes will be simulated using virtual hard-drives attached an Ubuntu Virtual machine. This setup allows us to use a single virtual machine to simulate a complete Swift environment for object storage.

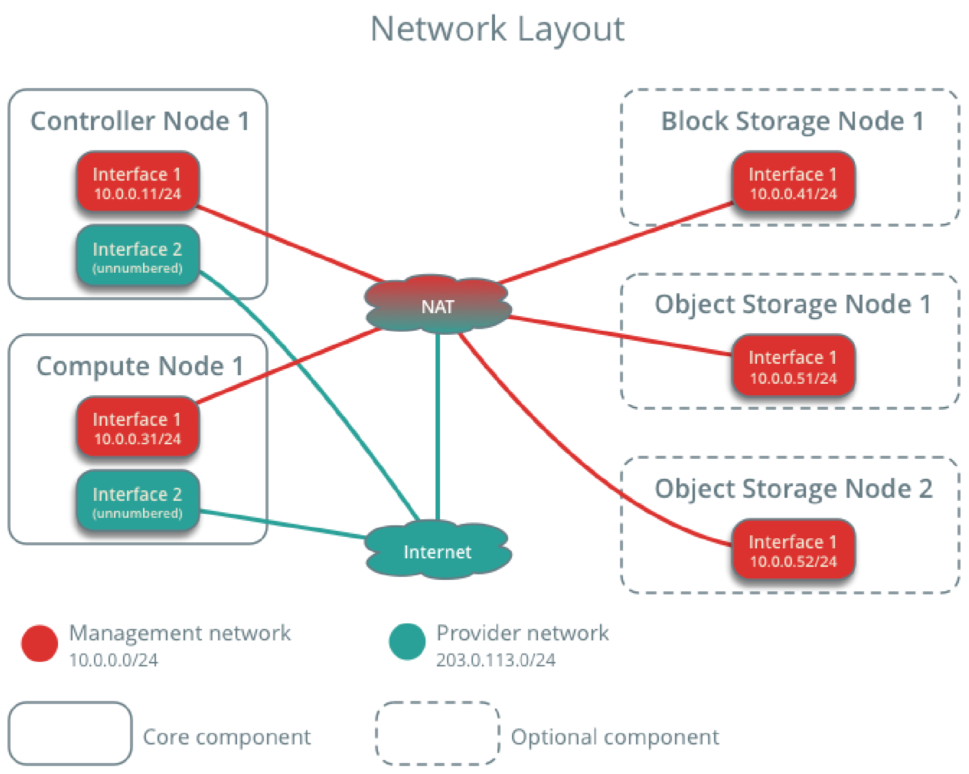


FIGURE 27 DIAGRAM SHOWING THE NETWORK SETUP FOR AN OPENSTACK -ENVIRONMENT. NOTE THAT THIS SHOWS A GENERIC SETUP WHICH HAS A COMPUTE NODE WHICH IS USED FOR OTHER OPENSTACK SERVICES WHICH ARE NOT USED IN LARMAS

3.4. Server Architecture

Server architecture refers how the client-server environment will be configured. There are different ways of configuring a client-server system, each with its advantages and disadvantages. The following are options that were considered for the server architecture for LARMAS.

3.4.1. Everything on One Server

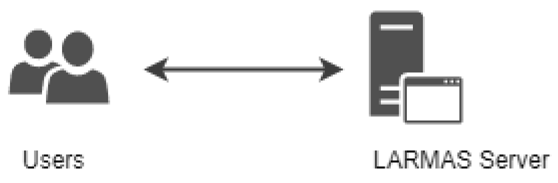


FIGURE 28: SINGLE SERVER SETUP.

Everything is installed on the server. The web server application on which the LARMAS web application will be running on and the database server will all run on the same server. This setup is very simple and quick to set up. It's is suitable for development and testing environments. However, this setup is prone to poor performance since all the server applications will be sharing the same resources (CPU, RAM, Network I/O, etc.). This setup also cannot scale out, therefore, it's scalability is limited by how a number of resources that can be added to the server. There is a single point of failure on the server; if it goes down, the whole system goes down.

3.4.2. Separate Database and Storage Servers

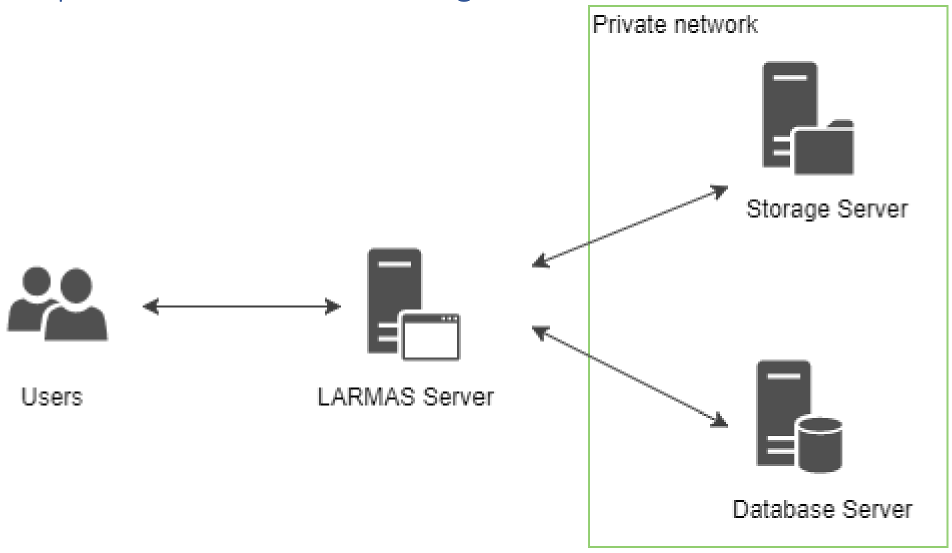


FIGURE 29: SEPARATE DATABASE AND STORAGE SERVER

For this setup, the DBMS and the storage system both have their own dedicated servers. The application and database servers will not fight for the same server resources, therefore, each server can be scaled up separately by adding more resources as needed. The database and storage servers can also be put in a private network which will improve security as these will not be accessible to the public.

This setup is a bit more complex than the single server setup and it is also hard to scale out. There is also a single point of failure on the LARMAS server. If it goes down, the whole system will also go offline.

3.4.3. Reverse Proxy (Load Balancer)

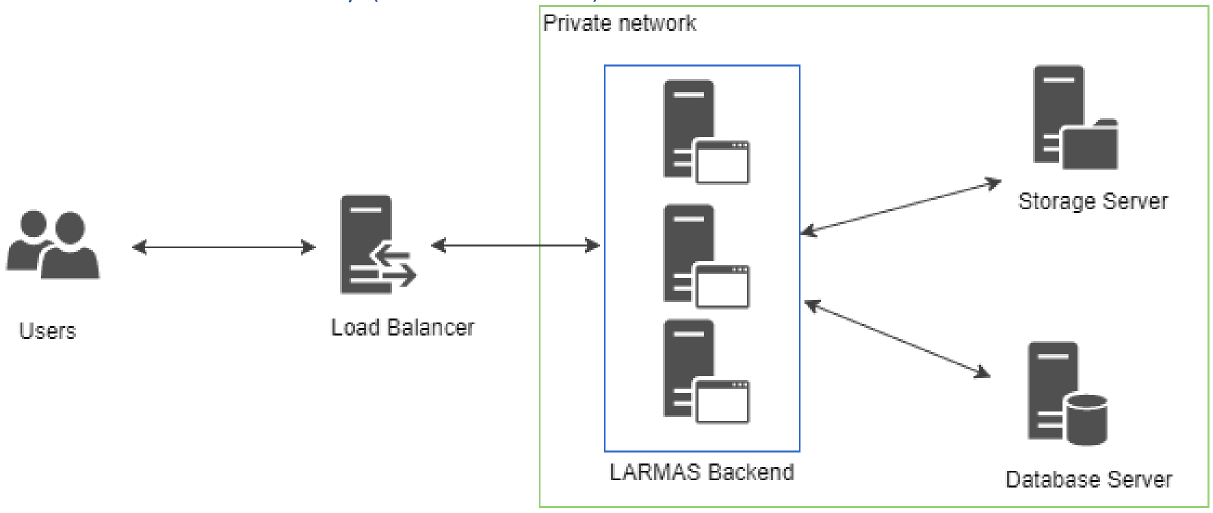


FIGURE 30: REVERSE PROXY SETUP USING A LOAD BALANCER

A reverse proxy is a type of proxy server that acts on behalf of the server to serve resources to the clients. A load balancer is a type of reverse proxy server that can be used in a client-server environment to improve performance, availability and reliability by spreading the workload across multiple application servers [80]. If one of the application server fails, the load balancer can distribute the requests to the other applications servers. Using a load balancer makes it easy to scale out when necessary while each of the application servers can still be scaled up. This setup improves security since the application servers can be put in a private network and DDOS attacks can be prevented by the load balancer by having it monitor and limit client connections. This setup is much more complex and also has a single point of failure at the load balancer server and it can also become a performance bottleneck if it is poorly configured or lacks resources.

3.4.4. Reverse Proxy with Floating IP

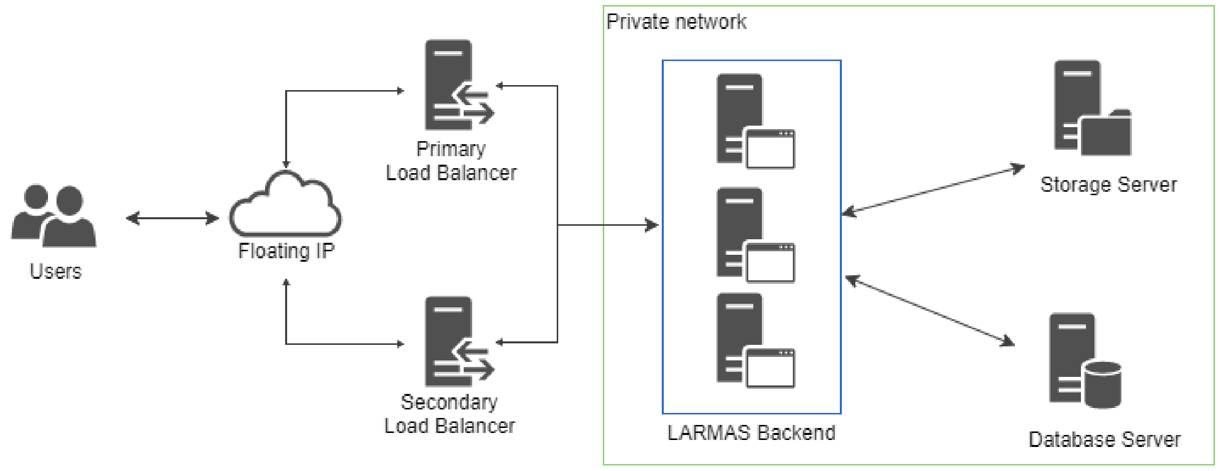


FIGURE 31: A SETUP USING A FLOATING IP TO USE MORE THAN ONE LOAD BALANCER

A floating IP is a publicly accessible static IP address that is used in high-availability server clusters [81]. This can be configured to point at a primary load balancer in LARMAS and if that primary load balancer fails, it can automatically switch to point at a secondary backup load balancer while the primary load balancer is being fixed. Use of floating IP provides a failover mechanism in the server cluster and removes the single point of failure that was at the load balancer in the reverse proxy setup.

3.4.5. LARMAS Server Architecture

Two of the LARMAS non-functional requirements is that it should be highly scalable and be able to handle large workloads. The Reverse Proxy with Floating IP server architecture with LARMAS can be scaled in the following ways:

- **Vertical Scaling**: Each node the LARMAS cluster (application servers, swift proxy, load balancer, database server) can be scaled up by adding more resources to the node to handle an increasing workload. The storage nodes can also be scaled up by increasing the size of the disks they use.
- **Horizontal Scaling**: More application server nodes can be added to the LARMAS backend to increase the system's ability to handle more request. More storage nodes can also be added to the Swift Object storage to increase redundancy and reliability and to also handle more incoming requests from the applications servers.

Figure 32 shows the overall client-server architecture used to implement the LARMAS system.

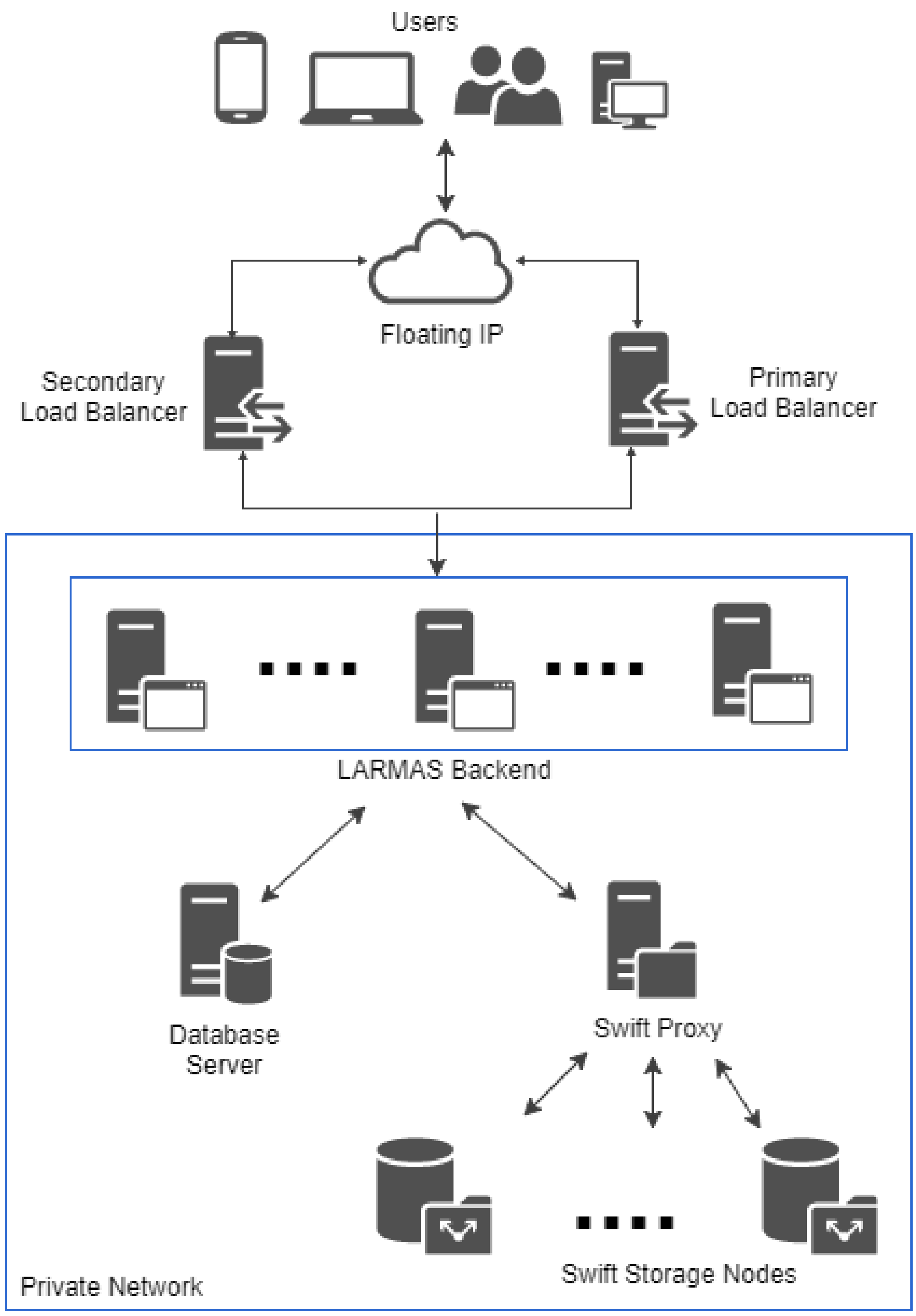


FIGURE 32: SERVER ARCHITECTURE USED FOR LARMAS

4. Tests and Experiments

4.1. Abstract

This chapter describes the test and experiments that were carried out on LARMAS. It starts off by describing the testing environment and the tools used for the tests. It then describes in detail, the three groups of experiments that were carried out.

4.2. Testing Environment

4.2.1. Hardware

LARMAS will be running on Amazon Web Services Elastic Compute Cloud (AWS EC2) instances of varying computing power. These will be referred to as the application servers or nodes. For the testing procedure, five EC2 instance types will be used. These are summarised in Table 11. CPU units refer to the number of CPU cores allocated to an instance. The CPU cores are the cores from Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz [82].

TABLE 11: AWS EC2 INSTANCE TYPES USED FOR TESTING [83].

Instance Type	CPU Units	Memory (GB)	Cost ($/hr)
t2.micro	1	1	0.0116
t2.small	1	2	0.0230
t2.large	2	8	0.0928
t2.xlarge	4	16	0.1856
t2.2xlarge	8	32	0.3712
m4.4xlarge	16	64	0.800

AWS also offers database management services under AWS Relational Database Services (RDS). The RDS databases that will be used in the experiments are MySQL and PostgreSQL. The databases instance types that will be used are shown under Table 12.

TABLE 12: AWS RDS INSTANCE TYPES USED FOR TESTING [84].

Instance Type	CPU Units	Memory (GB)	Cost ($/hr)
t2.medium	2	4	0.068
t2.large	2	8	0.136
m4.xlarge	4	16	0.350
M4.2xlarge	8	32	0.730

4.2.2. Workload Generation

JMeter is a free and open source software designed for load testing functional behaviour and measure performance of web applications (but has since expanded to cover different types of applications). It was developed by Apache Software Foundation in 1998 in Java [85]. JMeter has features that will be used to generate requests with the relevant weightings to simulate the workload that was defined under section 3.2.2.2.

4.2.3. Test Data

The LARMAS database was populated with text from OPUS. OPUS has a collection of translated texts from the internet. Some of these texts were loaded as prompts and translations into the database to be used as testing data. There is a total of 14,125 English, 13,786 Afrikaans and 341 Shona prompts that were loaded from OPUS into the database using a python script (located in the project's home directory under "`./extra/load_propts.py`".) This script parses the XML files downloaded from the OPUS website and inserts the text into LARMAS databases as prompts and translations. Random WAV files were generated to simulate recorded prompts and random strings were generated to represent the annotations. Requests also used random strings and numbers for parameters used in some of the API endpoints.

4.3. Functional Requirements

4.3.1. Unit tests

LARMAS provides an API for client-side applications, therefore, the functional requirements are all features implemented at the API endpoints. To test the API endpoints, unit tests are made for each endpoint. The tests should not only test whether the LARMAS returned the correct responses, but also verify the contents of each response. Bad API calls are also tested to check whether the server responds accordingly. Coverage tests are used to check that every line of code was executed except for the code that handles catastrophic errors.

Django comes with its own Uni Testing framework which was used to run the unit tests. Travis CI was configured to automatically install the server and run the unit tests after every git push to the remote repository. The following command is used to run the tests while in the project home directory:

```
# coverage run --source=. manage.py test -v 2 –noinput
# coverage report -m
```

4.3.2. Distribution of Prompts Experiment

This experiment is to verify whether LARMAS distributes prompts evenly. For file storage, a swift server along with its cluster will be running on an m4.4xlarge EC2 instance. This experiment will be run on a t2.large EC2 instance and it will be split into two parts.

Part 1

The functional requirement 3.2.1(d) specifies that the system must distribute prompts evenly. To test this functionality, a python script was created to:

1. Create 1500 users with first language as Shona (with a total of 326 prompts)
2. Each user requests for prompts (each gets 50 prompts at a time)
3. Each user uploads annotations (also referred to as recordings in these experiments) for each prompt they received. The total number of annotation contributions will be 1500 x 50 = 75,000.
4. After every user has uploaded all their prompts, the script will query the database for all the Shona prompts stored on the database and retrieve the ***number_of_recordings*** field from each prompt and export the results to a CSV file for the data to be analysed at a later stage.

After this script has finished running, the collected data will be analysed to verify whether the prompts were distributed evenly. This test will be run using the server architecture described in 3.4.1 where everything will be on one server. Because database performance is not being tested in this experiment, the development database, SQLite, will be used for this test.

Part 2

Part 2 of this experiment is to test the hypothesis that the distribution of prompts will remain normal after more prompts are added to LARMAS during production. After Part 1 of the experiment has run, 100 new Shona prompts will be added to the system and 6000 new users will be created to collect an additional 300,000 new recordings. The distribution of this part 2 of the experiment will be compared to the distribution from Part 1.

4.4. Response Times

This experiment is to determine the DBMS with the best performance. It will test and compare the response times for the API endpoints and the performance of the different DBMS. The DBMSs that will be tested are SQLite, MySQL and PostgreSQL. The server architecture described under section 3.4.2 will be used for the MySQL and PostgreSQL instances and the server architecture described under section 3.4.1 will be used for the SQLite database since the SQLite DBMS is a lightweight database designed for local usage, therefore, it does not have a server associated with it in the same way MySQL and PostgreSQL do [86].

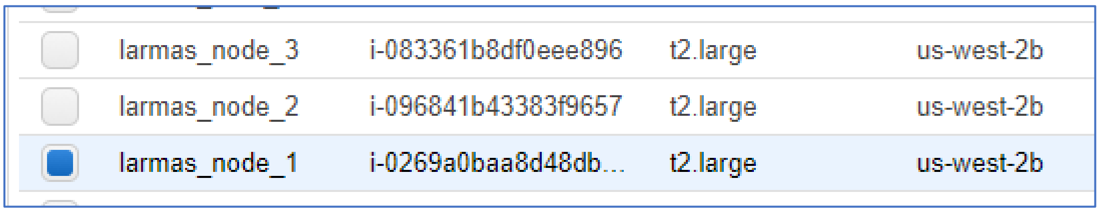
The endpoints used in this simulation were weighted to emulate the first 5 years workload described in Table 7 (under section 3.2.2.2 Workload Modelling) using the JMeter **Throughput Controller**. The actual endpoints used for this test are shown below. Explanations of these endpoints are found in Table 9.

- **POST** - `/v1/user/api-token-auth/`
- **GET** - `/v1/prompts/retrieve/`
- **POST** - `/v1/annotations/upload/`
- **GET** - `/v1/annotations/`
- **POST** - `/v1/prompt_translations/upload/`
- **GET** - `/v1/prompt_translations/parallel/ENG-ZA/SHO-ZW/`

JMeter will be used to simulate random concurrent users who will log in, browse prompts, annotations and translations and contribute annotations and translations. Figure 34 is a screenshot of the JMeter test bench setup used for this experiment. There will be two parts to this test. The first part of this experiment will simulate a normal workload where requests to the server will be sent to the server at a rate of 25,000 requests per hour. The second part of the experiment will simulate a workload of 50,000 requests an hour to test how the databases perform under peak and heavy workloads.

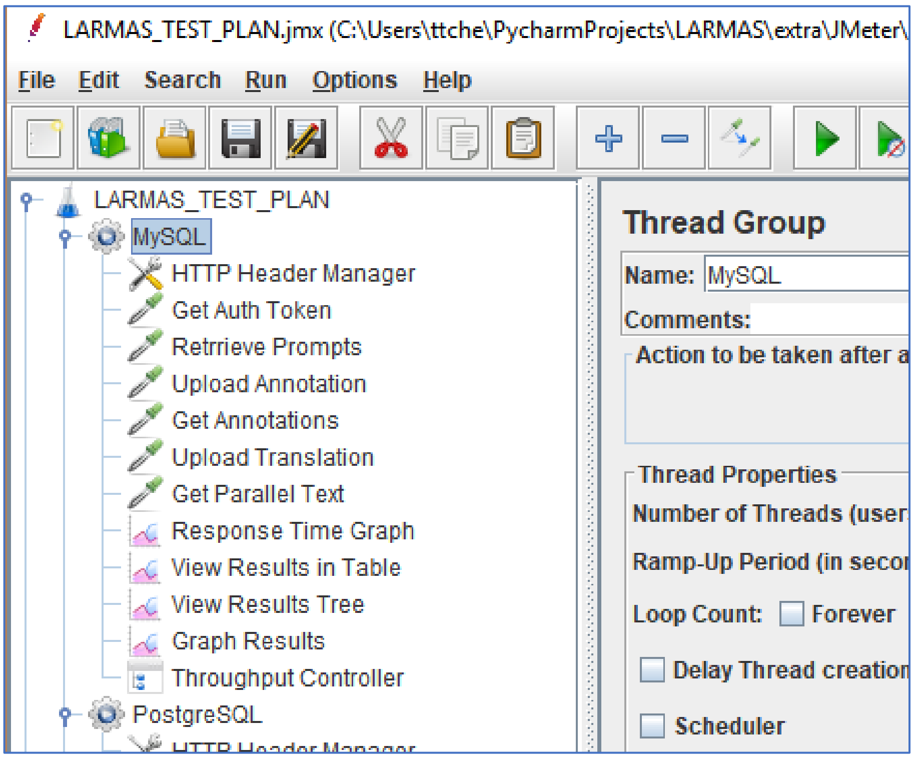
An Amazon Instance Image (AMI) with LARMAS installed on it was created. An AMI is a snapshot of an EC2 instance that can be used to create clones of an instance. The application servers will each run on EC2 t2.large instances (Figure 33) and the MySQL and PostgreSQL DBMS will each run on RDS t2.large instances. The SQLite DBMS will be on the same machine as the LARMAS server therefore both LARMAS and SQLite will be sharing the same resources. All three databases will be loaded with the same data at the beginning of the experiment and all three will be tested at the same time so that they experience the same network conditions. All application servers for this test will have their own Swift object storage server and cluster running on an m4.4xlarge EC2 instance.

The data centre where these instances are located in Oregon, USA therefore there will be a considerable latency since the requests will be coming from a local machine running in Cape Town, South Africa. The network latency will be calculated and subtracted from the observed response times to calculate the server's response times.



☐	larmas_node_3	i-083361b8df0eee896	t2.large	us-west-2b
☐	larmas_node_2	i-096841b43383f9657	t2.large	us-west-2b
🟦	larmas_node_1	i-0269a0baa8d48db...	t2.large	us-west-2b

FIGURE 33: SCREENSHOT OF THE EC2 INSTANCES USED FOR TESTING.



LARMAS_TEST_PLAN.jmx (C:\Users\ttche\PycharmProjects\LARMAS\extra\JMeter\

```
LARMAS_TEST_PLAN
  MySQL
    HTTP Header Manager
    Get Auth Token
    Retrieve Prompts
    Upload Annotation
    Get Annotations
    Upload Translation
    Get Parallel Text
    Response Time Graph
    View Results in Table
    View Results Tree
    Graph Results
    Throughput Controller
  PostgreSQL
    HTTP Header Manager
```

Thread Group

Name: MySQL

Comments:

Action to be taken after a

Thread Properties

Number of Threads (user

Ramp-Up Period (in secor

Loop Count: ☐ Forever

☐ Delay Thread creation

☐ Scheduler

FIGURE 34: SCREENSHOT OF THE JMETER TEST BENCH FOR THE MYSQL SERVER

4.5. Scalability

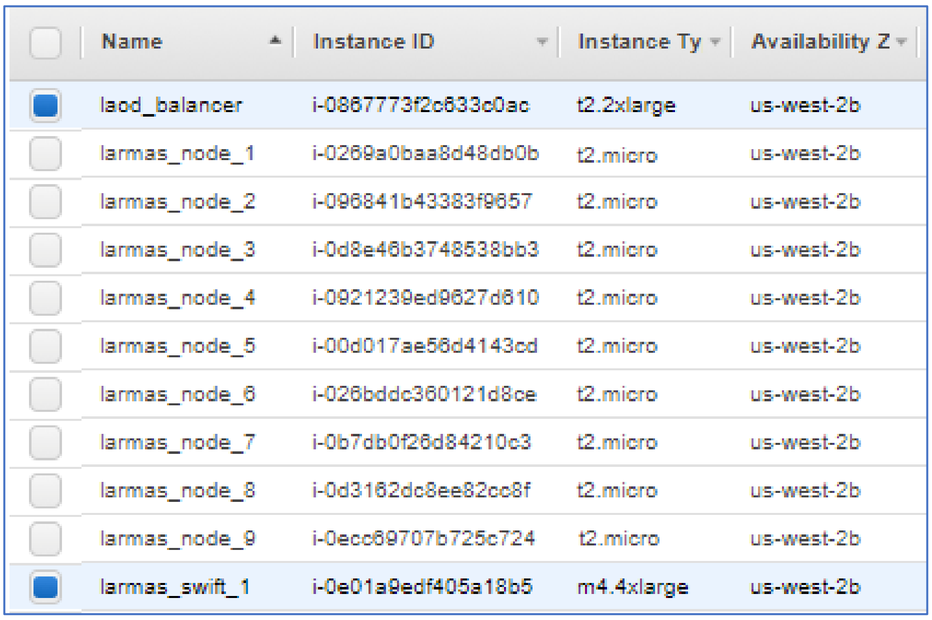
This set of experiments will test the scalability of LARMAS. There are two different ways of scaling a system, vertical scaling where more resources are added to a server, and horizontal scaling where more processing nodes are added to the system.

4.5.1. Vertical Scaling (scaling up)

This experiment will test how well LARMAS scales up. The server architecture described under section 3.4.2 was used where the database server is hosted on a separate server. A PostgreSQL database will be used for all tests due to its better performance over MySQL as shown in section 5.3 and SQLite cannot be used in a distributed environment. The RDS instance type will be m4.2xlarge, an instance type with a lot of resources to make sure that the database server would not be a bottleneck in the experiment. The application server was hosted on EC2 and scaling was done by changing the EC2 instance type. Four instance types were used for this test. The number of CPU units is increased from 1 to 16 while the RAM also increased from 2 to 64 GB. The application instance will be scaled up when the current setup has reached its maximum workload, a point at which requests will be rejected or experience timeout.

4.5.2. Horizontal Scaling (scaling out)

This experiment will test how well LARMAS scales out. The server architecture described under section 3.4.3 will be used for this test. Each application server node will be running on a t2.micro size EC2 instance and the database, which will be shared by all nodes, will be running on an m4.2xlarge RDS instance. The HAProxy load balancer will be running on a t2.2xlarge EC2 instance and will be using the default round-robin algorithm to distribute the workload to the application servers. The number of nodes in the server architecture will be increased from 1 to 9 nodes and a node is only added when the current setup reaches its maximum workload, a point at which requests will be rejected or experience timeout. All processing nodes will be using the same Swift storage cluster, therefore, a large m4.4xlarge instance was used for the Swift server.



	Name	Instance ID	Instance Ty	Availability Z
■	laod_balancer	i-0867773f2c633c0ac	t2.2xlarge	us-west-2b
☐	larmas_node_1	i-0269a0baa8d48db0b	t2.micro	us-west-2b
☐	larmas_node_2	i-096841b43383f9657	t2.micro	us-west-2b
☐	larmas_node_3	i-0d8e46b3748538bb3	t2.micro	us-west-2b
☐	larmas_node_4	i-0921239ed9627d610	t2.micro	us-west-2b
☐	larmas_node_5	i-00d017ae56d4143cd	t2.micro	us-west-2b
☐	larmas_node_6	i-026bddc360121d8ce	t2.micro	us-west-2b
☐	larmas_node_7	i-0b7db0f26d84210c3	t2.micro	us-west-2b
☐	larmas_node_8	i-0d3162dc8ee82cc8f	t2.micro	us-west-2b
☐	larmas_node_9	i-0ecc69707b725c724	t2.micro	us-west-2b
■	larmas_swift_1	i-0e01a9edf405a18b5	m4.4xlarge	us-west-2b

FIGURE 35: SCREENSHOT OF THE LARMAS SETUP FOR VERTICAL SCALING TESTING.

4.5.3. Load balancing algorithm

This experiment is to determine which load balancing algorithm has the best performance for use in LARMAS. The server architecture described under section 3.4.3 will be used for this test. 5 application server nodes running on EC2 t2.micro instances will be used for this test together with a swift storage server running on a t2.2xlarge EC2 instance and PostgreSQL as the database running on RDS m4.4xlarge instance.

HAProxy has three load balancing algorithms. These are round robin (denoted as **round-robin**), least number of connections (denoted as **leastconn**) and IP Hash (denoted as **source**). The same request rate of 120,000 requests per hour was used to simulate the workload for all three algorithms. HAProxy allows for load balancing algorithms to be changed and activated by using the “service reload” command which ensures that there is no download time for the server. The CPU usage of the nodes together with the response times will be monitored to assess how they react to the different load balancing algorithms.

5. Results and Discussions

5.1. Abstract

This chapter discusses the results of the tests and experiments carried out in Chapter 4. Results from each experiment are presented in the form of graphs, screenshots and tables, each with comments explaining what they represent. Discussions and conclusions about each experiment.

5.2. Functional Requirements

5.2.1. Unit Tests

Table 13 is a breakdown of the unit tests in LARMAS. The "Code Coverage" column indicates the percentage of the application code (not code from the tests) that was executed during the tests. The 8-9% of the code that is reported as "not covered" consists of exception handlers for catastrophic errors like HTTP 500 Server Errors which are not supposed to be executed under normal situations.

Table 13: Summary of the unit tests in LARMAS

Django App	Number of Tests (Total=74)	Code Coverage (%)	Test Group	What is tested.
Annotations	13	92	Annotation Details	Endpoints that retrieve annotations and download links to stored recordings.
			Annotation Upload	Endpoints that upload annotations for registered and one-time users.
Prompts	20	91	Prompt Details	Endpoints that browse and modify prompts.
			Prompt Distribution	Endpoints that distribute prompts and manage prompt rejection.
Translations	14	91	Translation Details	Endpoints that retrieve and allow for browsing of translations and retrieval of parallel text.
			Translation Upload	Endpoints for uploading of translations.
User	27	92	User Details	Endpoints to browsing and updating user profiles.
			User Registration	Endpoints to register new users to LARMAS.
			User Authentication	Endpoints to authenticate users and retrieve authentication tokens.

5.2.2. Prompt Distribution Experiments

This section discusses the results of the experiment described in section 4.3.2. A total of **75,080** recordings from **341** Shona prompts were uploaded to LARMAS for this experiment. Figure 36 shows the distribution of the number of recordings. It is evident that the distribution is shaped like a bell curve, a feature of a normal distribution [87], with a mean of **220** and a standard deviation of **17.8** recordings per prompt. Figure 37 shows the distribution of prompts after Part 2 of this experiment was carried out were an extra **305,407** recordings were collected. This distribution is also bell-curved meaning that it is a normal distribution; with a mean of **862** and standard deviation of **18.4** recordings per prompt. This experiment has shown that the method implemented to evenly distribute prompts to users works very well even if more prompts are added to the system after it has been released.

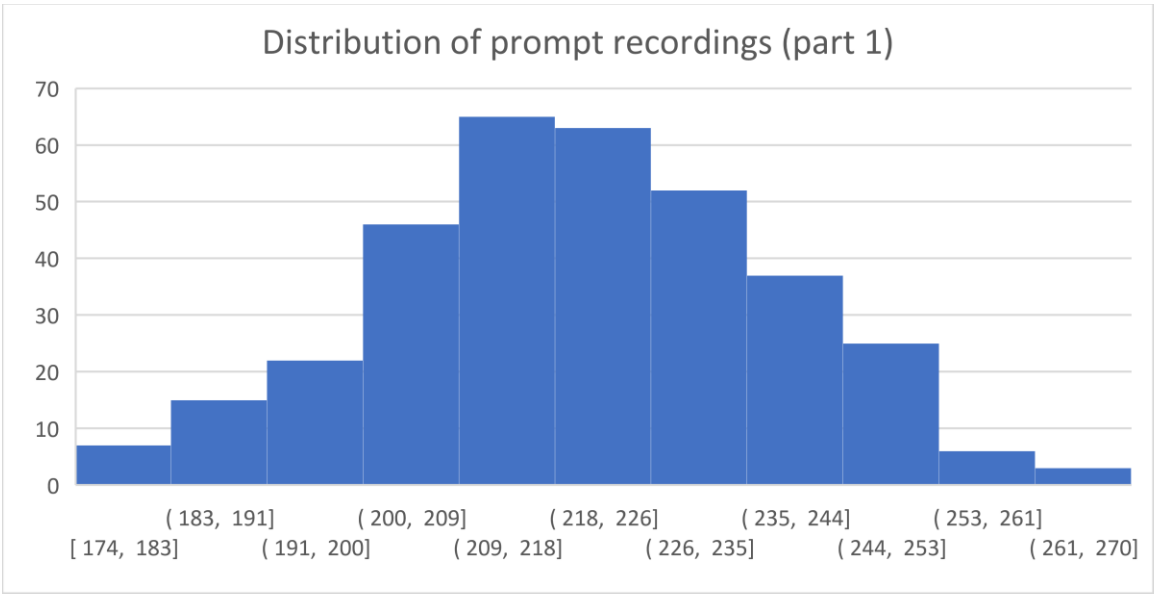


FIGURE 36: DISTRIBUTION OF THE NUMBER OF RECORDINGS PER PROMPT FOR PART 1 OF THE EXPERIMENT

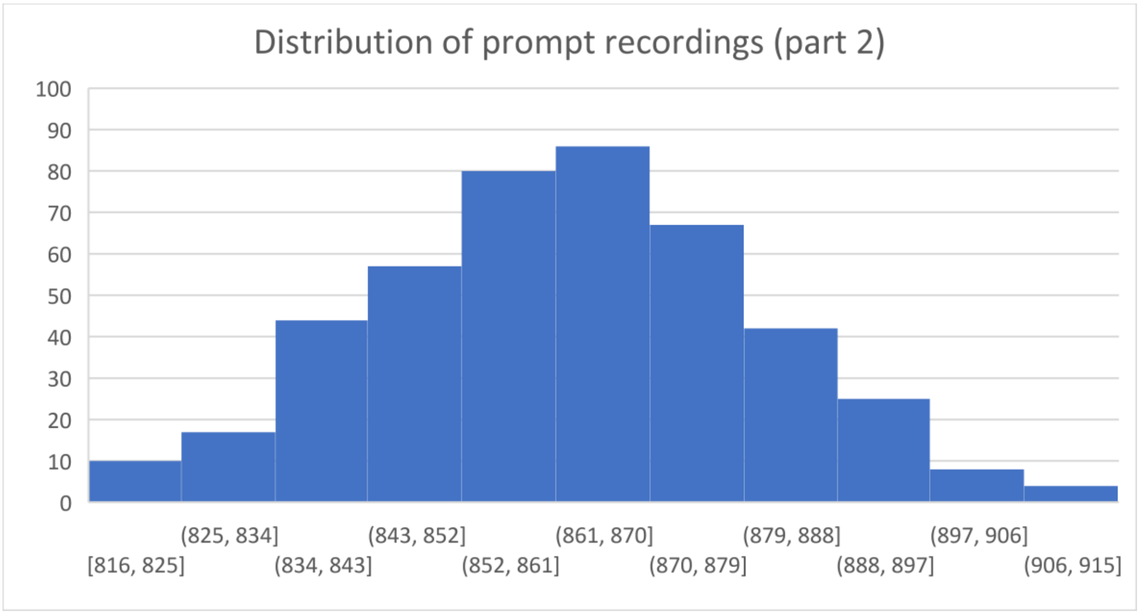


FIGURE 37: DISTRIBUTION OF THE NUMBER OF RECORDINGS PER PROMPT FOR PART 2 OF THE EXPERIMENT

5.3. Response Times

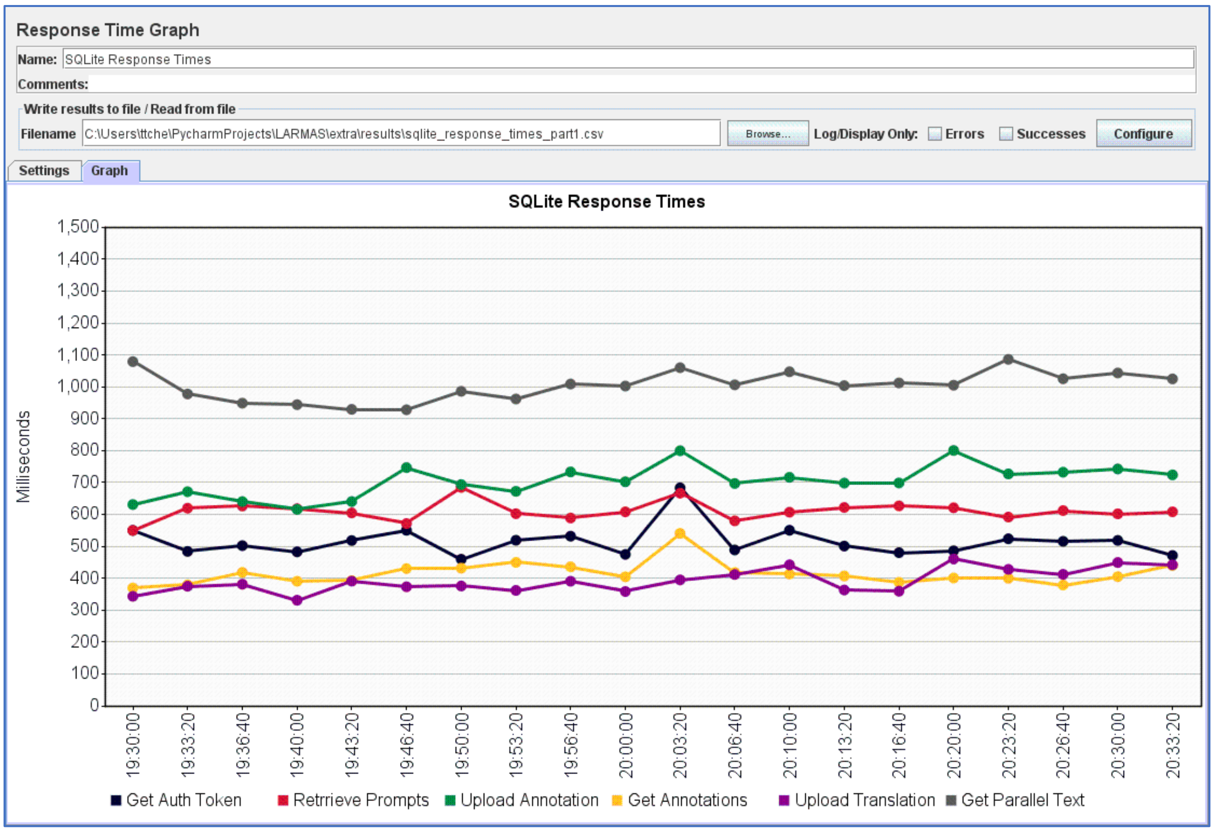
PING is a command line program that is used to verify if a machine can communicate with another over an IP network. It works by sending an ICMP echo requests to the target machine and waits for a response in the form of ICMP echo replies. The program then uses these requests and responses to report on packet losses, average roundtrip times (RTT) and standard deviation [88]. Because PING packets are very small (64 bytes) and require very little processing, it can be used to predict the network latency between two machines on an IP network. PING was used to determine the network latency between the origin of the requests (UCT, Cape Town) and the location of the 3 LARMAS servers used in this test (Oregon, USA). The results are shown in Figure 38.

Because all three servers are running in the same AWS region, the network latency was very similar (337ms). To calculate the server's response time (time taken by the server to process the request), the RTT from the PING commands will be subtracted from the response times calculated by JMeter (which reports the total latency from sending a request and receiving a response from the server.)

```
tbash@tino1b2be-PC:/mnt/e/VM$ ping -c 6 52.38.82.120
PING 52.38.82.120 (52.38.82.120) 56(84) bytes of data.
64 bytes from 52.38.82.120: icmp_seq=1 ttl=31 time=337 ms
64 bytes from 52.38.82.120: icmp_seq=2 ttl=31 time=337 ms
64 bytes from 52.38.82.120: icmp_seq=3 ttl=31 time=337 ms
64 bytes from 52.38.82.120: icmp_seq=4 ttl=31 time=337 ms
64 bytes from 52.38.82.120: icmp_seq=5 ttl=31 time=337 ms
64 bytes from 52.38.82.120: icmp_seq=6 ttl=31 time=337 ms
--- 52.38.82.120 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5002ms
rtt min/avg/max/mdev = 387.282/387.584/387.800/0.171 ms
tbash@tino1b2be-PC:/mnt/e/VM$ ping -c 6 35.166.33.8
PING 35.166.33.8 (35.166.33.8) 56(84) bytes of data.
64 bytes from 35.166.33.8: icmp_seq=1 ttl=32 time=337 ms
64 bytes from 35.166.33.8: icmp_seq=2 ttl=32 time=404 ms
64 bytes from 35.166.33.8: icmp_seq=3 ttl=32 time=336 ms
64 bytes from 35.166.33.8: icmp_seq=4 ttl=32 time=335 ms
64 bytes from 35.166.33.8: icmp_seq=5 ttl=32 time=335 ms
64 bytes from 35.166.33.8: icmp_seq=6 ttl=32 time=335 ms
--- 35.166.33.8 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5002ms
rtt min/avg/max/mdev = 335.542/3477.682/403.907/25.599 ms
tbash@tino1b2be-PC:/mnt/e/VM$ ping -c 6 52.36.28.196
PING 52.36.28.196 (52.36.28.196) 56(84) bytes of data.
64 bytes from 52.36.28.196: icmp_seq=1 ttl=32 time=339 ms
64 bytes from 52.36.28.196: icmp_seq=2 ttl=32 time=336 ms
64 bytes from 52.36.28.196: icmp_seq=3 ttl=32 time=337 ms
64 bytes from 52.36.28.196: icmp_seq=4 ttl=32 time=336 ms
64 bytes from 52.36.28.196: icmp_seq=5 ttl=32 time=336 ms
64 bytes from 52.36.28.196: icmp_seq=6 ttl=32 time=336 ms
--- 52.36.28.196 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5003ms
rtt min/avg/max/mdev = 336.508/337.219/339.144/1.034 ms
tbash@tino1b2be-PC:/mnt/e/VM$
```

FIGURE 38: RESULTS FROM THE PING TEST.

The figures on the following 3 pages show how the response times for the different API endpoints for each DBMS varied with time over the duration of the experiment. This is followed by Table 14 which summarises the response times for all three servers for both parts of the experiments and Table 15 which shows the percentage of requests which had a server response time of less than 200ms. After this are graphs (Figure 45 and Figure 46) showing the CPU usage for both the application and database servers.


Response Time Graph

Name: SQLite Response Times

Comments:

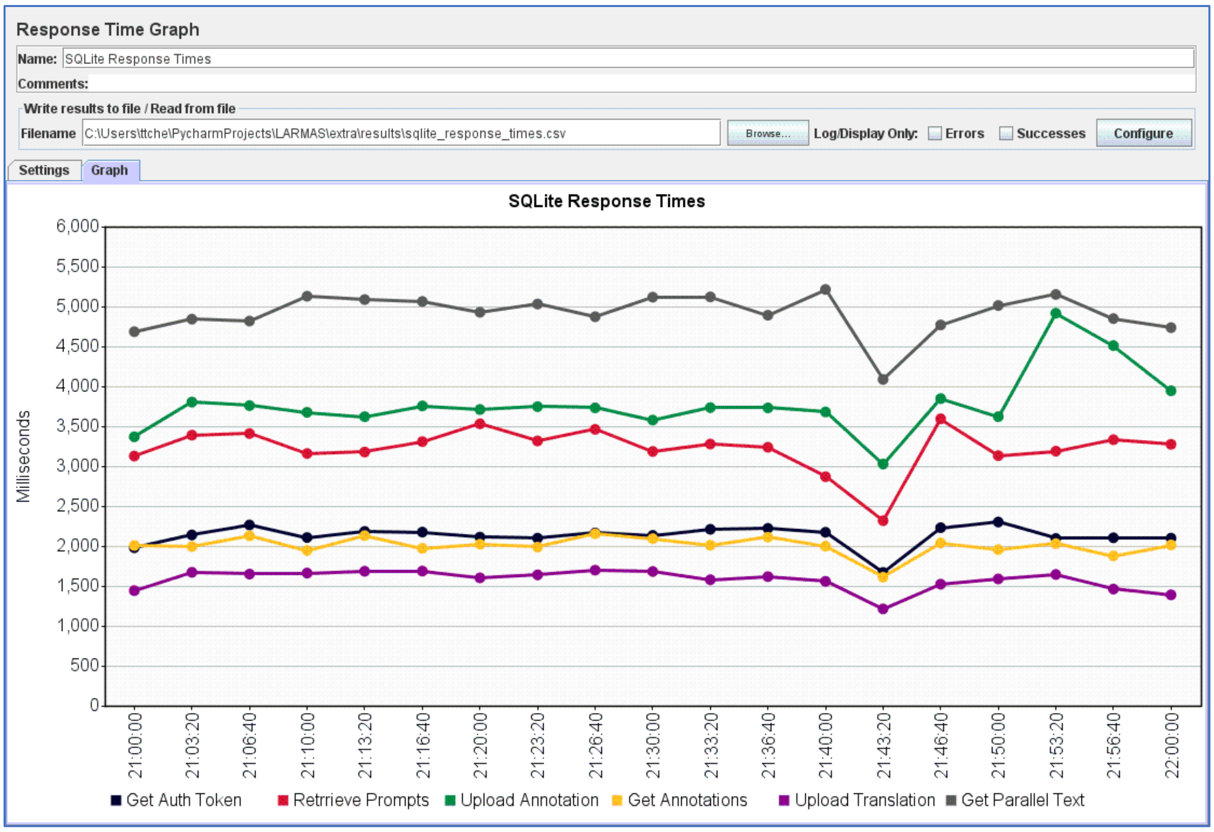
Write results to file / Read from file

Filename C:\Users\ttche\PycharmProjects\LARMAS\extra\results\sqlite_response_times_part1.csv

Browse... Log/Display Only: ☐ Errors ☐ Successes Configure

Settings Graph

FIGURE 39: RESPONSE TIME GRAPH FOR NODE 1 PART 1 (SQLITE)


Response Time Graph

Name: SQLite Response Times

Comments:

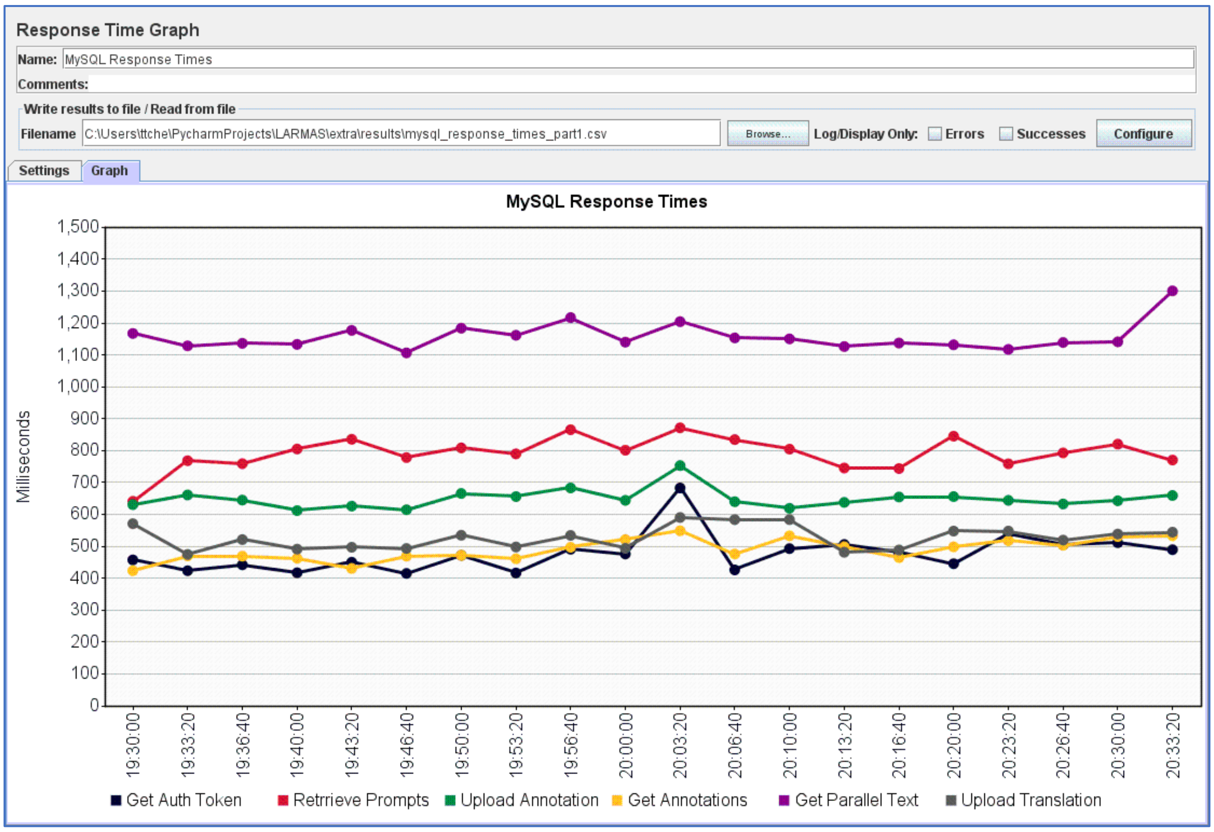
Write results to file / Read from file

Filename C:\Users\ttche\PycharmProjects\LARMAS\extra\results\sqlite_response_times.csv

Browse... Log/Display Only: ☐ Errors ☐ Successes Configure

Settings Graph

FIGURE 40: RESPONSE TIME GRAPH FOR NODE 1 PART 2 (SQLITE)



Response Time Graph

Name: MySQL Response Times

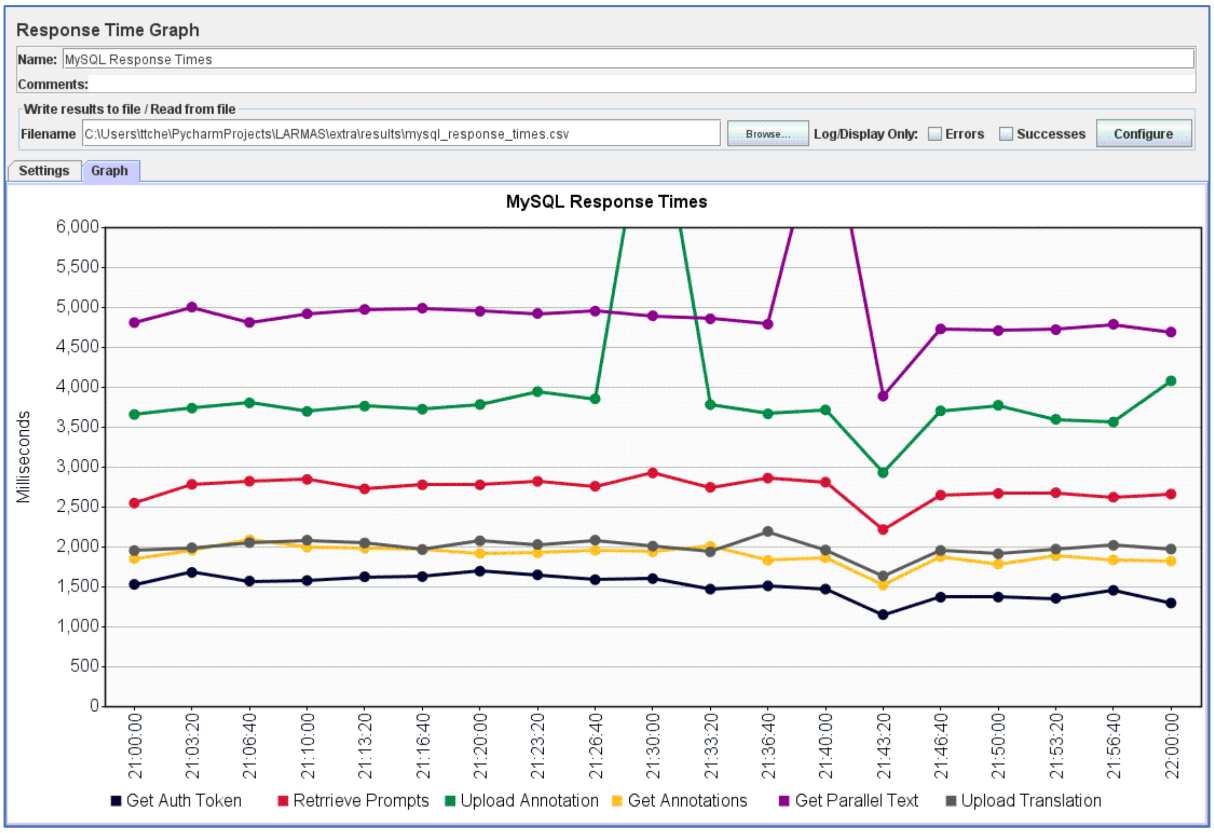
Comments:

Write results to file / Read from file

Filename C:\Users\ttche\PycharmProjects\LARMAS\extra\results\mysql_response_times_part1.csv

Log/Display Only: ☐ Errors ☐ Successes

Figure 41: Response time graph for Node 1 Part 1 (MySQL)



Response Time Graph

Name: MySQL Response Times

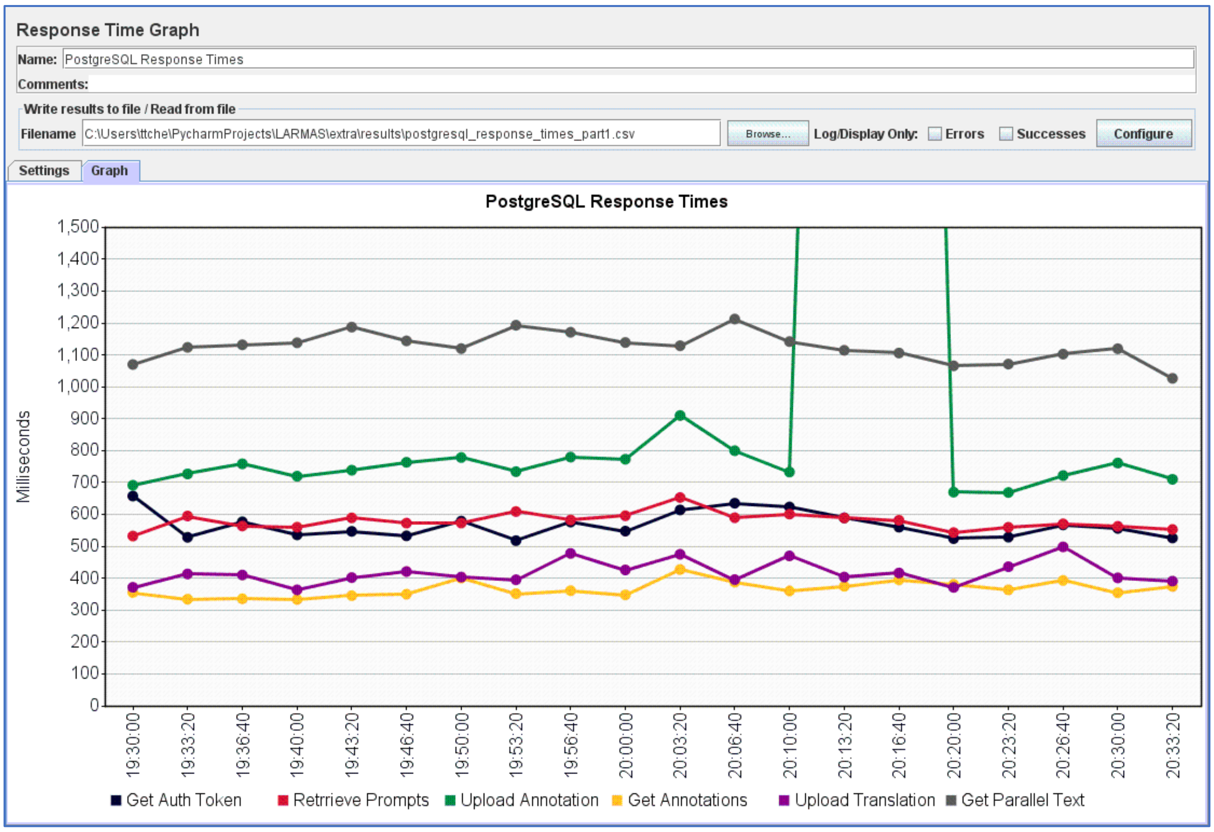
Comments:

Write results to file / Read from file

Filename C:\Users\ttche\PycharmProjects\LARMAS\extra\results\mysql_response_times.csv

Log/Display Only: ☐ Errors ☐ Successes

Figure 42: Response time graph for Node 1 Part 2 (MySQL)


Response Time Graph

Name: PostgreSQL Response Times

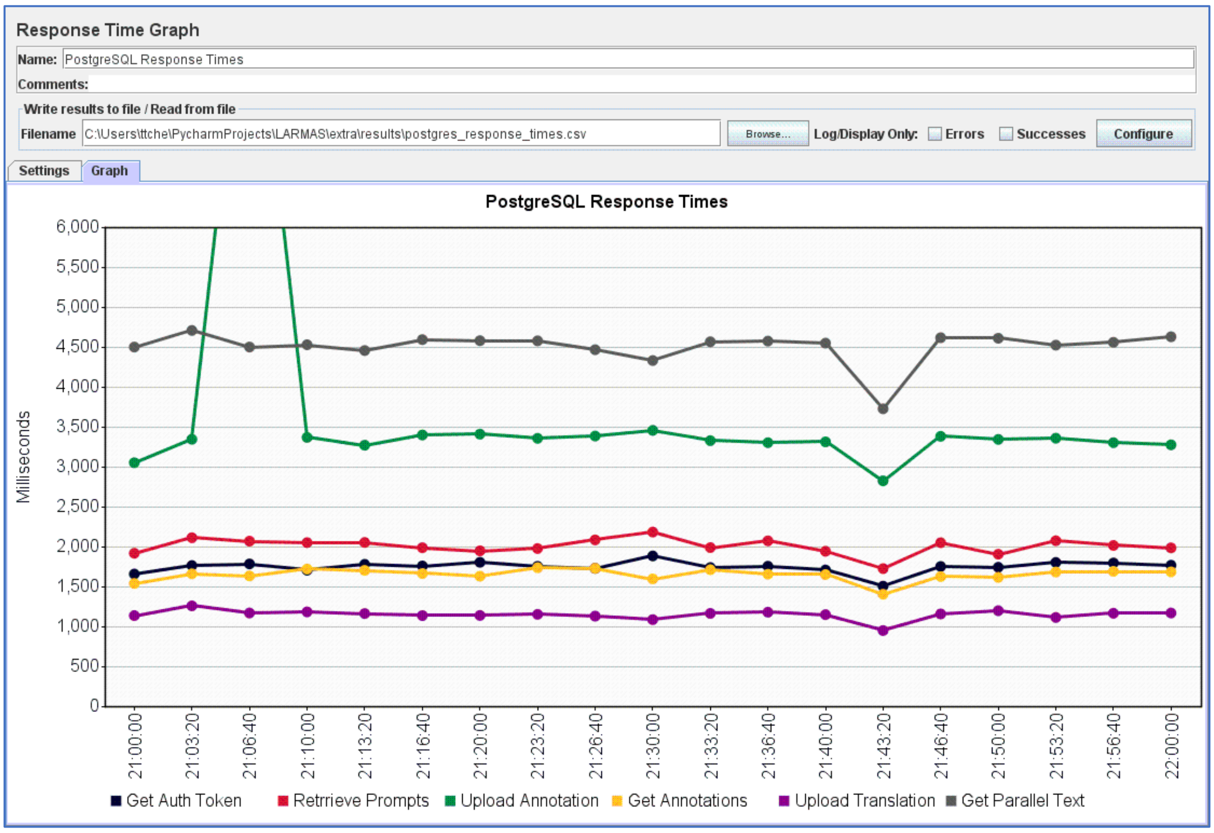
Comments:

Write results to file / Read from file

Filename C:\Users\ttche\PycharmProjects\LARMAS\extra\results\postgresql_response_times_part1.csv

Log/Display Only: ☐ Errors ☐ Successes

FIGURE 43: RESPONSE TIME GRAPH FOR NODE 1 PART 1 (POSTRESQL)


Response Time Graph

Name: PostgreSQL Response Times

Comments:

Write results to file / Read from file

Filename C:\Users\ttche\PycharmProjects\LARMAS\extra\results\postgres_response_times.csv

Log/Display Only: ☐ Errors ☐ Successes

FIGURE 44: RESPONSE TIME GRAPH FOR NODE 1 PART 2 (POSTRESQL)

TABLE 14: TABLE COMPARING THE SERVER RESPONSE TIMES FOR EACH ENDPOINT ON THE DIFFERENT DBMS FOR PART 1

Endpoint	Mean Server Response Times					
	SQLite		MySQL		PostgreSQL	
	Part 1	Part 2	Part 1	Part 2	Part 1	Part 2
Auth Token	173	1811	139	1173	226	1426
Retrieve Prompts	276	2911	459	2409	245	1691
Upload Annotation	368	3320	310	3353	414	2998
Get Annotations	104	1685	154	1612	28	1327
Upload Translation	57	1253	185	1674	130	818
Get Parallel Text	636	4552	793	4435	772	4172
Combined Average	***264***	***2588***	***340***	***2435***	***294***	***2072***

TABLE 15: PERCENTAGE OF REQUESTS WITH A SERVER RESPONSE TIME UNDER 200MS

DBMS	Part 1 (%)	Part 2 (%)
SQLite	54.9	0.84
MySQL	42.0	0.36
PostgreSQL	57.0	0.28

TABLE 16: PERCENTAGE OF REQUESTS WITH A SERVER RESPONSE TIME UNDER 1 SECOND

DBMS	Part 1 (%)	Part 2 (%)
SQLite	98.3	16.9
MySQL	96.7	16.9
PostgreSQL	97.7	17.9

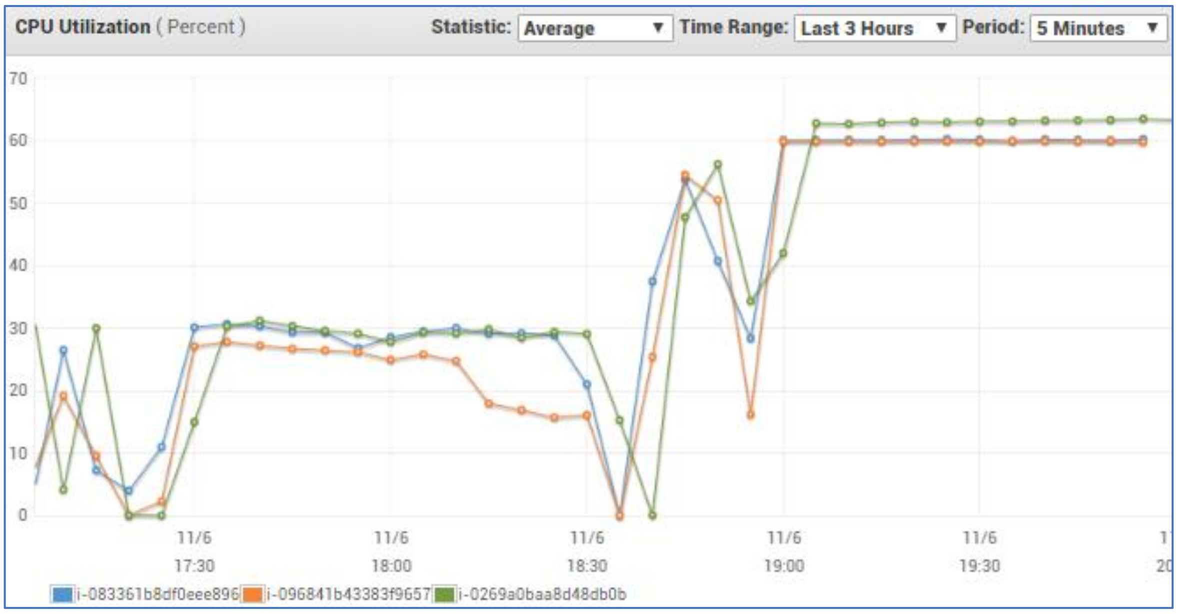


FIGURE 45: CPU USAGE FOR THE APPLICATION SERVERS INSTANCES RUNNING ON EC2 (ORANGE=POSTGRESQL, GREEN=SQLITE, BLUE=MYSQL)

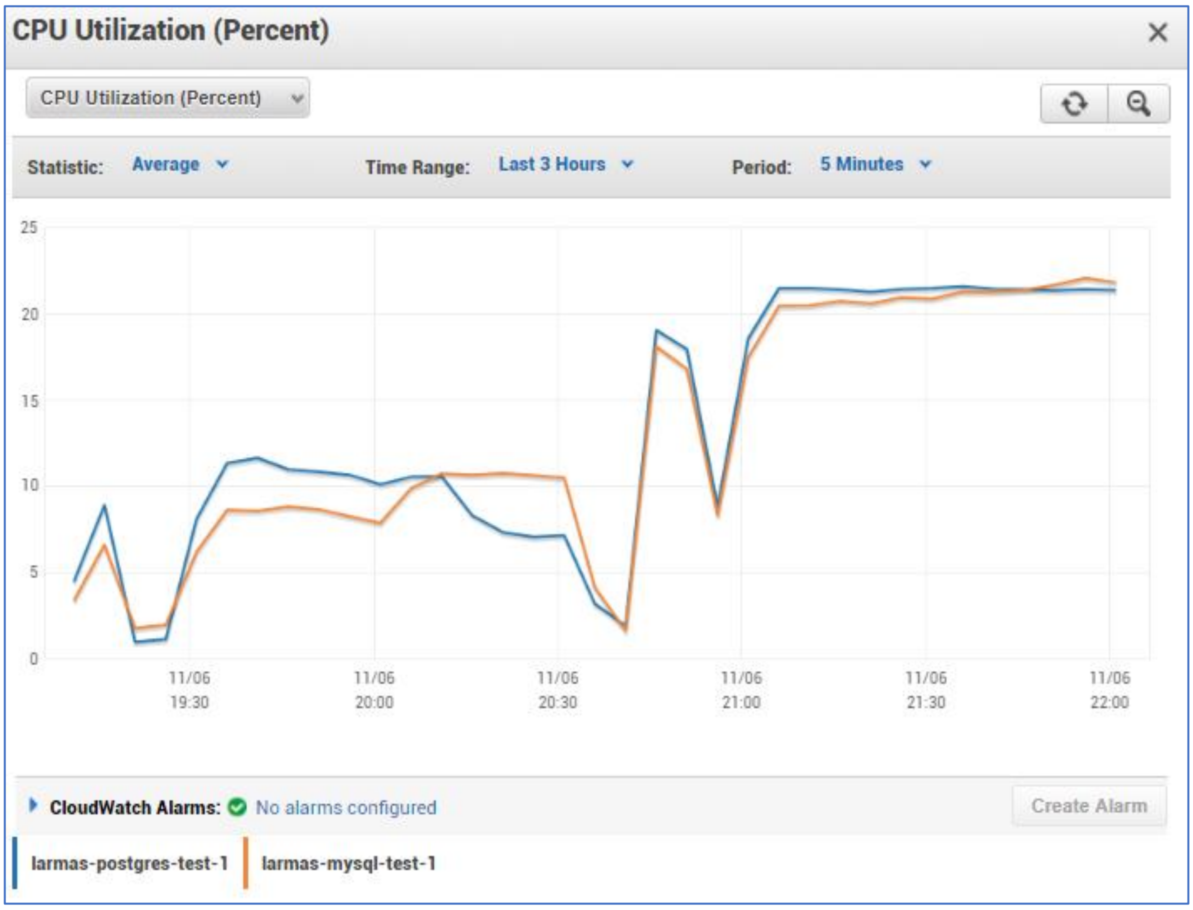


FIGURE 46: CPU USAGE FOR THE RDS INSTANCES DURING BOTH PARTS OF THE EXPERIMENT. (BLUE=POSTGRES, ORANGE=MYSQL)

Part 1 of the experiment ran for 1 hour and 5 minutes generating 28,500 requests for each server (just under the peak load discussed under section 3.2.2.2 Workload Modelling). Both the EC2 and RDS instances had enough resources to handle this workload. This is evident with the low CPU usage fluctuating at 30% for the application servers and 10% for the database servers shown in Figure 45, 17:30 to 18:30, and Figure 46 19:30 to 20:30, (The RDS instances were using CAT and EC2 instances were using UTC time zones hence the time differences.)

Part 2 ran for 1 hour 4 minutes generating 52,400 requests for each server. This workload put a strain on the server's resources as shown in Figure 45, 19:00 to 20:00, and Figure 46 21:00 to 22:00 which show a constant CPU usage of 60% for the application servers and 21% for the database servers. It is to be noted that the CPU usage for the application server using SQLite was higher than the other two instances using MySQL and PostgreSQL. This is expected because this application server had to share resources for SQLite database transactions, unlike the other two servers which had a separate database server.

All servers did not meet the requirement specified in section 3.2.2.1 of having 95% of the requests being processed under 200ms. PostgreSQL performed the best in this regard having 57% of its requests being processed under 200ms, this was followed by SQLite not far behind at 54.9%. MySQL had the worst performance with 42% of its requests being processed in under 200ms. However, all three databases had a more than 96 % of requests processed under 1 second, the maximum time a user's concentration and flow of thought remains uninterrupted. For part 2 of the experiment, all databases performed very poorly, as expected, having less than 1% of requests processed under 200ms and less than 18% of requests processed under 1 second. Both MySQL and PostgreSQL have big spikes that could be due to a networking errors between the storage server and the application servers.

Surprisingly SQLite performed very well for the first part of the experiment. This was not expected because SQLite documentation does not recommend using SQLite in client-server systems with a high concurrent number of users and large workloads [41] but SQLite had the lowest combined average response time for part 1 of the experiment. However, it had the highest combined average response time for part 2 of the experiment. This could be because part 2 had a much larger workload and concurrent users which it does not perform well with.

This experiment also highlights the different complexities of each endpoint. The Get Parallel endpoint consistently has the highest average response time in all tests. This is expected because this endpoint performs a complex database query on a large dataset (14,113 at the beginning of the tests) which checks two fields within the prompt translation table (the language of the original prompt and language of the translation). The actual query statement used in this test is shown in Figure 47.

```sql
SELECT "translations_prompttranslation"."id",

       "translations_prompttranslation"."original_prompt_id",

       "translations_prompttranslation"."text",

       "translations_prompttranslation"."verified",

       "translations_prompttranslation"."language_id",

       "translations_prompttranslation"."user_id",

       "translations_prompttranslation"."date"

FROM "translations_prompttranslation"

INNER JOIN "prompts_prompt" ON ("translations_prompttranslation"."original_prompt_id"

         = "prompts_prompt"."id")

INNER JOIN "user_language" ON ("prompts_prompt"."language_id" = "user_language"."id")

INNER JOIN "user_language" T4 ON ("translations_prompttranslation"."language_id" = T4."id")

WHERE ("user_language"."code" = ENG-ZA AND T4."code" = SHO-ZW)
```

FIGURE 47: SQL STATEMENT USED TO GET PARALLEL TEXT.

Two important observations were made during this experiment. The first one is that it the results show that the databases perform differently for each endpoint. For example, MySQL has similar response times for the 3 endpoints (Get Auth Token, Get Annotations and Upload Translation). The graphs for these endpoints continuously cross each other throughout the first part of the experiment but the response times for the same endpoints differ much more in the other databases. The PostgreSQL database appears to handle uploading translations faster than MySQL but MySQL can upload annotations faster than PostgreSQL. This information can be used to decide which database to use depending on the type of workload LARMAS would work with in case it changes in the future. In this case, if the workload comprises of mostly collection of translations then it would be better to use the PostgreSQL database since it handles uploading of translations faster.

The second observation was that the application servers do not use a lot of RAM. During these tests, RAM usage never went above 10% of the allocated RAM. Unfortunately, AWS EC2 console does not have an option to monitor the RAM usage of an EC2 instance, therefore, RAM had to be manually monitored using the "**top**" command in the Bash terminal. This means that when scaling up, CPU units would the major limiting factor and not RAM.

5.4. Scalability

5.4.1. Vertical Scaling

This section discusses the results from the vertical scaling experiment. Figure 48 shows how the maximum workload the server could handle before it starts rejecting requests varies as the number of CPUs is increased. Although only up to 16 CPU units were added, it looks like the server was already experience diminishing returns. To illustrate this, a projection was made based on the observed behaviour and it looks like the graph would have plateaued after adding 2^{10} CPU units. The largest available instance type on AWS is x1.32xlarge with 128 CPUs. According to the projections, this instance would only support a LARMAS workload of about 170,000 requests per hour.

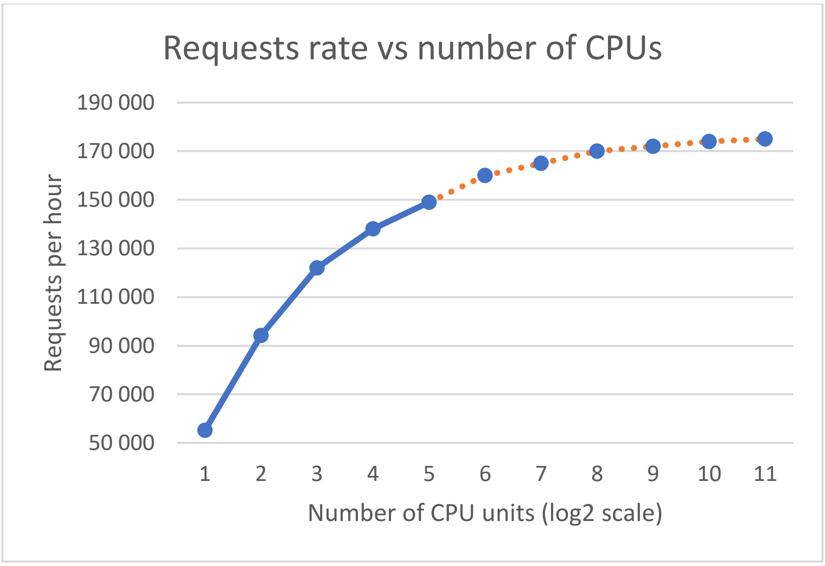


FIGURE 48: GRAPH SHOWING HOW WELL LARMAS SCALES VERTICALLY AS THE NUMBER OF CPU UNITS IS INCREASED. (ORANGE PLOT IS A PROJECTION)

5.4.2. Horizontal Scaling

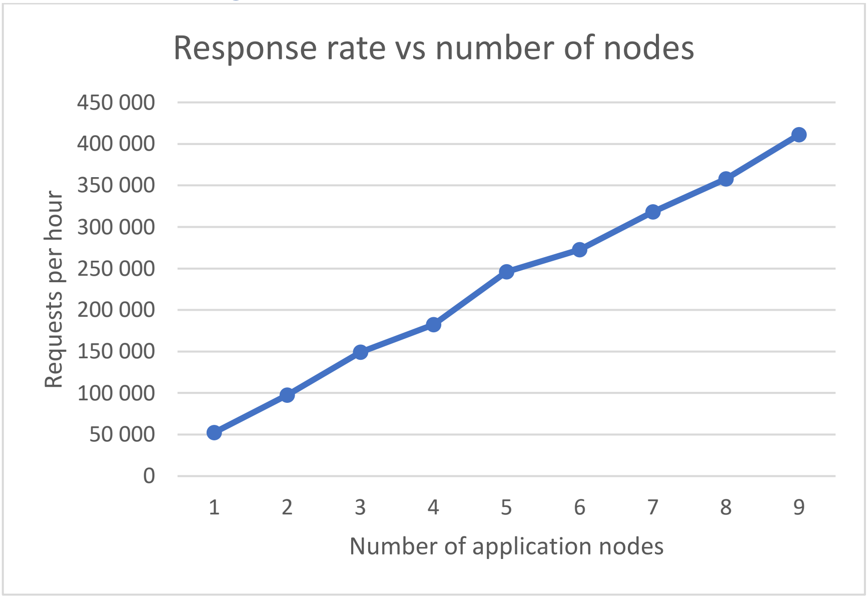


FIGURE 49: GRAPH SHOWING HOW LARMAS SCALES HORIZONTALLY AS THE NUMBER OF APPLICATION NODES INCREASES.

Figure 49 is a graph showing how maximum request rate LARMAS could support varies as the number of application server nodes increases. The graph shows a linear increase for the 9 nodes that are added and there is no sign of diminishing returns. In contrast to vertical scaling which could only support 145,100 requests per hour with 16 CPU units, this system could be vertically scaled up to support up to 411,100 requests per hour with 9 nodes (9 CPU units). This experiment has therefore shown that the LARMAS server architecture specified in 3.4.5 is a highly scalable architecture. This setup is capable of handling much more requests if each node is also scaled up.

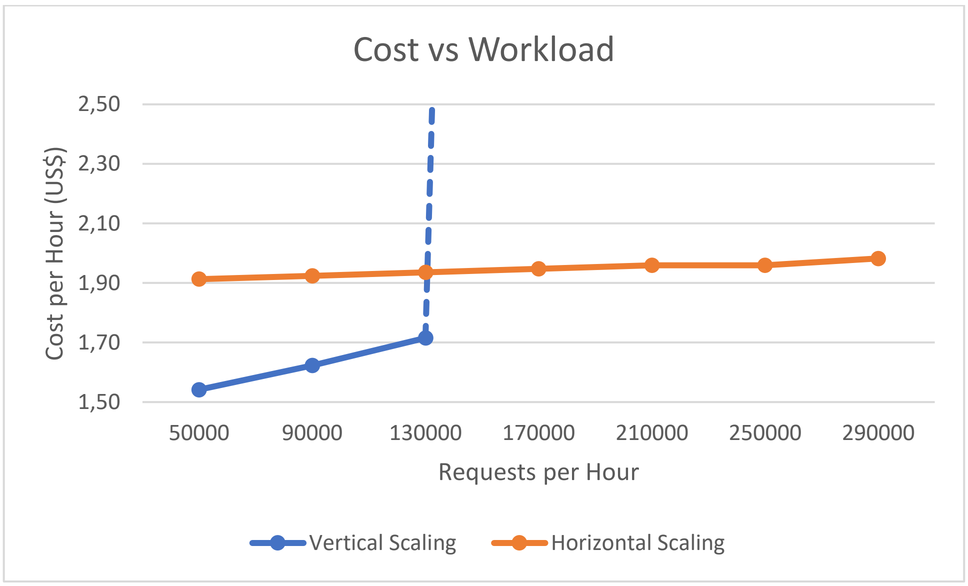
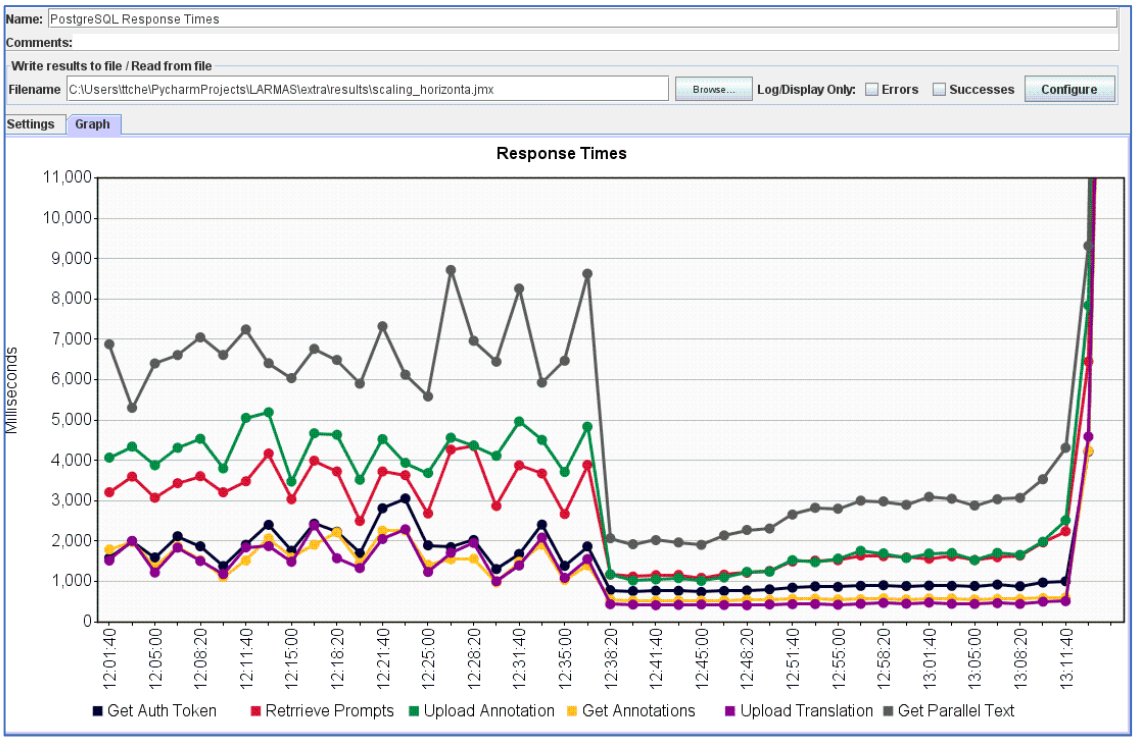


FIGURE 50: GRAPH OF HOW THE COST VARIES AS THE WORKLOAD INCREASES.

Figure 50 is a graph showing how the cost to run LARMAS varies as the workload increases for the 2 different methods of scaling. It would be more cost effective to host LARMAS on a single machine using the architecture described under section 3.4.2 if the workload is under 130,000 requests per hour. If the workload increases beyond this vertical scaling would rapidly become expensive requiring an instance with 128 instances to support a workload of 170,000 requests per hour. Such an instance costs $14.338 per hour to run. The server architecture described under 3.4.4 would be more cost-effective. This setup costs more for smaller workloads because it uses an extra server for load balancing.

5.4.3. Load balancing algorithms

Figure 51 shows how the response times varied with the different load balancing algorithms and Figure 52 shows how the CPU usage of the nodes varied. The first load balancing algorithm to be tested was the default algorithm, round robin, shown in Figure 51 from 12:01 to 12:35 CAT and in Figure 52 from 10:01 to 10:35 UTC. The second algorithm to be tested was “Least number of connections” algorithm (**leastconn**) which ran from 1235 to 1310 in Figure 51 and in Figure 52 from 10:35 to 11:10 UTC.

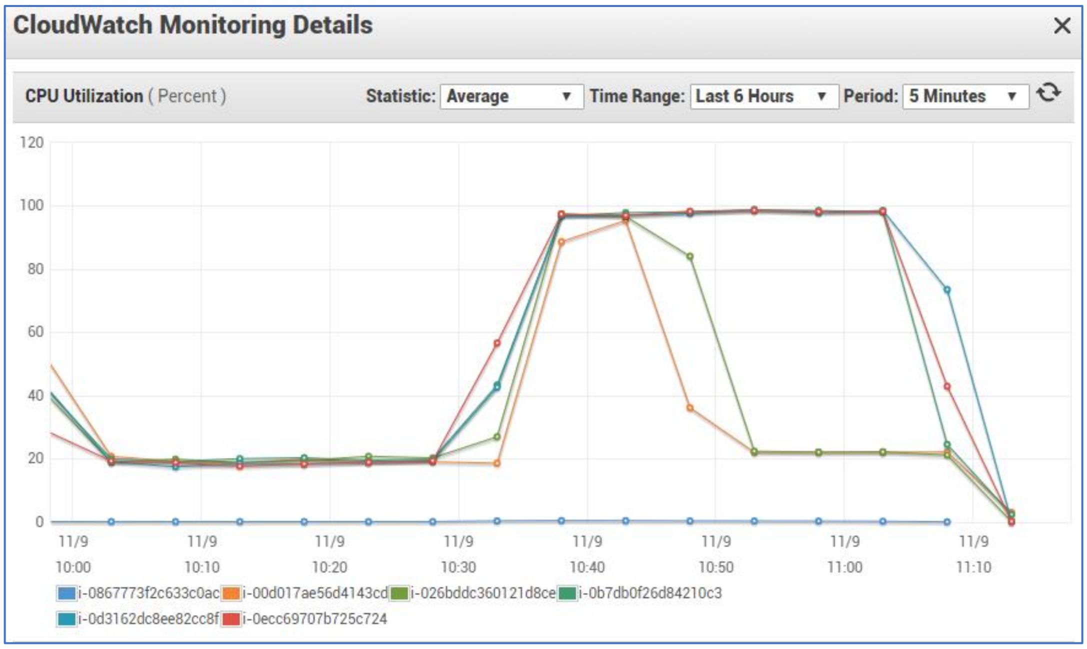

Name: PostgreSQL Response Times

Comments:

Write results to file / Read from file

Filename C:\Users\ttche\PycharmProjects\LARMAS\extra\results\scaling_horizonta.jmx

FIGURE 51: GRAPH SHOWING THE PERFORMANCE OF THE 3 LOAD BALANCING ALGORITHMS PROVIDED BY HAPROXY.


CloudWatch Monitoring Details

CPU Utilization (Percent) Statistic: Average Time Range: Last 6 Hours Period: 5 Minutes

FIGURE 52: CPU USAGE FOR THE DIFFERENT LOAD BALANCING ALGORITHMS

The response times for the leastconn algorithms are significantly less than those with round robin and also do not fluctuate as much with the round robin algorithm. The ineffectiveness of the round robin algorithm is also shown by the CPU usage where only 20% of the CPU was used in contrast to 100% CPU usage with the leastconn algorithm. The third algorithm to be tested produced very high response times of up to 90 seconds (not shown in the graph Figure 51). The test had to be stopped after a few minutes because of these erratic results.

6. Conclusions

6.1. Abstract

This chapter concludes the report on LARMAS. It reviews each of the objectives set out in the introductions and explains how each objective was achieved.

The field of NLP has grown rapidly over the past 70 years of its existence. Nowadays, many people enjoy interacting with their phones through voice commands and this field of research has also made certain technologies accessible to disabled people and allowed people from different languages to connect. Development of NLP for South African languages has been slower than it has been for many Western languages and this is mainly due to the lack of raw data needed for research and development. This project expressed the need for efficient data collection methods for NLP and the proposed solution was to create an open source client-server application to collect and manage these language resources. Six objectives were put forward and these objectives will be reviewed below.

6.2. Review of Objectives

6.2.1. Objective 1

Build a system to collect language resources for NLT. The system must allow for efficient collection of language resources to be used for NLT. These resources could be text, audio, video or other formats.

LARMAS has been built to focus on collecting data for machine translation and automatic speech recognition. It has an API with endpoints that allow contributors to upload annotations and translations of prompts stored on LARMAS. Users upload annotations along with the raw data they recorded. This raw data could be in the form of an audio file for speech annotation, or video and sensor signals for sign-language annotations. Tests have shown that the prototype can process annotation uploads in under 400ms and translations in under 200ms with minimum optimization.

6.2.2. Objective 2

Highly scalable. To allow for the large-scale collection of data, the system should be able to handle the extra load when multiple users are using the system.

LARMAS was designed to be highly scalable to support many users and store a lot of resources. It can be scaled in different ways on different levels too. The application server nodes in the distributed system can be individually scaled vertically to support more requests and more nodes can be added to the system to scale it horizontally. Tests showed that LARMAS scales linearly when it is scaled horizontally. The chosen storage system can also be scaled independently to store more data. More storage nodes can be added to the Swift cluster to improve reliability, redundancy and availability and the storage nodes can be scaled vertically to increase the storage capacity. The database is decoupled from the application server nodes to allow it to be scaled independently too.

6.2.3. Objective 3

Access to the language resources. Users of this system must have access to the resources it manages. Users will include NLP researchers and NLP tools and engines that need access to language resources to function.

LARMAS has endpoints to browse the data that has been collected. The endpoints that return the prompt annotations have links to download the raw files (audio, video, etc.) and the responses are also paginated to reduce data usage. There is also an endpoint to return parallel text between two languages and translations of any chosen language. Third-party developers can also extract the prompts used in LARMAS to build their own dictionaries. A browsable API was created to allow users to interact with the API in a browser instead of having to test it in a command line or to write code. The LARMAS home page is the API documentation which explains and gives examples of how all the endpoints work.

6.2.4. Objective 4

Must be a smart-client and/or thick-client based system. The system must allow the clients to be able to do some of the processing required on the raw data before it is transmitted to the storage server.

LARMAS was designed to be the backend system for third-party client-side applications that will be developed to collect data. Minimum processing is done on the uploaded files because LARMAS relies on the client-side applications to check and verify the data before transmitting it. LARMAS also has an admin panel that can be used by admin users to interact with the data stored on LARMAS, verify it and make any changes where necessary.

6.2.5. Objective 5

The project must provide enough documentation and support to allow for further development.

All API endpoints are well documented. The implementation of each endpoint has comments in the source code on how it was implemented and there are numerous unit and integration tests which can be used as a starting point for anyone who wants to contribute to the project. There are instructions in the appendices on how to setup and configure the different tools and technologies used in the project. The PEP8 style guide was used and the source code can be found on the public Git repository at https://github.com/tino1b2be/LARMAS.

6.2.6. Objective 6

Fully open-source. To allow for further development, the project must be fully open-source. All the tools and resources used in this project should be open-source and the project itself should be licensed under an open source licence.

The project is licensed under the GNU AGPLv3 license has the following permissions:

- **Commercial use**: This software and its derivatives can be used for commercial purposes.
- **Modification**: The software can be modified.
- **Distribution**: The software may be distributed.
- **Patent use**: The license provides an express of patent rights from contributors.
- **Private use**: The software can be modified in private.

The conditions for these permissions are:

- **License and copyright notice**: A copy of the license and copyright must be included in the software.

- **State changes**: Changes made to the code must be documented.
- **Disclose source**: Source code must be made public and available when software is distributed.
- **Network use is distribution**: Users who interact with the software via a network are given the right to receive a copy of the source code.
- **Same license**: Modifications must be released under the same license when distributing the software. In some cases, similar or related licenses can be used.

The following is a list of open-source software and tools used in the development of LARMAS:

1. **Ubuntu 16.04.3 LTS** (Xenial Xerus): This was used as the operating system for LARMAS application servers, swift object storage server and database servers.
2. **Python 3.5:** This was the programming language used to program the LAMRAS application server.
3. **Django Framework:** The web application framework used to build LARMAS.
4. **Django Rest Framework**: A Django plugin used to build the REST API for LARMAS.
5. **PyPi:** Python package manager.
6. **Git:** Version control system used for LARMAS.
7. **Swift Object Storage:** Object storage system used for storing data collected by LARMAS.
8. **SWAuth:** Authentication middleware used for swift object storage
9. **MySQL:** A DBMS used in the experiments to test the performance of LARMAS.
10. **PostgreSQL:** A DBMS used in the experiments to test the performance of LARMAS. This is also the default DBMS for LARMAS.
11. **SQLite:** The development DBMS for LARMAS and used in the experiments to test LARMAS.
12. **TravisCI:** The Continuous Integration platform used to practice Continuous Integration during the development of LARMAS.
13. **Flake8:** Tools used to enforce PEP8 programming style guide standards for Python.
14. **Coverage.py**: Used for coverage testing.
15. **HAProxy**: Used for the reverse proxy load balancer.
16. **Apache HTTP Server:** To server LARMAS in tests and experiments.

7. Recommendations and Future Work

7.1. Abstract

This chapter explains the work that still needs to be done on LARMAS and gives recommendations for anyone who wants to continue the LARMAS project.

7.2. Recommendations

The following are recommendations for further development of LARMAS:

17. **Authentication**: The authentication system in the prototype developed for LARMAS allows any client-side application to use the API. This means that data may be collected from untrusted sources or from applications that have not been approved by the administrators. To overcome this, OAuth2 should be used instead. OAuth2 is an authorization mechanism that allows third-party applications to obtain limited access to resources on an HTTP service. OAuth2 can be used to only allow authorised client-side applications to contribute to LARMAS.
18. **Faster Language**: A faster language like Java, C# or C++ may be considered for implementing LARMAS. These languages will offer must better performance and speed than Python. Alternatively, if Python meets the requirements, these faster languages can be used for processing and verify the data that has been uploaded to LARMAS.
19. **Stored Procedures**: Stored procedures are much faster at querying the database than the normal SQL statements. They should be considered for implementing some of the more complex endpoints.
20. **GraphQL**: The GraphQL query language offers more flexibility for third-party developers who want to interact with the LARMAS API. It would be better to use this to implement the API for users who want to use resources stored in LARMAS.
21. **Larger Datasets for Testing**: Larger datasets should be used for testing. LARMAS is expected to work with large amounts of data and the performance needs to be tested with such large datasets.

7.3. Future Work

- **Native Applications**: Native web, mobile and desktop client-side applications should be created for LARMAS. These will generally offer better performance if they are allowed direct access to the storage system or the database and they can provide a good guideline to third-party developers on how to develop client-side applications for LARMAS.
- **Beta Version Release**: A beta version of LARMAS needs to be released to get feedback from users before any further development. This feedback will allow for better estimates of the workload and more accurate predictions and modelling of key scenarios.
- **Experiments with NoSQL**: NoSQL DBMS promise high performance and it is worth experimenting with them. There was not enough time to implement the backend for NoSQL DBMS for this project.
- **Data Warehousing**: A data warehouse should be implemented to allow third-party developers to perform data mining projects.
- **Processing Uploaded Data**: Background services and cron jobs must be created to carry out asynchronous operations to verify the data uploaded to LARMAS. These operations may include automated feature extraction and annotation, noise reduction, compression, etc.

8. References

[1] J. Hutchins, “Warren Weaver and the launching of MT,” *Early years in machine translation,* pp. 17-20, 2000.

[2] L. Irwin, “Machine Translation,” KantanMT, 2017.

[3] R. V. Yampolskiy, “AI-Complete, AI-Hard, or AI-Easy: Classification of Problems in Artificial Intelligence,” University of Louisville Department of Computer Engineering and Computer Science , Louisville, 2011.

[4] F. J. Och, C. Tillmann and H. Ney, “Improved Alignment Models for Statistical Machine Translation,” *Computational Linguistics,* vol. 29, no. 1, pp. 19-51, 2003.

[5] C. Chelba, D. B. M. Shugrina, P. Nguyen and S. Kumar, “Large Scale Language Modeling in Automatic Speech Recognition,” Google, 2012.

[6] S. Merity, “The wikitext long term dependency language modeling dataset,” Salesforce Metamind, 09 2016. [Online]. Available: https://einstein.ai/research/the-wikitext-long-term-dependency-language-modeling-dataset. [Accessed 01 10 2017].

[7] J. Tiedemann, “The OPUS corpus - parallel & free,” *Proceedings of the Fourth International Conference on Language Resources and Evaluation ,* 2004.

[8] J. S. G. J. G. F. W. F. D. P. David Graff, 1996 English Broadcast News Speech (HUB4) LDC97S44. DVD., Philadelphia: Linguistic Data Consortium, 1997.

[9] G. Books, “Google Books Ngram Viewer,” Google, 07 2012. [Online]. Available: https://storage.googleapis.com/books/ngrams/books/datasetsv2.html. [Accessed 24 10 2017].

[10] VoxForge, “About VoxForge,” 01 2017. [Online]. Available: http://www.voxforge.org/home/about. [Accessed 01 10 2017].

[11] A. Rousseau, P. Deléglise and Y. Estève, “Enhancing the TED-LIUM Corpus with Selected Data for Language Modeling and More TED Talks,” *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC’14),* 2014.

[12] Appen, “Speech Data Collection,” 2017. [Online]. Available: https://appen.com/services/speech-data-collection/. [Accessed 24 10 2017].

[13] Globalme, “Our Data Collection Services,” 2017. [Online]. Available: https://www.globalme.net/data-collection. [Accessed 24 10 2017].

[14] A. Projects, “Open Speech Recording,” Google, 06 2017. [Online]. Available: https://aiyprojects.withgoogle.com/open_speech_recording. [Accessed 24 10 2017].

[15] N. J. D. Vries, “Effective automatic speech recognition data collection for under-resourced languages,” North-West University, Potchefstroom, 2011.

[16] S. Tripathi and J. Sarkhel, “Approaches to machine translation,” *Annals of Library and Information Studies,* vol. 57, pp. 388-393, 2010.

[17] P. Lison, “Introduction to Statistical Machine Translation,” University of Oslo Language Technology Group, Oslo, 2016.

[18] M. Collins, “Language Modeling,” Columbia University, Columbias, 2013.

[19] D. Parsons, Dynamic Web Application Development Using XML and Java, London: Gaynor Redvers-Mutton, 2008.

[20] R. Cummings, “RESTful APIs and 802.1Qcc,” in *IEEE 802.1*, Budapest, 2016.

[21] Instagram, “User Endpoints,” Instagram API Documentation, 2017. [Online]. Available: https://www.instagram.com/developer/endpoints/users/. [Accessed 28 10 2017].

[22] J. M. Tekli, E. Damiani, R. Chbeir and G. Gianini, “SOAP Processing Performance and Enhancement,” *IEEE Transactions on Services Computing,* vol. 5, no. 3, pp. 387-403, 2012.

[23] TutorialsPoint, “SOAP - Examples,” TutorialsPoint, 2017. [Online]. Available: https://www.tutorialspoint.com/soap/soap_examples.htm. [Accessed 28 10 2017].

[24] L. Byron, “GraphQL: A data query language,” Facebook, 15 09 2015. [Online]. Available: https://code.facebook.com/posts/1691455094417024. [Accessed 28 10 2017].

[25] GraphQL, “Introduction to GraphQL,” Facebook Open Source, 2017. [Online]. Available: http://graphql.org/learn/. [Accessed 28 10 2017].

[26] V. Okanovic, “Web application development with component frameworks,” *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO),* pp. 889-892, 2014.

[27] L. Daly, Next-Generation Web Frameworks in Python, Boston: O'Reilly Media, 2007.

[28] Instagram, “Web Service Efficiency at Instagram with Python,” Instagram Engineering, 21 07 2016. [Online]. Available: https://engineering.instagram.com/web-service-efficiency-at-instagram-with-python-4976d078e366. [Accessed 10 10 2017].

[29] M. Robenolt, “Scaling Django to 8 Billion Page Views,” Disqus, 24 09 2013. [Online]. Available: https://blog.disqus.com/scaling-django-to-8-billion-page-views. [Accessed 10 10 2017].

[30] D. H. Hansson, “Ruby On Rails,” Ruby On Rails, 2017. [Online]. Available: https://rubyonrails.org/. [Accessed 12 10 2017].

[31] G. Abowd, R. Allen and D. Garlan, “Using Style to Understand Descriptions of Software Architecture,” *Symposium on Foundations of Software Engineering,* pp. 9-20, December 1993.

[32] M. Richards, Software Architecture Patterns, Sebastopol,: O'Reilly Media, Inc, 2015.

[33] B. Foote and J. Yoder, "Big Ball of Mud," *Patterns Languages of Programs,* 06 1999.

[34] J. Thönes, "Microservices," *IEEE Software,* vol. 32, no. 1, pp. 116-116, 2015.

[35] Oracle, "What is MySQL?," 2017. [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/what-is-mysql.html. [Accessed 18 09 2017].

[36] T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys,* vol. 15, no. 4, pp. 289-290, 1983.

[37] O. Java, "A Relational Database Overview," Oracle Documentation, 2015. [Online]. Available: https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html. [Accessed 18 September 2017].

[38] Oracle, "InnoDB and the ACID Model," Oracle Corporation, 2017. [Online]. Available: https://dev.mysql.com/doc/refman/5.6/en/mysql-acid.html. [Accessed 19 09 2017].

[39] L. Whitney, "Oracle pledges to play well with MySQL," CNET, 14 12 2009. [Online]. Available: https://www.cnet.com/news/oracle-pledges-to-play-well-with-mysql/. [Accessed 18 09 2017].

[40] Oracle, "The Main Features of MySQL," Oracle Corporation, 2017. [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/features.html. [Accessed 18 09 2019].

[41] SQLite, "Appropriate Uses For SQLite," SQLite, 2017. [Online]. Available: https://sqlite.org/whentouse.html. [Accessed 18 September 2017].

[42] C. Luo, H. Okamura and T. Dohi, "Modeling and Analysis of Multi-version Concurrent Control," *2013 IEEE 37th Annual Computer Software and Applications Conference,* pp. 53-58, July 2013.

[43] F. Zendaoui and W. K. Hidouci, "Performance evaluation of serializable snapshot isolation in PostgreSQL," *2015 12th International Symposium on Programming and Systems (ISPS),* pp. 1-11, 2015.

[44] M. Aslett, "How Will The Database Incumbents Respond To NoSQL And NewSQL," 451 Research, London, 2011.

[45] A. Pavlo, "NewSQL," Brown University, Providence, Rhode Island, 2012.

[46] W. Naheman and J. Wei, "Review of NoSQL databases and performance testing on HBase," *Proceedings 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC),* pp. 2304-2309, 2013.

[47] Y. Gu, S. Shen, J. Wang and J.-U. Kim, "Application of NoSQL database MongoDB," *2015 IEEE International Conference on Consumer Electronics - Taiwan,* pp. 158-159, 2015.

[48] Redis, “Introduction to Redis,” Salvatore Sanfilippo, 2017. [Online]. Available: https://redis.io/topics/introduction. [Accessed 20 09 2017].

[49] Resis, “How fast is Redis?,” Salvatore Sanfilippo, 2017. [Online]. Available: https://redis.io/topics/benchmarks. [Accessed 20 09 2017].

[50] A. Chebotko, A. Kashlev and S. Lu, “A Big Data Modeling Methodology for Apache Cassandra,” *2015 IEEE International Congress on Big Data,* pp. 238-245, 2015.

[51] M. S. R. Manual, “How MySQL Uses Memory,” Oracle Corporation, 01 01 2017. [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/memory-use.html. [Accessed 12 10 2017].

[52] M. Factor, K. Meth, D. Naor, O. Rodeh and J. Satran, “Object storage: the future building block for storage systems,” *Local to Global Data Interoperability - Challenges and Technologies,* pp. 119-123, 2005.

[53] K. Schwaber, Agile Project Management with Scrum, Microsoft Press, 2004.

[54] K. Beck, Test-Driven Development By Example, Addison-Wesley Professional, 2002.

[55] B. O'Sullivan, Mercurial: The Definitive Guide, O'Reilly Media, 2009.

[56] C. Oldwood, “Branching Strategies,” *Overload Journal,* vol. 121, 2014.

[57] J. Holck and N. Jørgensen, “Continuous Integration and Quality Assurance: a case study of two open source projects,” *Australasian Journal of Information Systems ,* vol. 11, no. 1, 2003.

[58] L. Chen, M. A. Babar and B. Nuseibeh, “Characterizing Architecturally Significant Requirements,” *IEEE Software,* vol. 30, no. 2, pp. 38-45, 2013.

[59] J. Nielsen, Designing Web Usability: The Practice of Simplicity, Thousand Oaks: New Riders Publishing, 1999.

[60] J. Brutlag, “Speed Matters for Google Web Search,” Google Inc, Melno Park, 2009.

[61] PageSpeed, “Improve Server Response Time,” Google Developers, 08 04 2015. [Online]. Available: https://developers.google.com/speed/docs/insights/Server. [Accessed 26 10 2017].

[62] CHET, “Headcount and Full-time Equivalent (FTE) Enrolments,” *South African Higher Education Performance Indicators 2009-2015,* 2015.

[63] Hootsuite, “Digital In 2017 Global Overview,” We Are Social, 2017.

[64] M. Jensen, The Everything Business Planning Book: How to Plan for Success in a New or Growing Business, Sandy Hook: Everything Books, 2001.

[65] E. S. D. Team, Microsoft Exchange Server 2003 Performance and Scalability Guide, Microsoft Corporation, 2006.

[66] Django, "Django Success Story Bitbucket," Django Software Foundation, 08 09 2008. [Online]. Available: https://web.archive.org/web/20160420214550/https://code.djangoproject.com/wiki/DjangoSuccessStoryBitbucket. [Accessed 10 10 2017].

[67] Django, "Django in use at washingtonpost.com," Django Software Foundation, 08 12 2005. [Online]. Available: https://www.djangoproject.com/weblog/2005/dec/08/congvotes/. [Accessed 10 10 2017].

[68] T. Chemvura, "LRMS Repository," Github, 3 10 2017. [Online]. Available: https://github.com/tino1b2be/LARMAS. [Accessed 10 10 2017].

[69] T. Chemvura, "tino1b2be/LRMS Travis Build," Travis CI, 03 10 2017. [Online]. Available: https://travis-ci.org/tino1b2be/LARMAS/. [Accessed 10 10 2017].

[70] K. Yao and J. Gao, "Law of Large Numbers for Uncertain Random Variables," *IEEE Transactions on Fuzzy Systems,* vol. 24, no. 3, 2016.

[71] A. Murgu, "EEE3036S Probability and Statistical Design in Engineering," Department of Electrical Engineering, University of Cape Town, Cape Town, 2015.

[72] M. Shum, "SS-222a Lecture 5: Large Sample Theory," *California Institute of Technology,* 2017.

[73] P. G. Marr, "Lecture 5: Testing for Normality," *GEO444 Shippensburg University,* p. 28, 2016.

[74] Django, "Django Documentation," Django Software Foundation, 04 04 2017. [Online]. Available: https://docs.djangoproject.com/en/1.11/. [Accessed 22 10 2017].

[75] OpenStack, "Object Storage API," OpenStack Documentation, 20 10 2017. [Online]. Available: https://developer.openstack.org/api-ref/object-store/index.html. [Accessed 24 10 2017].

[76] OpenStack, "Erasure Code Support," OpenStack Documentation, 23 10 2017. [Online]. Available: https://docs.openstack.org/swift/latest/overview_erasure_code.html. [Accessed 24 10 2017].

[77] A. Chow, "OpenStack Series: Part 7 – Swift – Object Storage Service," Blogspot, 06 11 2014. [Online]. Available: https://cloudn1n3.blogspot.co.za/2014/11/openstack-series-part-7-swift-object.html. [Accessed 25 10 2017].

[78] OpenStack, "The Auth System," OpenStack Documentation, 23 10 2017. [Online]. Available: https://docs.openstack.org/swift/latest/overview_auth.html. [Accessed 25 10 2017].

[79] OpenStack, “Hardware Requirements,” OpenStack Documentation, 12 06 2017. [Online]. Available: https://docs.openstack.org/mitaka/install-guide-rdo/overview.html. [Accessed 25 10 2017].

[80] M. Anicas, “5 Common Server Setups For Your Web Application,” Digital Ocean, 30 05 2014. [Online]. Available: https://www.digitalocean.com/community/tutorials/5-common-server-setups-for-your-web-application. [Accessed 31 10 2017].

[81] W. C. a. C. C. Franco Callegati, “Virtual Networking Performance in OpenStack Platform for Network Function Virtualization,” *Journal of Electrical and Computer Engineering,* vol. 2016, p. 15, 2016.

[82] AWS, “Amazon EC2 Instance Types,” Amazon Web Services, 2017. [Online]. Available: https://aws.amazon.com/ec2/instance-types/. [Accessed 03 11 2017].

[83] AWS, “Amazon EC2 Pricing,” Amazon Web Services, 2017. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/. [Accessed 03 11 2017].

[84] AWS, “Amazon RDS for MySQL Pricing,” Amazon Web Services, 2017. [Online]. Available: https://aws.amazon.com/rds/mysql/pricing/. [Accessed 09 11 2017].

[85] Apache, “Apache JMeter,” Apache Software Foundation, 2017. [Online]. Available: https://jmeter.apache.org/. [Accessed 03 11 2017].

[86] SQLite, “SQLite Is Serverless,” SQLite, 2017. [Online]. Available: https://www.sqlite.org/serverless.html. [Accessed 10 11 2017].

[87] S. Gordon, “The Normal Distribution,” *Mathematics Learning Centre, University of Sydney,* 2006.

[88] D. Mills, “RFC 889 Internet Delay Experiments,” *Network Working Group,* 1983.

[89] P. Bhattacharyya, Machine Translation, Boca Raton: Chapman and Hall/CRC, 2015.

[90] OpenStack, “Erasure Code Support,” OpenStack Documentation, 23 10 2017. [Online]. Available: https://docs.openstack.org/swift/latest/overview_erasure_code.html. [Accessed 24 10 2017].

[91] OpenStack, “Object Storage API,” OpenStack Documentation, 20 10 2017. [Online]. Available: https://developer.openstack.org/api-ref/object-store/index.html. [Accessed 24 10 2017].

[92] Oracle, “A Relational Database Overview,” Oracle Java Documentation, 2015. [Online]. Available: https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html. [Accessed 18 September 2017].

Appendix A: Setting up the development environment

1. Install Ubuntu 16.04 on the machine from https://www.ubuntu.com/download
2. Install Git, Python 3 and PyPi package manager.
   ```
   # sudo apt update
   # sudo apt install -y git python3=3.5.2 python-pip3
   ```
3. Clone the Git repository for LARMAS
   ```
   # git clone https://github.com/tino1b2be/LARMAS.git
   ```
4. Install LARMAS packages using PyPi
   ```
   # cd LARMAS
   # sudo pip3 install -r requirements.txt
   ```
5. Run LARMAS tests.
   ```
   # coverage run --source=. manage.py test -v 2 -noinput
   # coverage report -m
   ```
6. Launch the development serverto verify functionality
   ```
   # python3 manage.py runserver 0.0.0.0:8000
   ```

Appendix B: Setting up swift object storage.

This is a guide to install Swift Object Storage on Ubuntu 16.04

The username and hostname for the VM MUST be set to “swift” for the scripts in this guide to work.

NB: All commands to be run as sudo.

1. Configure the storage devices

- Create 3 virtual hard disks (50MB each) and attach them to the VM. Verify that these storage devices are located at /dev/sdb, /dev/sdc and /dev/sdd. If they aren’t, the scripts in this guide will not work. To do this you can use:
  ```
  $ sudo fdisk -l /dev/sdb /dev/sdc /dev/sdd
  ```
- Install supporting utility packages:
  ```
  $ sudo apt-get install xfsprogs rsync
  ```
- Format the /dev/sdb, /dev/sdc and /dev/sdd devices as XFS:
  ```
  $ sudo mkfs.xfs /dev/sdb -f
  $ sudo mkfs.xfs /dev/sdc -f
  $ sudo mkfs.xfs /dev/sdd -f
  ```
- Create the mount point directory structure:
  ```
  $ sudo mkdir -p /srv/node/sdb

  $ sudo mkdir -p /srv/node/sdc

  $ sudo mkdir -p /srv/node/sdd
  ```
- Edit the **/etc/fstab** file and add the following to it:
  ```
  /dev/sdb /srv/node/sdb xfs noatime,nodiratime,nobarrier,logbufs=8 0 2

  /dev/sdc /srv/node/sdc xfs noatime,nodiratime,nobarrier,logbufs=8 0 2

  /dev/sdd /srv/node/sdd xfs noatime,nodiratime,nobarrier,logbufs=8 0 2
  ```
- Mount the devices:
  ```
  $ sudo mount /srv/node/sdb

  $ sudo mount /srv/node/sdc

  $ sudo mount /srv/node/sdd
  ```
- Download the preconfigured file /etc/rsyncd.conf:
  ```
  $ sudo curl -o /etc/rsyncd.conf
  https://raw.githubusercontent.com/tino1b2be/swift-on-
  vm/v0.1/config_files/rsyncd-sample.conf
  ```
- Edit the /etc/default/rsync file and enable the rsync service:
  ```
  RSYNC_ENABLE=true
  ```
- Start the rsync service:
  ```
  $ sudo service rsync start
  ```

2. Install and Configure Swift

- Install memcached and swift packages

  ```
  $ sudo apt-get install memcached python-memcache swift swift-proxy
  python-swiftclient swift-account swift-container swift-object
  ```

  - Download and run the script to install and configure swift.

    ```
    $ sudo curl -o /tmp/swift_storage_script
    https://raw.githubusercontent.com/tino1b2be/swift-on-
    vm/v0.1/scripts/swift_storgae_script

    $ . /tmp/swift_storage_script
    ```

  - The “swift_storage_script” does the following:
  - Installs and configures memcached (used to store tempauth tokens)
  - Installs ad configures the Swift Proxy servers
  - Installs and confiures the Account, Container and Object servers.
  - Sets proper ownership of the config files and also for the recon directory that’s also created.
  - Creates the account, container and storage rings.
  - Restarts the services used by Swift (swift servers and memcached)

- Verify operation

```
$ swift stat

                        Account: AUTH_swift

                     Containers: 0

                        Objects: 0

                          Bytes: 0

 Containers in policy "policy-0": 0

    Objects in policy "policy-0": 0

      Bytes in policy "policy-0": 0

                    X-Timestamp: 1444143887.71539

                     X-Trans-Id: tx1396aeaf17254e94beb34-0056143bde

                   Content-Type: text/plain; charset=utf-8

                  Accept-Ranges: bytes
```

3. Install Swauth:

```
$ sudo apt install swauth
```

- Modify the pipeline in /etc/swift/proxy-server.conf to replace “tempauth” with “swauth”

  ```
  [pipeline:main]

  pipeline = catch_errors cache swauth proxy-server
  ```

- Add a new section for swauth in the same file /etc/swift/proxy-server.conf

  ```
  [filter:swauth]
  use = egg:swauth#swauth
  Set log_name = swauth
  Super_admin_key = <swauthkey>
  ```

- Modify the [app:proxy-server] section in the /etc/swift/proxy-server.conf file:

  ```
  [app:proxy-server]
  use = egg:swift#proxy
  allow_account_management = true
  account_autocreate = true
  ```

- Restart the swift proxy server:

  ```
  $ swift-init proxy restart
  ```

- Initialize the Swauth backing store in Swift:

  ```
  $ swauth-prep -K <swauthkey>
  ```

- Verify operation:
  Create a new account (account = swift, user = swift, password = swift):

  ```
  $ swauth-add-user -A http://127.0.0.1:8080/auth/ -K swauthkey -a swift
  swift swift
  ```

- Try out the new auth system with swift (check the account information):

  ```
  $ swift -A http://127.0.0.1:8080/auth/v1.0 -K swift -U swift:swift stat
  ```

- Or a shorter way of doing that is to make use the OpenRC file saved in $HOME/swiftrc(or just create your own) to shorten the command to just:

  ```
  $ swift stat
  ```

For troubleshooting refer to OpenStack documentation at:
https://docs.openstack.org/newton/install-guide-ubuntu/environment.html

Appendix C: Serving LARMAS with Apache

This is a guide to serve LARMAS on Apache in Ubuntu 16.04.

- Install Apache and Mod-WSGI

```
sudo apt-get -y install apache2 libapache2-mod-wsgi-py3
```

- Enable WSGI

```
sudo a2enmod wsgi
```

- Disable default Apache site

```
sudo a2dissite 000-default
```

- Create a new Apache site for LARMAS by create a new configuration file and adding the following configurations (/etc/apache2/sites-available/larmas.conf):

```
# sudo nano /etc/apache2/sites-available/larmas.conf

<VirtualHost *:80>

      ServerName larmas


      WSGIDaemonProcess larmas user=larmas group=larmas threads=5
python-path="/home/larmas/grouped/LARMAS/"

      WSGIScriptAlias / /home/larmas/grouped/LARMAS/LARMAS/wsgi.py

      <Directory /home/larmas/grouped/LARMAS/>

          WSGIProcessGroup larmas

          WSGIApplicationGroup %{GLOBAL}

          WSGIScriptReloading On

          Require all granted

      </Directory>

     Alias /static /home/larmas/grouped/LARMAS/static

     <Directory /home/larmas/grouped/LARMAS/static>

         Require all granted

     </Directory>

 </VirtualHost>
```

- Install MySQL client, Django and PyPi:

```
sudo apt install libmysqlclient-dev python3-pip python3-django
```

- Download LARMAS from github:

```
# sudo mkdir /home/larmas/grouped

# cd /homr/larmas/grouped

# git clone https://github.com/tino1b2be/LARMAS.git
```

- Install LARMAS dependencies

```
# sudo python3 pip3 install -r requirements.txt
```

- Create a new user.

```
# sudo adduser larmas

Enter new UNIX password:   [always use only good passwords]

Full Name []: WSGI user [always put name of responsible admin]

# sudo usermod --lock larmas
```

- Change ownership and permissions for all files in the LARMAS directory

```
# sudo chmod u=rwx,g=srwx,o=x /home/larmas/grouped

# sudo chown -R larmas.larmas /home/larmas/

# sudo find /home/larmas/grouped/ -type f -exec chmod -v ug=rw {} \;

# sudo find /home/larmas/grouped/ -type d -exec chmod -v
u=rwx,g=srwx {} \;
```

- Add the current user to larmas and create a new user group

```
# sudo adduser $(whoami) larmas

# newgrp larmas
```

Add LARMAS server to Apache and restart Apache Server.

```
# sudo a2ensite larmas

# sudo service apache2 restart
```

LARMAS should now be running and accepting HTTP requests on port 80

Appendix D: Configuring HAProxy Load Balancer

This is a guide to install and configure HAProxy on Ubuntu 16.04

1. Install HAProxy onto the machine
   ```
   # sudo apt install haproxy -y
   ```
2. Enable HAProxy. Open the file /etc/default/haproxy and set the ENABLE variable to 1 (it is 0 by default)
   ```
   # sudo nano /etc/default/haproxy
   ________________
   ENABLED=1
   ```
3. Modify HAProxy config file /etc/haproxy/haproxy.cfg and add the following at the end of the file:

   ```
   # sudo nano /etc/haproxy/haproxy.cfg
   ________________
   ...
   frontend larmas_in
         bind *:80
         default_backend larmas_back


   backend larmas_back
         balance roundrobin
         mode http
         server node1 <node1_ip_address>:80 check
         server node2 <node1_ip_address>:80 check
         server node3 <node1_ip_address>:80 check
   ____________________
   ```

   This configures HAProxy to redirect any requests coming in on port 80 to the 3 nodes using a round robin algorithm.

4. Restart HAProxy.
   ```
   # sudo service haproxy restart
   ```
5. HAProxy is now configured to route all incoming traffic to the configured nodes