A Reliability-Aware Framework for Service-Based Software Development

by

Ian Andrusiak

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of

Masters of Applied Science

in

Electrical and Computer Engineering
University of Ontario Institute of Technology

Oshawa, Ontario, Canada
© Ian Andrusiak, 2017

Abstract

A Reliability-Aware Framework for Service-Based Software Development

Ian Andrusiak
University of Ontario Institute of Technology, 2017

Advisor:
Dr. Qusay H. Mahmoud

It is becoming common to see software applications taking advantage of web services available publicly to meet their needs, rather than developing an in-house solution. This introduces the problem where failures can occur on the network, or on the service provider, outside the influence of the developer. This thesis proposes a reliability-aware framework with a focus on availability, which applies a recovery block scheme to services provided by different developers. The proposed framework allows developers to specify alternative services which meet the core specifications of their primary service. When a failure is determined to have occurred, the request to the primary service is mapped to an alternative service. A prototype has been developed as a proof of concept, which has been evaluated on metrics based on potential use cases. The experimental results show that the system is successful at providing availability when failure occurs, at a cost to overall performance.

Acknowledgments

I would like to express my sincere gratitude to my thesis supervisor Dr. Qusay Mahmoud, for his continued support, motivation, and guidance throughout my time as a graduate student. I would not have been able to complete this work without his guidance and motivation. During my studies under him I have developed both my research and development skills.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software systems have traditionally been developed in a consistent way, whereby software developers would either design and develop their software to meet their needs, or use an off-the-shelf software solution. This is how a lot of software is developed today and still produces effective results. These two development methodologies have proven that they are effective at meeting the needs of projects. In recent years, a new development style is being adopted by many developers. Similar to the idea of using off-the-shelf software to meet the developer's needs, developers have been embracing the idea of developing software via Representational State Transfer (REST) services available on the Internet. This process is done through packaging up data and sending it to an appropriate web service, instead of running all the software locally. These services provide functionality to the developers without the overhead of development. The developer only needs to know how to interact with the service and handle the various responses. This development style is a realization of concepts provided from Service Oriented Architecture (SOA) and more specifically the Microservice Architecture. These architectural patterns describe how services are developed and exposed to other software. Services are developed as a black box and exposed to be used by developers, where the service boundaries are clearly defined [1, 2]. Software developers using services within their software implementation receive many benefits over a traditional system. Maintenance on software built with services is easier as parts of the implementation are being handled by the services. Changes can be

applied easily as long as the interface with the service remains the same. Another benefit is the abstraction of the complexities of the service. Software developers only need to worry about the interaction with the service. Lastly the developer is transferring the overhead cost of development and infrastructure to network and service charges. Using services introduces challenges that software developers need to take into consideration such as the reliability and availability of the service, and the latency introduced through network conditions.

The popularity of developing software through services can be attributed to both improvements in network infrastructure and the changes in service technologies. Web service models have two core components that allow for software to interact with them. These components are the web service itself and the web service definition which describes the interaction with the service. Registries exist which store definition files for many services allowing software developers to discover new services. Once the software developer has the definition file, they are able to begin interacting with the service. Originally this was all handled using the Simple Object Access Protocol (SOAP) [3]. In SOAP, the file that describes how to interact with the web service is known as a Web Service Definition Language (WSDL) document. This document is an eXtensible Markup Language (XML) file which defines interaction with the web service [3]. This technology has given way to the web technology Representational State Transfer (REST). REST was used for serving web pages before being adopted to web services. The popularity of RESTful services over SOAP services can be attributed to many different factors. The most common reasons being how lightweight it is, how easily they can be integrated, and its ease of use [3]. The main principle behind RESTful services is that everything is considered a resource which can be accessed via a Unique Resource Identifier (URI). REST services are

both flexible in the protocol they can use and the data type for interaction. Accessing recourses via REST is a stateless operation, but sessions can be maintained via session ID's [4].

With the surge in popularity of web services developed through the REST web technology, a new problem emerged in regard to how to manage and handle the vast amount of services. This problem is made worse through applications which are composed through the usage of multiple web services. It is not uncommon to see applications utilization more than one service from different providers. This also introduces another problem where it is cumbersome on the developer to register for each Application Program Interface (API) and learn how to execute them. Each web service is designed differently which means that the software developer would first have to learn how to interact with it. Platforms within recent years have been created where their business model aims to assist developers in managing and consuming API's from different providers. A few popular examples of platforms that provide API management would be Mashape and Apigee [5, 6]. These platforms empower developers through the use of consistency in the interaction with services through their API's. Users only need to learn how services are consumed on the management platform once, as they can consume any other service using the same process. They also provide service providers a valuable infrastructure in which they can allow developers the usage of their service with tools such as authentication, usage metrics, and payment management [7].

By developing software systems using web services, new concerns are apparent that didn't exist within local software systems. Common issues such as latency, availability, and reliability become more prominent when the software is executed online. The reliability is

defined as the probability of receiving correct results while the availability is the likelihood of receiving any results from the service. Service providers can alleviate potential concerns by implementing software fault-tolerant techniques within their system through the use of redundancy and diversity across their deployment zones. Availability and reliability are guaranteed to a certain degree to service consumers through a Service Level Agreement (SLA) between the service provider and consumer. Potential failures can be beyond the control of service providers and consumers. Zhou et al. describe that there are many different levels of failures for a web service [8]. Failures can occur at the service, host, provider, or network level. Failures occurring either in the network or provider are beyond the control of the service providers. It then becomes the service consumer's responsibility to employ a level of software fault-tolerance.

1.1 Motivation

The motivation for this work is to enable software developers everywhere to develop software systems through the many publicly available services available, instead of either developing it in-house or using publicly available solutions and integrating them locally. In developing systems in this manner, they are able to choose from a wide variety of components with minimal integration time, as they only need to be aware of the interface they are interacting with, similar to hardware development via components. When a developer considers integrating an existing solution locally, there is the potential for issues to occur through hardware used, operating system distributions, and other components interacting locally. Development through an existing service only requires knowledge of the interface and authentication, from there they are able to integrate it locally. For developers to truly embrace this design style, their concerns of issues related to

development using services such as the services reliability, availability, and latency need to be addressed.

1.2 Contributions

To this end, this thesis introduces a reliability-aware framework that provides developers with a course of action if an important web service becomes inaccessible. This is accomplished through providing developers improved reliability through availability. This is done through a scheme where developers can select alternative service providers to use if their primary service provider fails. Upon failure, their request is repackaged and sent to one of their alternative service providers which provide a similar service. Upon completion, the results received from the alternative service are repackaged to the structure the primary service provider with the most important parameters preserved. The contributions of this thesis are:

- A framework allowing developers to easily have requests failover to alternative services
- A flexible easy to use template scheme for repackaging requests
- A proof of concept implementation has been constructed and evaluated in the cloud

A potential use case of this proposed framework is service composition. Through integrating the proposed framework with service composition, a higher level of reliability can be ensured. Service providers generally offer users of their services a high-level guarantee of uptime on their services. This level of availability becomes compounded when composing multiple services together for a single objective. The more services incorporated results in a decreased availability. Through using the framework, users are able to select alternative service providers for each service in the composition.

1.3 Thesis Outline

The remainder of this thesis is structured as follows: **Chapter 2** covers the related work in both industry and literature. This includes appropriate concepts as background information regarding the topics. **Chapter 3** presents the design for the proposed reliability-aware framework. **Chapter 4** describes a proof-of-concept implementation of the design to both demonstrate the feasibility of the reliability-aware framework and evaluate its effectiveness. **Chapter 5** presents the evaluation criteria, experimental setup, and the results of the evaluation on the prototype. **Chapter 6** presents conclusions drawn on the reliability-aware framework design and prototype and offers insight for future work on this project.

Chapter 2

Background and Related Work

This chapter covers an overview of important background knowledge related to this thesis. This includes the implementation of the Service Oriented Architecture through web services. Important web technologies to build web services like Representational State Transfer and modeling languages have been summarized. Cloud computing technologies have been explored as hosting platforms for the proposed framework. Lastly software fault-tolerant techniques which use alternative services have been defined.

2.1 Service Oriented Architecture

SOA like other design paradigms, provides developers with the methodologies and models to create well structured software. Service oriented architecture was originally proposed in 1996 to help share data and logic among different applications. It provides developers a way to organize units of functionality such that they are distinct services which can be executed across the network [9]. This organization of services allows for other services or users to execute them without concerns for their implementation, only worrying about the interface to communicate with them. SOA is generally seen to have emerged from the software engineering concept known as separation of concerns [10].

SOA is defined as “A set of components which can be invoked, and whose interface descriptions can be published and discovered” by the World Wide Web Consortium (W3C) [11]. SOA is commonly applied to web services but is a paradigm which can be applied to concepts beyond software development. SOA describes a wide range of concepts and designs. It can be defined into the following key aspects [10]:

1. Loose Coupling: Services minimize the dependencies between each other. The only requirement is that the services are aware of each other.
2. Service Contract: Services must follow an agreed upon communication agreement.
3. Autonomy: Services have complete control over their own logic.
4. Abstraction: Services hide logic from the external users.
5. Reusability: Services contain divided code with the purpose of reusability.
6. Composability: Services can be collected together and composed to form composite services.
7. Statelessness: Service interaction is done independent of other interactions with the service.
8. Discoverability: Services provide descriptions such that they can be discovered and accessed.

SOA can be implemented through three core components depicted in Figure 2.1 adapted from [12]. The first component is the Service Registry. This registry provides discoverability functionality for service consumers. First a service provider must publish their service within the service registry. This allows for users to search for existing services and determine how to communicate with the desired service and use it. The next component is the Service Provider. The service provider is the entity which provides the actual service. The service provider hosts the service on a platform and publishes to the service registry how to use it. The last component is the service consumer. The service consumer is a user desiring specific functionality from a service. Once they have found the service they want from the service registry, they are able to begin interaction with the service provided from a service provider.

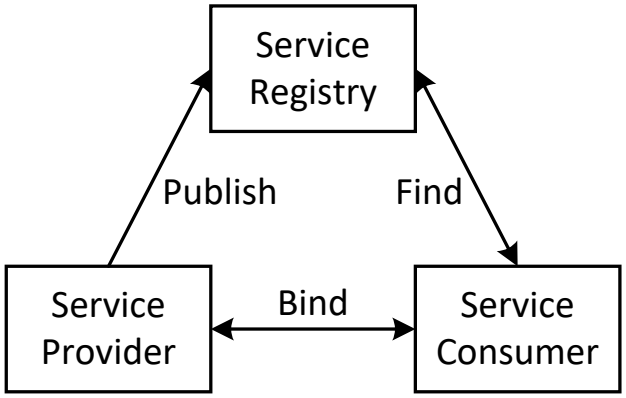

Figure 2.1: Service Oriented Architecture

A common form of SOA used today is known as the microservice architecture. Microservices can be seen as an extension or a subdivision of concepts and paradigms used within the SOA. Like SOA, the microservice architecture focuses on the concept of breaking up software logic into independent software services. With microservices, there is a strong focus on the size of each service. Each service is to be as minimal as possible only fulfilling a single goal [13]. These services can be easily distributed and scaled to meet the needs of the providers. Compared to traditional monolithic software, microservices can be scaled independently of each other (Figure 2.2) [14].

Following the design principles of SOA results in services which can be easily interconnected together, which is known as service composition. In service composition, multiple services are connected together where the outputs of one service are fed as inputs into another service. The result of composing multiple services is a new service which provides a different functionality than each of the individual services. One common technology used to assist in building service compositions is the Web Services Business Process Execution Language (WS-BPEL) [15].

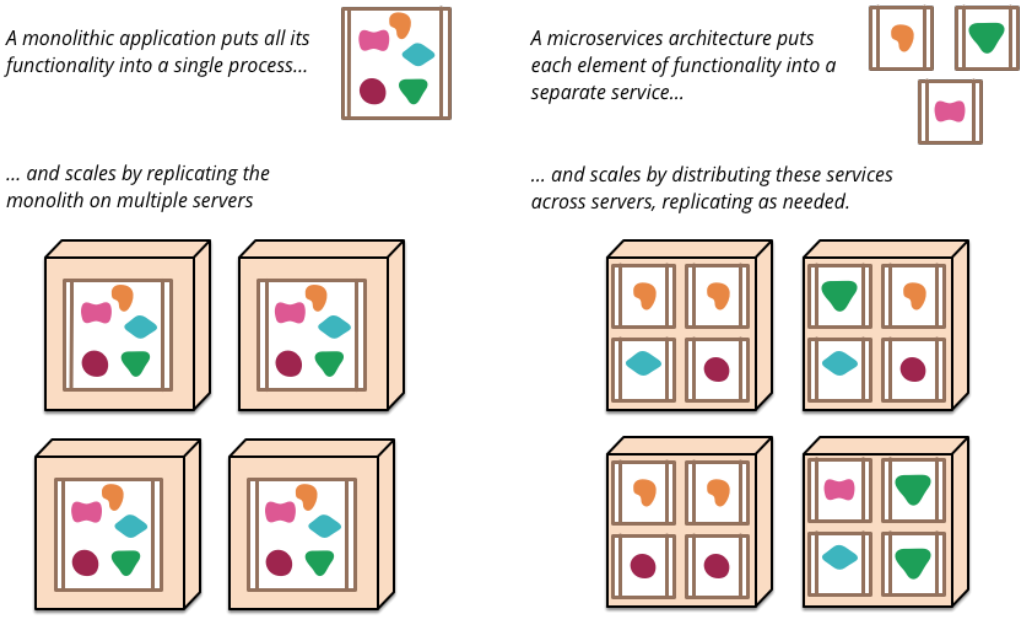

Figure 2.2: Monolith vs Microservices *[14]*

2.1.1 Web Services

One way to implement the service oriented architecture is through the use of web services. The concept of web service has changed since it was originally developed. The World Wide Web Consortium (W3C) defines a web services as a machine-to-machine interaction over a network using SOAP messages [11]. As SOAP is no longer the most predominate technology, it can also be used to refer to web services which use REST technology. The term RESTful web service or RESTful service can be used instead to specifically refer to the technology used. In general, these web services need to be capable of being defined, described and discovered [16]. Essentially this means that services should be running and accessible for users to interact with.

2.1.2 Representational State Transfer

The REpresentational State Transfer (REST) architectural principal has seen wide spread adoption since its creation. This design was originally proposed in 2000 by Roy Fielding in his doctoral dissertation [17]. This design is seen as highly successful and important as

most of the current world wide web runs using REST. From this dissertation, most literature has agreed that the following four architectural principals summarize REST [18]:

1. The central entity of a RESTful system is a resource.
2. The use of a uniform interface.
3. Resources interacted through representations.
4. The flow of the application is primary through hyperlinks within the resource representation.

Within the REST design, all important things that may need to be accessed can be considered a resource. To truly consider an object a resource it needs to have a Uniform Resource Identifier (URI) [4]. This is required as the resource can't be seen or accessed without a unique identifier. A resource can be found within the system using this URI.

With this addressing scheme, an interface is now required to allow for external users to accesses and manipulate the resources. In REST, this is done through a standard uniform interface [18]. The standard uniformity of this interface is important as it allows for a consistent interoperability among different providers. For REST, the most common used interface is the Hyper Text Transfer Protocol (HTTP) [4]. HTTP provides a common set of operations (Table 2.1) which can be used to access and manipulate the resources [19].

Table 2.1: HTTP Operations *[18]*

Operation	Description
GET	Get resource representation
HEAD	Get resource headers
PUT	Replace the resource
DELETE	Delete the resources
PATCH	Update the state of the resource
POST	Create a resource

Using the uniform interface, resources are not directly accessed. Resources are interacted through representations of that resource [18]. The representation provides information about the current state of the resource. An example of a widely-used representation format is Hyper Text Markup Language (HTML). The user can request information about a specific resource and it can be provided through an HTML page.

The last design principal refers to the traversal of resource representations for users on the platform. This principal states that the flow is primarily through hyperlinks. This was originally referred to as Hypermedia as The Engine of Application State [17]. This allows for easy traversal for users instead of having to manually craft the identifier for the representation which they want to access.

2.1.3 Modeling Language

Modeling languages are used to describe and represent the web service. A modeling language is made in such a way that it can be interpreted by either a human or a machine [20]. When a service is developed, the service provider will also develop a definition file

using this modeling language. With this definition, service consumers can understand how to interact with the service and what the service provides.

For different communication technologies, different modeling languages are used. For SOAP services, a Web Service Definition Language (WSDL) document is used to describe the service [3]. For REST services, a variety of modeling languages are available to developers which provide expanded functionality compared to the traditional WSDL document. Three popular modeling languages for REST are Swagger, RESTful API Modeling Language (RAML), and API Blueprint [20]. Each of these modeling languages provide functionality such as code generation and document generation which is beyond the core functionality of a modeling language. Each of the modeling languages use a different language format as demonstrated in Table 2.2.

Table 2.2: Modeling Language Formats

Modeling Language	Service Type	Language Format
API Blueprint [21]	REST	Markdown
RAML [22]	REST	YAML
Swagger [23]	REST	JSON
WSDL [3]	SOAP	XML

2.2 Cloud Computing

The concept of computing has changed dramatically over the past decade. Computing has seen a shift from local processing and supercomputing towards computing as a service with off-the-shelf hardware [24]. The research community has accepted this type of computing to be referred to as cloud computing. This is further defined by the National Institute of Standards and Technology (NIST) as “a model for enabling ubiquitous, convenient, on-

demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [25]. Cloud computing provides developers with tools to provide control and flexibility of their instances. Moving from traditional computing to cloud computing allows for some benefits to be realized for the developers [24]:

1. The illusion of infinite computing resources.
2. The elimination of an up-front commitment.
3. The ability to pay for the use of computing resources.

Traditionally, when having software front facing the public, it required a significant upfront cost in the form of hardware. The developers were required to predict the amount of traffic their software would generate and ensure that enough hardware is available to meet these needs. If the estimate is too conservative, then the user experience would be slowed down or halted all together. If the estimate is too liberal on the hand, there would be wasted resources, which means wasted money. Cloud computing eliminates these concerns by allowing developers to provision as many resources as they need at any given time while only paying for what they use [26].

2.2.1 Service Models

Cloud computing is defined as providing developers resources in three forms (Figure 2.3). The three service models are Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) [25]. Developers are provided cloud computing resources through these three service models which are built on top of physical hardware.

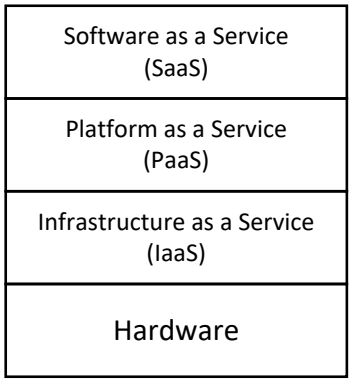

Figure 2.3: Cloud Architecture

In IaaS developers are provided a connection to a Virtual Machine (VM) on the specified hardware. This allows the developers to have complete control of their machine, storage, and networking components. In this case the developer is charged based on how many VMs they have provisioned [27]. Current cloud providers which provide IaaS include Amazon Web Services (AWS) and Microsoft Azure [28, 29].

PaaS provides developers with a platform to build their applications on. This allows the developers to focus on their application logic instead of worrying about the underlying architectures and technologies that are running their application [30]. Current providers of PaaS include Microsoft Azure and Google App Engine [29, 31].

The last service model is SaaS which provides developers a platform to deploy subscription based services to users. This allows users to access the application and information across the network. Users only pay based on their usage of the software [27]. A popular provider of a SaaS service model is Salesforce [32].

2.2.2 Deployment Models

The cloud computing services can be provided to developers through four different types of deployment. The four deployment models are the private cloud, community cloud, public cloud, and hybrid cloud [25]. The choice of deployment model depends on the level of

security and assurance a developer requires [33]. The public cloud offers its services via a third-party platform while the private cloud offers its services within the organization. The primary differences in these cases is that in the private cloud the hardware is within control of the organization. The community cloud is a private cloud managed and used by a group of organizations. The community cloud is similar to the private cloud where the organizations share concerns such as privacy. The last deployment model is the hybrid cloud, which is a combination of two or more deployment models [34].

2.2.3 Scalability

The amount of traffic on an application is not constant, at different times there may be an increased or decreased number of users. Cloud computing provides scalability to address this concern where developers can either scale up or down their applications. This way the amount of idle resources is minimized therefore maximizing profits.

Cloud platforms provide two types of scalability for applications. The first type of scalability is known as vertical scaling or scaling up the application. This type of scaling increases the capabilities of a single machine. The changes can be to add more CPU's or change the type of CPU, more or faster memory, and increased storage capacity. For a change to take effect the system is required to reboot [35]. The second type of scalability is known as horizontal scaling or scaling out. Horizontal scaling involves provisioning new instances to balance the load across [35]. In this case, some sort of load balancer is required to distribute the load across all of the available instances. This type of scaling is typically done through launching another VM with the same image/snapshot as the other instances.

2.3 API Management

As mentioned earlier, API and web service management has been a field which has seen a surge of growth within recent years with many new platforms emerging. These platforms have the primary focus of providing both developers a deployment platform between their service and users, and a consistent endpoint for users to execute services in a known way. This is beneficial to service providers as it limits the amount of work required to get their service in developer's hands and provides a platform in which they can charge for their service.

From a service provider's perspective, there are different key features each API platform aims to address. A set of common features addressed on multiple API platforms are as follows [36]:

1. Authentication: Authentication through authorization methods such as using OAuth or an API key.
2. Tiered Pricing: Multiple pricing plans for developers which provide different feature sets like daily request limits.
3. Enterprise Plans: Customized plans for enterprises which provide a guaranteed level of availability
4. Request Limiting: The ability to limit the number of requests generated through each user.
5. Usage Metrics: Provides the service provider usage metrics of the users on each of their services.
6. API Versions: Supports multiple versions of the service to be used by developers.
7. Documentation: Provides a developer portal with generated documentation on the services.

Each API management platform provides at least one of the listed features. These features can be classified into four categories. These categories are security, pricing plans, lifecycle control, and other features [36].

Even through there are many API management platforms by different developers providing different features, they can all fall under similar architecture categories. These two categories are implementation through a proxy and implementation through an agent [37]. Each implementation type focuses on different feature types for providers or consumers.

An API management platform implemented as a proxy sits between the service provider and the service consumer. All traffic is routed through the proxy between the provider and the consumer [37]. This type of API management platform allows for benefits to be realized by both the provider and consumer. For a service provider, it allows for discoverability, a business model, and separation from service consumers. With a provider's service on the platforms ecosystem, it becomes exposed to potentially more developers to use their service. This also opens up the provider to monetizing their service on the platform. All of the required infrastructure is in place to sell their service instead of having to develop it from scratch. Lastly service consumers are not directly interacting with the provider's service. This protects their services from direct attacks and allows for usage metrics to be observed through the platform. For a developer, a proxy API management platform provides a consistent way to execute services. This allows developers to have 1 API key with the platform which links to multiple API keys.

An API management platform implemented as an agent integrates with the service providers service. This integration allows providers to implement core features such as authentication, usage metrics, and limiting users traffic [37]. This benefits the service provider by providing core functionality for monetizing and managing a service. For a

service consumer, this type of management platform is beneficial when compared to a proxy as it doesn't introduce latency by being routed through a proxy. A hybrid of these two implementations can also be provided by API management platforms [37].

2.4 Software Fault-Tolerance

It is important for software to be robust in the face of failure. Software fault-tolerance is the field in which software is able to detect failures and recover from them [38]. Without actively preparing for faults and developing robust software, there is no guarantee of successful operation.

This framework focuses on the reliability and availability of using web services available on the Internet. Reliability in regards to web services, refers to the probability or likelihood of successful operations when executing a web service. The availability of a web service is the probability of the service successfully responding to a service consumers request [39]. To this end, common fault-tolerance techniques have been explored which incorporate alternatives.

2.4.1 Recovery Blocks

The recovery block technique was first proposed by Horning et al. in 1974 [40]. This was later implemented by Randell in 1975 [41]. The design behind recovery blocks is that the software system can be broken down into a set of distinct fault-tolerant recovery blocks. A specification would be developed for each of the blocks. Multiple versions of software would be developed which conform to each of the blocks specifications. One of the implementations is chosen as the primary version to be executed, where the others versions are left as alternatives. After execution of a block, an acceptance test is applied by a controller to ensure the correctness of the results from the block. If the results are deemed

unacceptable, the state of the system is returned to a checkpoint before execution on the block. An alternative is selected and executed. This continues in a sequential process until a successful execution is confirmed.

Recovery block has been adapted and implemented in many different formats. Peng et al. propose an extended recovery block scheme where the recovery block technique is applied to web services with multiple acceptance tests to improve their reliability [42]. Scott et al. proposed a consensus recovery block mechanism which combines concepts from N-Version programming by running multiple versions and selecting successful results based on multiple versions producing the same results, otherwise the highest rated version is used [43].

2.4.2 N-Version Programming

The N-Version Programming technique was first proposed by Elmendorf in 1972 [44]. This was later implemented by Avizienis and Chen in 1978 [45]. This fault-tolerant technique is similar to recovery blocks through the use of alternatives. A specification is developed for the software system. Multiple versions of the software are implemented in different versions. To improve the fault-tolerance of this system, each of these versions can be run on different hardware systems. Each of the versions are executed in parallel to each other. Upon completion of the execution, a decision-making algorithm is used to determine the correct output. This decision-making algorithm varies based on implementation. Examples of possible decision-making algorithms are majority vote and consensus algorithms.

Like recovery blocks, Peng et al. propose an extended n-version programming scheme which incorporates multiple acceptance testing on each of the versions before passing the results to a decision-making algorithm [42]. When developing multiple versions of

software conforming to the same specification, each of the versions can be prone to the same Common Cause Failure (CCF). Dai et al. propose a model to address this concern which involves decomposing the n-version software into logically exclusive components [46].

2.5 Summary

This chapter has presented an overview of important related concepts to this work. This includes important web technologies such as REST through web services, and cloud computing concepts. Software fault-tolerant techniques were explored which used alternatives to handle failures. These fault tolerant techniques have the alternative implementations following a specification for the block. The proposed framework will be presented in the next chapter which has the specification based on the primary service.

Chapter 3

Proposed Framework

This chapter covers a detailed explanation of the main components for the proposed reliability-aware framework. The proposed framework was designed to address issues of reliability through availability when utilizing publicly available services over the Internet. The proposed framework has been incorporated as middleware to allow for users to ensure the availability of their services. This section is broken down into a high level illustrative overview of the framework in Section 3.1. The following section provides an overview of goals this design aims to meet while considering assumptions when designing the framework. A detailed breakdown of the primary components is explored in Section 3.3.

3.1 High Level Overview

An overview of the proposed framework is presented in Figure 3.1. This figure shows a high-level interaction of the components in the framework. This shows how users interact indirectly with web services through the proposed framework. A detailed breakdown of each component is explored in the following sections.

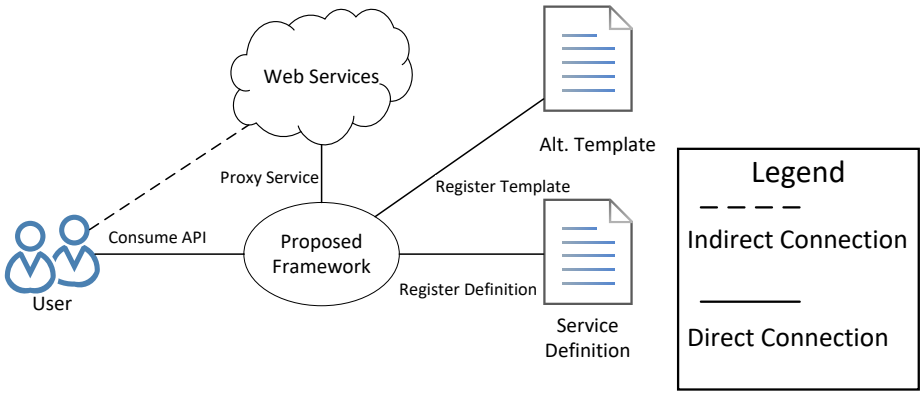

Figure 3.1: High Level Architecture

3.2 Design Goals and Assumptions

This framework was designed to address the issue of reliability when introducing third party off-site services. This is performed through improving the availability of the service functionality. The availability of the web services used can only be guaranteed through a service agreement between the developer and service provider. Realistically 100% uptime cannot be guaranteed for every service used as more points of failure are introduced when moving computation off site. The primary goal of this framework is to assist developers in minimizing the downtime of their software by keeping it functional during possible downtime of required web services. This framework is based off the idea of a recovery block mechanism [47]. Similar to how reliability is ensured through alternative recovery blocks, availability can be ensured through distributed alternatives from different service providers. This is performed by first determining if a failure has occurred between the software and a service. If a failure has occurred a pre-selected alternative is executed instead. The following assumptions are considered:

- The first important design concern is allowing developers to easily incorporate the framework with their software. Instead of having the developers having their services conform to a template or abstract template, the primary service selected should require no additional effort. This will allow developers to easily transition to the framework with no added availability requiring minimal effort. Only the core results can be mapped as the differing services conform to different specifications.
- Services are either developed as REST or SOAP services. This framework is designed to only focus on RESTful services due to the ease of use and the popularity in industry for RESTful web services. When inspected services available on an

online service directory, there are four times more RESTful web services publicly available compared to SOAP services [48].

- Web services typically only expose their interface through a definition file and not the systems implementation. This introduces the concern that the primary and alternative web services selected share a dependence on another service on the Internet. If a service that both the primary and alternative service rely on goes down, then the availability of both the primary and alternative service are affected. Independence of services is a separate research area [49] and therefor it is not the focus of this research. For the purpose of this study, it is assumed that the services are independent of each other.
- API management platforms are commonly being employed by both service providers and service consumers. As the framework is to be positioned between the service provider and service consumer, it would be ideal for the framework to exist as an extension to a proxy API management platform. This level of integration requires full access to the code of an API management platform. For this work a sample API management platform is developed with base functionalities, which is referred to as the server app in this thesis.
- As RESTful web services are flexible with data types, there are a variety of data types available when working with services. The service can provide interaction with text, structured data, and even image files. For simplicity of this study, only services that work with semi-structured data types will be considered. Although the framework design can be expanded to consider other types as future work.

- Interacting with most services usually requires some level of authentication. This is done for operations such as limiting usage, pricing tiers, and usage metrics. Two common types of authentication are OAuth and API Keys [50]. OAuth is typically paired with services that interact with users and allow for third-party clients to work with that data. Users permit services to only have access to certain resources [51]. API Keys are different than OAuth by only providing a key which authenticates the user attempting to access a resource. This thesis only focuses on plain API Keys compared to OAuth authentication. Services which use OAuth typically don't have alternatives available due to their specialized purpose.
- The last major assumption is that for each of the services tested with this framework, there exists an alternative service which provides a similar if not same solution to the desired web service. This is required for the framework to provide additional availability to software using services if their primary service provider fails.

3.3 Architecture Breakdown

The core of this framework can be broken down in two components. The first component is the client application which is the primary component of the framework. The client application is responsible for monitoring operation of services and intervening if a failure occurs. When a failure occurs, an alternative service is selected and executed. The client application is to be integrated with the API management platform. Having it as a standalone component for this framework allows for testing and flexibility. The server application is the second component of this framework. The server application is the basic implementation of an API management platform like existing platforms in the market which provides discoverability and maintainability of the services and configurations for

the client application. During normal operation, a service consumer is only interacting with the client application. A detailed breakdown of these two components and their primary operations are explored in the following sections.

3.3.1 Client Application

The client application is the middleware aspect of the framework which is located between the system running the software and the web service being executed. The client application would be normally run on top of the API management platform or locally to address security concerns of the framework. The framework is required to buffer and map results, which means it will have access to the unencrypted data running through it which is required as the data needs to be transformed and sent to an alternative service provider if a failure occurs. An example of the flow of failure occurring can be observed in the sequence diagram, depicted in Figure 3.2

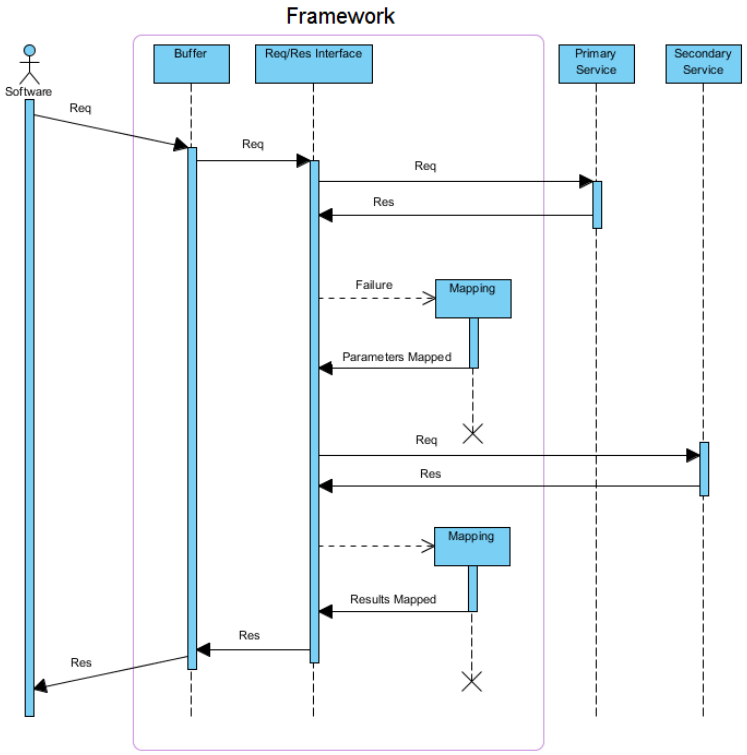

Figure 3.2: Alternative Mapping Sequence Diagram

When a new request is received at the appropriate endpoint, the first step is to buffer the request which allows for failover to occur without a repeat request from the client. Once the request is fully buffered it is sent to the primary service provider specified by the request endpoint. When the response from the service provider is received or not received, the controller needs to determine if a failure has occurred. If a failure has occurred, the request parameters are mapped to the format of the first alternative. After the request is sent to the alternative service provider, if a successful response is received, the appropriate results are mapped back to the specifications of the primary service provider and returned to the service consumer.

3.3.1.1 Failure Detection

The middleware is positioned between the software and desired services which allows both failure detection and transformation of the data to alternative services as required. Two scenarios are considered during operation within the framework. The first scenario considered are error codes within the HTTP requests. As all services in this design are RESTful services, HTTP status codes determine when a failure occurs. If the response status code is not between 200 and 299, alternative mapping is considered. If the response status code is in the 200's, the request is considered successful and allowed to be passed to the software. The other scenario of failure considered for this framework is when no response to the sent request is received. Built-in timeouts in HTTP allow for detection of requests which are not received. If no response is received within the specified timeout, then alternative mapping is applied to the request. If failures are found on alternative services along with the primary service it is passed on to the next alternative service. If no successful responses are received, the original failure information is returned.

3.3.1.2 Service Template

The template was designed to allow developers to specify alternative service providers and how to use them in case of failure with the primary provider. Instead of having each service conform to an abstract template, the primary service is selected as a basis for the template. This allows developers to use the framework without having to modify existing code. The template is saved as a semi-structured data type which describes the mapping of both parameters and results for each alternative service. An example of the template design can be observed in Figure 3.3. This shows how the template specifies the appropriate mapping locations. The first part of the template represents the parameter mapping and the second part the result mapping. The attribute represents the format of the request/response and the value represents the location to find the value in the appropriate alternative response. Multiple alternatives can be specified in the form of a list. For each alternative service the following information must be provided:

- Service Name
- Service Endpoint
- Parameter Mapping
- Result Mapping

Figure 3.3: Template Design

3.3.1.3 Service Mapping

When a failure is detected on the service being executed, alternative services are selected and executed. The data from the primary service must be mapped to the alternative service, as the endpoint information is different. The template selected by the developer as described in the previous section, provides the information to map the parameters and results to the alternative services. Two modules are needed for mapping parameters and results between services. The first is the parsing of the template, as the template for each alternative is in a tree structure, a general tree traversal algorithm is sufficient. Each of the leaves in the tree represents an element which requires its data to be mapped. When a leaf is reached, a mapper module transfers the data. The mapper takes the string value representing the data location in the other service and translates it to a location in the structure. It traverses the data from the other service and replaces the location information with the actual element. The same process is applied to both the parameters and results; the only difference is the direction of the mapping.

3.3.2 Server Application

The server application is the basic API management platform sample which provides discoverability and maintainability aspects for the framework. It allows for developers to share and use existing service templates provided by other developers. Basic search is provided on the added templates via keyword tagging. When the user finds a service and template, they can start using the information with the client and it will save it locally for future usage. Services are not hosted on the server application; they are hosted by their respective providers.

3.3.2.1 Storage

The server application needs to store information about the users, services, and templates. Before a developer can execute a desired service, it needs to be registered within the framework. The developer specifies the appropriate service to register on the server application. As all the data used is either text or binary, a database is used to store the information. Ideally a NoSQL database is employed which allows for easier scalability. A separation of the data within the database ensures privacy of the user information.

3.3.2.2 Service and Template Registration

Before a service or template can be selected, it must first be registered by a user. This is due to services being referenced by a unique identifier in the framework. The required information for each service is submitted through a normal web form. The template provided needs to first be validated to ensure that the format conforms to the specifications.

3.3.2.3 Search and Discovery

Each registered service is represented with a unique identifier in the framework. An endpoint is represented through this unique identifier. Discovery of this service through

only its unique identifier on its own is not sufficient. To allow for effective searching on the services, tags are applied to each service. The user can then specify key terms which describe the service they are looking for. Templates are discovered through the service they are linked to. A short description by the developer is attached with each template so the appropriate template can be selected.

3.4 Summary

This chapter presented the proposed framework for service reliability and availability of public web RESTful web services through a recovery block mechanism. Developers are able to provide alternative services to their desired service which allow them to ensure their software remains operational if a failure occurs on the primary provider. With this design, the primary provider is selected as a basis for a template which allows for easy adoption of the framework. The template allows for developers to easily use alternative service providers. Upon failure, the parameters and results are mapped to and from appropriate services to the structure of the primary service to allow for consistent interaction.

The next chapter describes the implementation of a prototype based on the design of the proposed framework.

.

Chapter 4

Prototype Implementation

This chapter describes the implementation of a prototype based on the proposed design. Detailed documentation on core components and algorithms that are implemented will be covered in this chapter. First in Section 4.1 it will cover an overview of the prototype with detailed information on various components implemented from the design. This includes hardware information selected that hosts the proposed framework and technologies used in the implementation. Next in Section 4.2, advantages and limitations of the programming language selected will be discussed.

4.1 Architecture

The implementation of the design integrates two different platforms. The client and server applications are implemented on the cloud, which allows for the system to be easily scaled to meet a heavy load. The client platform provides the improved availability through mapping requests to alternative services when a failure occurs with the primary service. The server platform demonstrates how the recovery mechanism could be incorporated with an API management platform. This allows developers to discover and share configurations for alternative services. The interaction and high level implementation of these two platforms is depicted in Figure 4.1

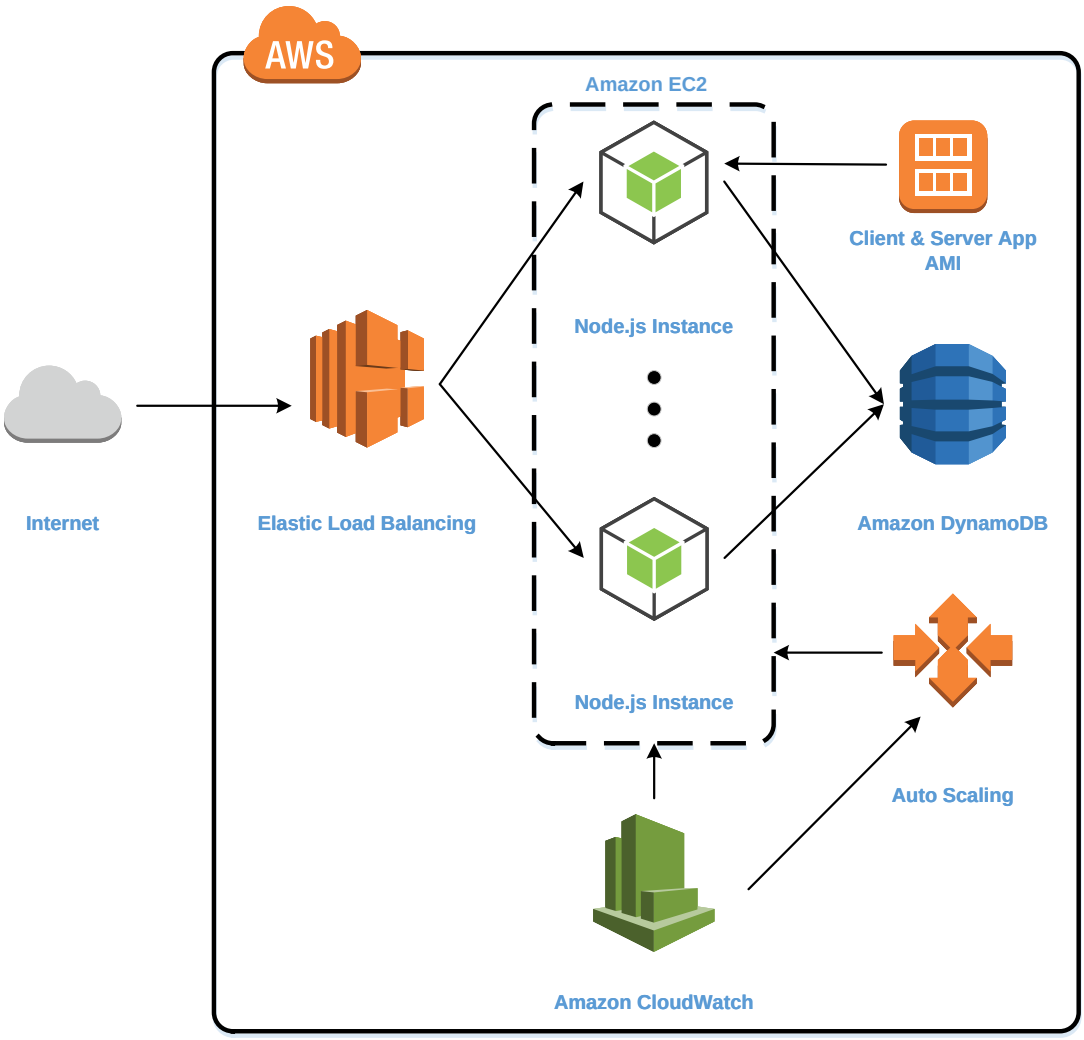

Figure 4.1: High Level Implementation Architecture

4.1.1 Overview

During normal operation, the developer will only be interacting with the client platform. The client platform acts as a middleware between the developers and services they are invoking with their software. Requests are first received by the AWS Elastic Load Balancer (ELB). The ELB is responsible for distributing the service traffic among many NodeJS instances. These node instances are independent of each other to allow for scalability. This ensures that the additional latency by having the framework in-between the service provider and service consumer remains minimal under heavier loads. These instances can be an

instance of the server application or the client application. This allows for web traffic to remain separated from the service traffic. Each of these node instances store configuration data temporarily and connect to the DynamoDB database which provides persistent storage of the configurations. This includes service information and template configurations for those services.

Instead of connecting directly to a service, the developer connects to the service through the client platform. The endpoint on the client platform is specified by a unique service identifier. Each of node workers keep a copy of the service endpoint information stored in memory for faster access after initial requests. If the specified endpoint exists, the request is routed to the appropriate service. If the endpoint doesn't exist, the node worker attempts to discover it on DynamoDB database. If a failure is returned from the service, the node worker attempts to map the request to an alternative service if an alternative service configuration is provided. Parameters and results are mapped based on the provided template, which is formatted in JSON.

Developers are able to discover existing configurations and upload their configurations through the server platform. This platform is a basic proxy API management platform which provides developers a selection of registered services. This platform was implemented for this prototype to allow users to search for and discover alternative configurations on their desired services as if it was a normal API management platform.

4.1.2 Hosting Platform

The application was chosen to be hosted on a cloud platform due to the resources available and ease of scalability of applications. Both the client and service applications are deployed through virtualization on the Infrastructure as a Service (IaaS) through Amazon Web

Services (AWS). This cloud platform was chosen over others due to their vast feature set and catalog of software solutions available for IaaS. AWS provides easy instance configuration and deployment on their cloud platform. With their powerful dashboard, it limits the amount of time required to set up highly scalable builds with affordable pay per use options.

Four services provided through AWS were incorporated to implement the framework. The first is Amazon's Elastic Cloud Compute (EC2) service. This provides easy virtualization on a wide variety of hardware and distributions. The Elastic Load Balancer (ELB) service distributes incoming traffic among the available EC2 instances. To allow for scalability of the EC2 instances, Amazon's CloudWatch is selected. CloudWatch allows for alarms to be configured which monitor the EC2 instances. These alarms are configured with rules which allow for more or less EC2 instances to be provisioned. The last service used is Amazon's highly scalable NoSQL database implementation DynamoDB.

4.1.3 Elastic Cloud Compute

Elastic Cloud Compute (EC2) provides hardware virtualization on a variety of hardware and distributions provided by Amazon. This provides a solid IaaS development platform where new instances can be provisioned within minutes. The distribution selected for the server implementation was Ubuntu 14.04 over Amazons primary distribution. This distribution was chosen due to the applications being developed locally first on this distribution before deployment to the cloud. All of the EC2 instances were implemented on the t2.medium hardware. This type of instance specifications is depicted in Table 4.1.

Table 4.1: AWS Instance Specifications

Instance Type	No. of vCPU	CPU	CPU Clock Rate	Memory
t2.micro	1	Intel(R) Xeon(R) E5-2676 v3	2.40GHz	1 GB
t2.medium	2	Intel(R) Xeon(R) E5-2676 v3	2.40GHz	4 GB
t2.large	2	Intel(R) Xeon(R) E5-2676 v3	2.40GHz	8 GB

4.1.4 Elastic Load Balancer

The Elastic Load Balancer (ELB) provided by Amazon allows for incoming traffic to be distributed evenly among a set of instances. In this case, it balances the traffic coming on the NodeJS web servers. The health of the running EC2 instances connected to the ELB are monitored to ensure that the ELB is only sending new requests to EC2 instances it knows are operational. If the instance is non-responsive, it does not receive new requests and is set for termination. The ELB distributes the incoming requests to all of the active instances. If each instance is saturated with traffic, new EC2 instances are set to automatically be provisioned to handle the influx of traffic. This also allows for instances not being utilized to be shut down.

4.1.5 CloudWatch

Amazon CloudWatch provides monitoring of resources on AWS. For this framework, CloudWatch is exclusively used with EC2 through cloud alarms. Cloud alarms are configured to monitor a specific metric on a specified resource. These alarms are configured in this framework on the EC2 instances to monitor for CPU usage, network usage, and memory usage. The threshold for these alarms is configured to 70% total usage

on a 5-minute interval. If this threshold is passed, an alarm is raised. The scalability rules for the EC2 instances are configured to increase the instance count when an alarm is raised.

4.1.6 DynamoDB

Amazon provides an easy to setup and maintain NoSQL database on their cloud platform. DynamoDB is a key value pair NoSQL database which allows for an extra sorting key and indexing of other attributes. Provisioning of resources is automatically handled through this service. A well documented API is provided with SDKs available in the most common programming languages. These SDKs support a variety of Create Read Write (CRUD) operations on the data sets. For this framework two tables are implemented. The first table contains the service information (Table 4.2). This contains information required for resolving the service name within the framework to the actual service on the Internet. The second table contains the Template information (Table 4.3). This table contains the actual template for performing the mapping operations to alternative services. Along with the template, it also contains a description of the functional operations of the template and which service it belongs to.

Table 4.2: Database Service Fields

Value	**Description**
Service	Unique name for service endpoint.
Host	Host address via DNS or IP Address.
Port	Host port.
Description	Brief description of the service.
Tags	Tags which can be queried for to find service.

Table 4.3: Database Template Fields

Value	**Description**
Name	Name of the template.
Description	Brief description of which endpoint the template works with and the alternatives used.
Service	Name of the service which the template belongs to.
Template	JSON Template containing alternatives.

4.2 Implementation Language

As mentioned in the prior sections, the client and server application for the prototype were implemented in NodeJS. This language has seen a wide surge in popularity in recent years due to its flexibility and vast availability of open source modules. For the prototype, it was chosen due to its simplicity, asynchronously in design, and how it manages incoming HTTP traffic. The most common data form used in node is data streams. In the popular ExpressJS module in node, incoming data can be streamed through middleware which allows for operations to be performed on the data easily before handling the requests normally. This is invaluable to the implementation as it allows for easy manipulation of the data without major implementation.

4.3 Summary

This chapter has presented the implementation of a prototype based on the proposed design from the previous section. The prototype implements the core features described in the design with an emphasis on the scalability of the components. Scalability of the client platform was a crucial focus as the framework needs to be able to handle the typical load

placed on services. The components were built in NodeJS and deployed on Linux instances online through AWS EC2 instances.

Chapter 5

Evaluation and Results

This chapter describes and presents the evaluation of the implemented prototype based on the proposed framework. The framework was evaluated on its performance, scalability, and availability. The performance was evaluated based on the introduced latency through the framework, compared with the latency of directly executing the service without the framework. This is discussed in detail in Section 5.1. In Section 5.2, the scalability of the framework is evaluated through measuring the latency of the framework when increasing the amount of traffic through the framework. Section 5.3 evaluates the availability of the framework by examining scenarios of failure within the framework. For each of the evaluations, the design and methodologies behind the experiment are provided.

5.1 Performance

This evaluation tests the proposed framework based on its introduced latency when compared with directly using a service. The response time of executing services is evaluated and compared when incorporating the framework.

5.1.1 Methodology

The introduced latency when executing services through the framework compared to directly executing a service is an important metric. The most common scenario when executing a service through the framework is no failure occurring and the primary service just being used. It is important that the introduced latency is as minimal as possible. To

determine the amount of introduced latency through the framework, multiple scenarios must be tested and compared. For all of the tests the following two scenarios are considered:

1. Directly executing the service (No Framework)
2. Executing the service through the framework

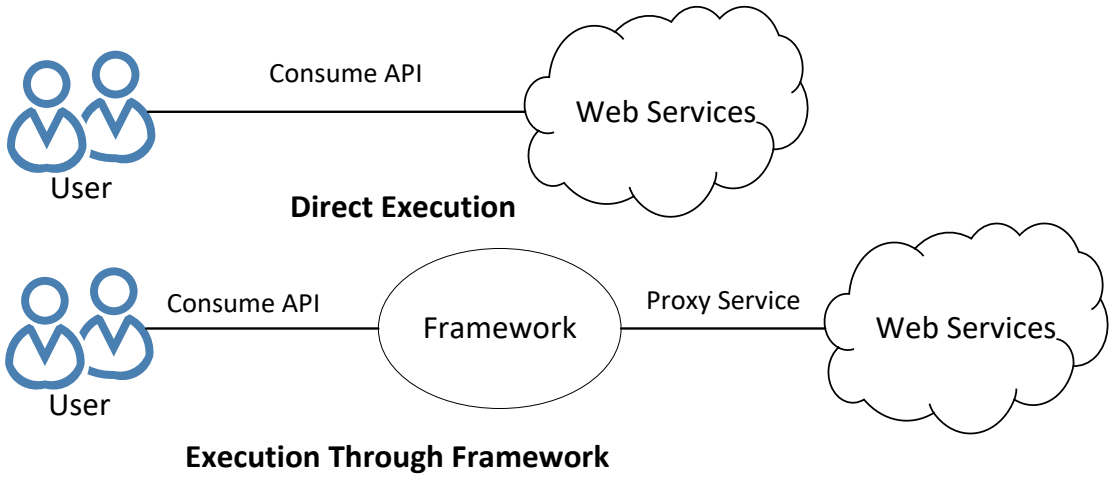

Figure 5.1: Performance Test Setup

When executing the service through the framework the user can have local configuration data stored in their client application about how to access their desired service. The user only needs the name of the service and the name of the template. If the endpoint information is not stored locally within the client application, it first needs to be retrieved from the DynamoDB database online.

These scenarios are tested on multiple services. Each of these services transmit data of varying sizes. Two services are evaluated which have a transfer size of 2,472 bytes and 50,713 bytes. The second service was selected which has a transfer size 20 times larger than the other for varying transfer sizes.

5.1.2 Experimental Setup

The test environment was implemented on the cloud with the application to minimize potential variances introduced by local network conditions. The hardware configuration on AWS EC2 was a t2.medium instance. The specifications for all of the used instances is depicted in Table 4.1. With this configuration, the response time was measured using a Java tool called JMeter. This tool allows for precise round trip measurements on HTTP requests made through the framework.

5.1.3 Results

Tables 5.1 and 5.2 show the average response time for executing the services. These tests were performed a total of 100 times to ensure the accuracy of the results and minimize any deviations caused by the network. Each of the results are shown in milliseconds.

Table 5.1: Small Transfer Size Performance Results / ms

	Direct	**Indirect**
Min	18	31
Max	386	436
Average	38	142

Table 5.2: Large Transfer Size Performance Results / ms

	Direct	**Indirect**
Min	511	683
Max	852	1007
Average	643	821

5.1.4 Analysis

The first clear point that can be made from the performance test is that directly executing the service results in the lowest response time on average. On average executing the service through the framework instead of directly connecting to the service introduces an added response time of 104 ms for smaller transfer rate. On the larger transfer rate test, there was an additional 178 ms response time. The variance can be attributed to the size of the request. The larger request required more time to buffer before sending it to the service.

5.2 Scalability

This evaluation tests the proposed framework based on its ability to scale to meet heavier loads. This is important as the framework needs to handle all the load without failure and introducing minimal latency.

5.2.1 Methodology

This framework is intended to improve the reliability of developer's software through ensuring the availability of services they depend on. This stresses the importance that the framework is able to handle all the incoming requests without failing or introducing a significant latency. To evaluate the scalability of the framework, load testing needs to be performed where sample traffic is generated to analyze the affect it has on the application. As the framework is run in between the client and the service, it is important to isolate the scalability to the framework. This is important to ensure the results are only affected from the scalability of the framework and not the scalability of the service being executed. If the service selected is saturated, failures and increased latency will occur. To isolate the frameworks scalability from the scalability of the services used, sample traffic is generated on another service while measuring the desired service at a controlled rate. This way the

amount of traffic going to the desired service is minimal so it doesn't saturate that service. The saturation will occur in the framework from the sample traffic which will affect the response rate of the control service. Like the previous evaluation, the scalability is tested on multiple services with small and large transfer rates.

5.2.2 Design

Testing the scalability of the client application cannot be performed through a single machine due to the large amount of traffic being generated. To ensure the accuracy of the results, the sample traffic also needs to be isolated from the control traffic. JMeter provides this functionality through its distributed testing setup. This allows for a large amount of traffic to be generated, but requires multiple machines. For distributed load testing in JMeter, there are three different types of instances:

- Master: The instance that controls the test and stores the results. This is the control instance for the entire test suite.
- Slave: The instance that generates traffic for the Master and sends results back.
- Target: The server to be load tested. In this case, it's the framework.

For this load test, one slave instance will be running the control test for the desired service. This instance will be generating requests at a constant interval to measure the load effects. There will be N slave instances generating traffic towards the framework. Each instance is limited by the capabilities of the machine. The limitations include processing power, network bandwidth, and memory of the machine generating traffic. To generate a large amount of traffic, more than one instance is necessary.

5.2.3 Experimental Setup

Implementation of the client application is done on a cloud platform to allow for ease of scaling. This is due to horizontal scaling being provided natively through most cloud platforms. The client application for this test were hosted on AWS. This cloud platform provides scalability through its CloudWatch interface and policies. Distribution among the instances is done through their Auto Load Balancer (ALB) tool.

The testing platform was implemented on a variety of different EC2 instances. The slave instance running the control service tests was run on a t2.micro machine. This was due to the low requirements from the minimal number of requests required for the control signal. The heavy traffic generation slave instances and the master were running on a t2.large instance. This is due to a higher memory requirement for storing the response data.

The client application was configured for auto scaling based on the incoming traffic using CloudWatch and ELB. Each instance was running on a t2.medium instance. To access the application the user would connect through the ELB which directs them to one of the available instances. When one of the instances is at its limit a new instance is provisioned and made available to the ELB.

5.2.4 Results

The figures below show the average response time for executing the service with varying user loads. The number of users represent how many concurrent users were sending requests through the framework. The response times in milliseconds are the average of 10 separate tests. Each of the data points from each test represents the average of the relative number of users.

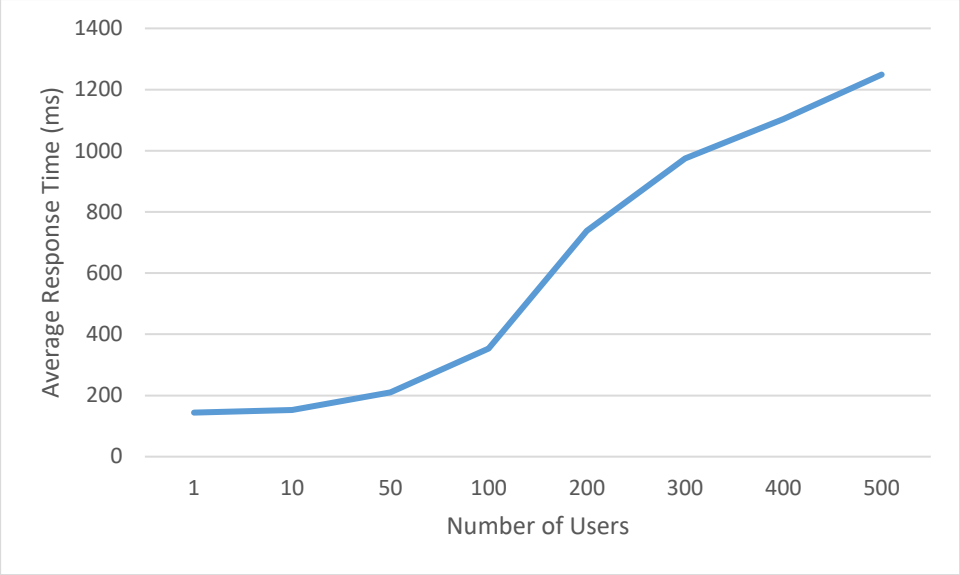

Figure 5.2: Small Transfer Size with no Scaling Results

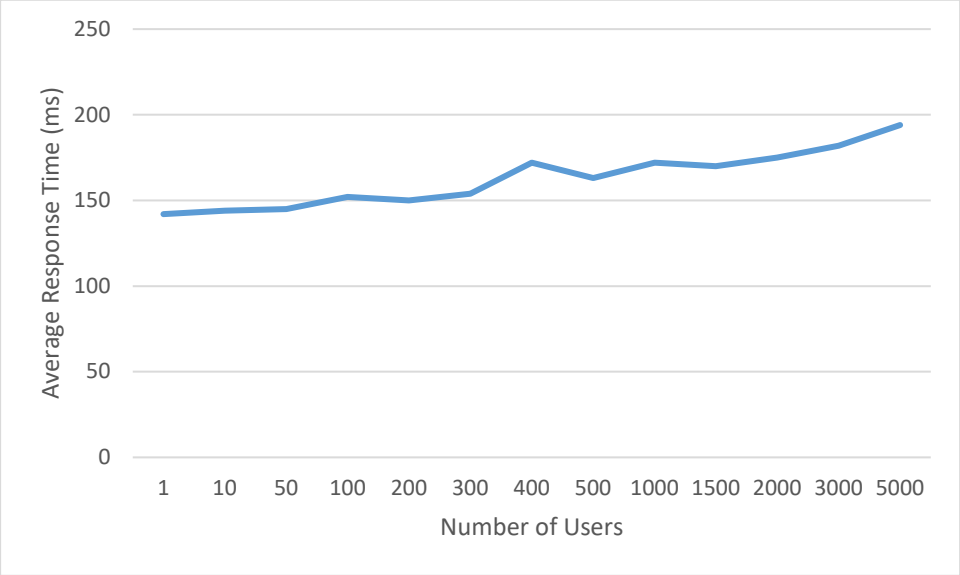

Figure 5.3: Small Transfer Size with Scaling Results

5.2.5 Analysis

Analyzing the results there is a clear distinction between the two different test scenarios. When comparing the test cases where only a single node instance is running, the larger request size tests resulted in significantly higher response times at a much lower user count.

Figure 5.2 takes double the amount of time at around 100 users while Figure 5.4 takes double the amount of time at around 10 users. These differences are due to the capabilities of the machine running the instance. The larger requests require more memory and time to buffer the request. This shows that the implementation isn't as effective at handling larger requests under heavy loads.

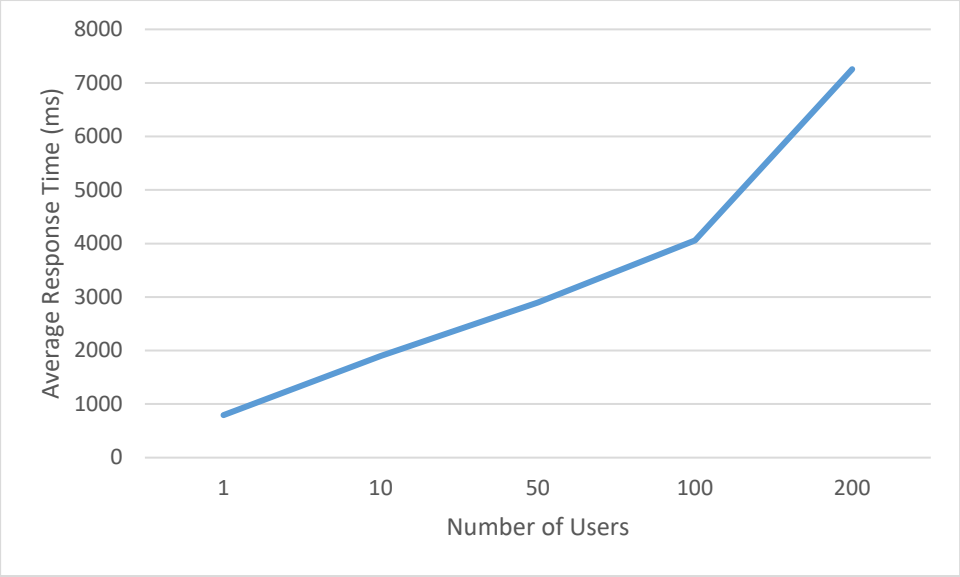

Figure 5.4: Large Transfer Size with no Scaling Results

When scalability is enabled for the tests, the results become much more reasonable. The larger requests required more instances to produce reasonable response times under load compared to the smaller requests. In both cases, it shows that response time is within a reasonable range when compared to lower loads.

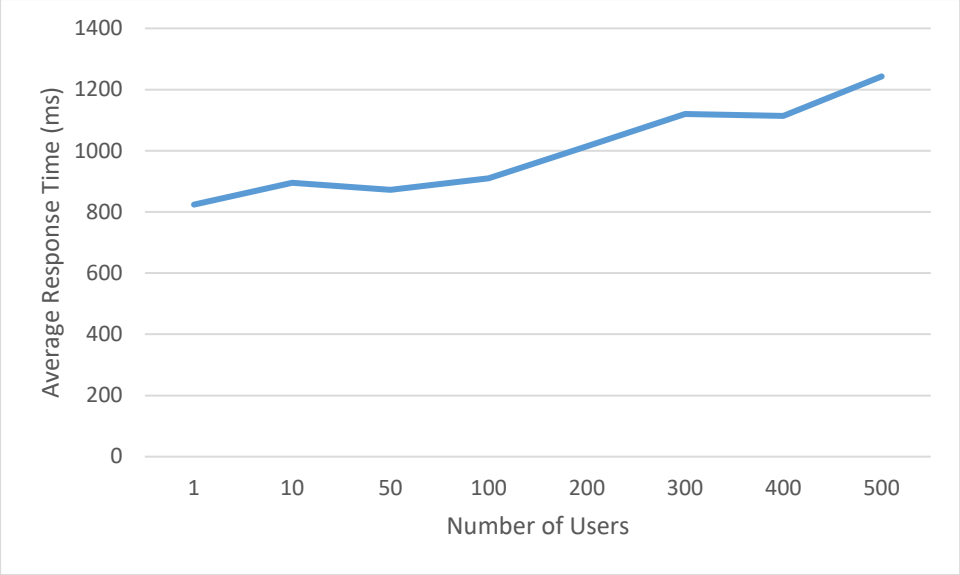

Figure 5.5: Large Transfer Size with Scaling Results

5.3 Availability

This evaluation tests the proposed framework based on its availability to failures within the framework. The framework provides alternative mapping if the failure is on the service providers end. This test is to find out what occurs when failures are generated at different points of execution.

5.3.1 Methodology

As the primary goal of the framework is to provide increased availability of services, the availability of the framework needs to be measured. To measure the availability within the framework, failures are generated at various points and the end results are measured. These failures were generated at both the components and the connections between the components. The following test scenarios are considered:

1. Primary Service Failure
2. Connection to Primary Service Failure
3. All Service Failure

4. Connection to Configuration Database Failure (No Local Config Data)
5. Web Framework Failure
6. Web Framework Connection Failure

5.3.2 Design

Control is required over each of the components and the connections between them to generate the required scenarios. Sample services were created for these test cases and registered within the framework. This allowed for failures to be generated on the service providers end. To generate connection errors, requests sent to the specified services were blocked at their firewall. This would block the requests from reaching their destinations. Failures within each of the components were generated through points within their execution. Each of these requests were run multiple times to ensure the consistency of the results.

5.3.3 Experimental Setup

Implementation of the test platform was implemented on a cloud platform for ease of configuration of the connections. The testing platform was implemented across multiple EC2 instances. All of the instances were run on a t2.micro machine except the framework instances which are running on a t2.medium (Table 4.1). This is due to the performance requirements not being evaluated in this test. This meant that one of the lower tiered machines was used for the instances. The different instances run was the client application and 2 sample services with one of the services considered the primary service.

5.3.4 Results

Table 5.3 shows the results from the described availability test. For each test case the response status is provided and a brief description of the contents of the response. Each of these tests were done 10 times with each of the results being the same.

Table 5.3: Availability Results

Test Case	Response Code	Response Summary
Primary Service Failure	200	Results Received
Connection to Primary Service Failure	200	Results Received
All Service Failure	500	Internal Server Error. Response from Primary Service
Connection to Configuration Database Failure	500	Internal Server Error. Unable to connect to Database
Web Framework Failure	500	Internal Server Error
Web Framework Connection Failure		Time out

5.3.5 Analysis

Whenever an error occurred on the service providers end, the framework successfully repackaged the requests to an alternative service. Whenever an error occurs on either the communication to the service or within the service, the framework is able to return results if another alternative service exists. The error observed when all services were down was the error from the primary service provider. This is due to if no alternatives are available, it should be treated as if no alternatives were used, which returns the appropriate

information from the primary service failure. Errors occurring between the user and the web framework resulted in loss of functionality. This shows that in its current state there exists a single point of failure on the framework as all the services are currently being routed through the framework on the API management platform. In this case, the developer has more control of the failure and can introduce new fault tolerant techniques.

5.4 Summary

This chapter has presented a description of the experimented and evaluation criteria performed on the prototype of the proposed framework. The framework was evaluated based on its performance, scalability, and availability provided. The results from the performance evaluation showed that an average 141 ms latency was introduced through the framework. When under heavy load with scalability techniques introduced, the introduced latency increased minimally. This framework provides an increased availability of the desired services through providing alternatives but also introduces a single point of failure in the system. In the next chapter, conclusions will be drawn from these results and future work will be explored.

Chapter 6

Conclusion and Future Work

In chapter 3, a framework was proposed which incorporates design philosophies from a recovery block mechanism to provide improved reliability of software through improving the availability of services it is dependent on. The design allows for developers to specify alternative services, which can fulfil the requirements of the existing service they are using in case of failure. The service's requests and results were mapped to the specifications of the main service through a simple template, which provides developers the ability to specify alternative services. The framework was implemented in Chapter 4 as a proof of concept using a NodeJS web server to act as a middleware between the desired services and the developer's software, like an API management platform would supply.

A test suite was implemented in the cloud in Chapter 5 to measure overheads introduced from the framework compared to directly interacting with services. The results showed that the framework introduced an average 141 ms latency. Smaller sized payloads on average introduced less latency compared to larger payloads as requests and responses needed to be buffered within the framework. Under load, the framework was able to scale up horizontally to meet the additional demand.

Future work on this framework design and implementation can be done to improve integration and lower the performance cost associated with executing services through it. Incorporation with a popular proxy API management platform would provide a list of existing registered services and user infrastructure. It is the end goal to see this form of alternative recovery blocks implemented with service compositions seamlessly on publicly

available web services. When connecting together multiple services together, the overall availability is the composition of all the services used. This is where failures become more apparent as the overall availability decreases with more added services.

In conclusion, a reliability aware framework was produced which allows for increased reliability through availability. Through evaluation, it is shown that the framework is scalable, though introduces an additional latency compared to directly using the service, at the cost of software fault tolerance.

Bibliography

[1] R. Perrey and M. Lycett, "Service-oriented architecture," *2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings.*, pp. 116-119, 2003.

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

[3] S. Kumar and S. K. Rath, "Performance comparison of SOAP and REST based Web Services for Enterprise Application Integration," *2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 1656-1660, 2015.

[4] L. Richardson and S. Ruby, RESTful web services, O'Reilly Media, Inc, 2007.

[5] "Mashape - Powering API Driven Software," [Online]. Available: http://www.mashape.com.

[6] "API Management | Predictive Analytics | Apigee," [Online]. Available: http://apigee.com.

[7] Y. Raivio, S. Luukkainen and S. Seppala, "Towards Open Telco - Business Models of API Management Providers," *2011 44th Hawaii International Conference on System Sciences*, pp. 1-11, 2011.

[8] B. Zhou, K. Yin, S. Zhang, H. Jiang and A. J. Kavs, "A tree-based reliability model for composite web service with common-cause failures," *International Conference on Grid and Pervasive Computing. Springer Berlin Heidelberg*, pp. 418-429, 2010.

[9] K. Laskey and K. Laskey, "Service oriented architecture," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 1, pp. 101-105, 2009.

[10] T. Erl, Service-oriented architecture: concepts, technology, and design, Pearson Education India, 2005.

[11] H. Haas and A. Brown, "Web services glossary," W3C Working Group Note, 2004.

[12] T.-H. Kuo, C.-H. Chen, H.-Y. Kung and Y.-S. Liao, "Applications of the Web Service Middleware Framework Based on the BPEL," in *IEEE 5th Global Conference on Consumer Electronics*, 2016.

[13] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin and L. Safina, "Microservices: yesterday, today, and tomorrow," arXiv preprint arXiv:1606.04036, 2016.

[14] M. Fowler and J. Lewis, "Microservices," Viittattu, 28, 2015.

[15] C. Ouyanga, E. Verbeek, W. M. v. d. Aalst, S. Breutel, M. Dumas and A. H. t. Hofstede, "Formal semantics and analysis of control flow in WS-BPEL.," *Science of Computer Programming*, vol. 67, no. 2, pp. 162-198, 2007.

[16] G. Alonso, F. Casati, H. Kuno and V. Machiraju, "Web services," Springer Berlin Heidelberg, 2004.

[17] R. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," University of California, 2000.

[18] C. Davis, "What if the web were not RESTful?," in *Proceedings of the Third International Workshop on RESTful Design*, ACM, 2012.

[19] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1," RFC 2616, June 1999.

[20] V. Surwase, "REST API Modeling Languages-A Developer's Perspective," *IJSTE - International Journal of Science Technology & Engineering*, vol. 2, no. 10, pp. 634-637, 2016.

[21] "API Blueprint," [Online]. Available: http://apiblueprint.org/.

[22] "RESTful API Modeling Language (RAML)," [Online]. Available: http://raml.org/.

[23] "Swagger," [Online]. Available: http://swagger.io/.

[24] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," Reliable Adaptive Distributed Systems Laboratory, 2009.

[25] P. Mell and T. Grance, "The NIST definition of cloud computing," 2011.

[26] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation computer systems*, vol. 25, no. 6, pp. 599-616, 2009.

[27] H. Dinh, C. Lee, D. Niyato and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless communications and mobile computing*, vol. 13, no. 18, pp. 1587-1611, 2013.

[28] "Amazon Web Services," [Online]. Available: http://aws.amazon.com/.

[29] "Microsoft Azure," [Online]. Available: http://azure.microsoft.com.

[30] B. P. Rimal, E. Choi and I. Lumb, "A taxonomy and survey of cloud computing systems," in *Fifth International Joint Conference on INC, IMS and IDC*, 2009.

[31] "Google App Engine," [Online]. Available: http://appengine.google.com.

[32] "Salesforce," [Online]. Available: http://www.salesforce.com.

[33] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of network and computer applications,* vol. 34, no. 1, pp. 1-11, 2011.

[34] H. Takabi and a. G.-J. A. James Joshi, "Security and Privacy Challenges in Cloud Computing Environments," *IEEE Security & Privacy,* pp. 24-31, 2010.

[35] L. Vaquero, L. Rodero-Merino and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM Computer Communication Review,* vol. 41, no. 1, pp. 45-52, 2011.

[36] A. G´amez-D´ıaz, P. Fern´andez-Montes and A. Ruiz-Cort´es, "Towards SLA-Driven API Gateways".

[37] G. Psistakis, "API Management tools: How to find the one for you," 2015.

[38] C. Inacio, "Software fault tolerance," Carnegie Mellon University, 1998. [Online]. Available: https://users.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/.

[39] E. M. Maximilien and M. P. Singh, "A framework and ontology for dynamic Web services selection," *IEEE Internet Computing,* vol. 8, no. 5, pp. 84-93, 2004.

[40] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith and B. Randell, "A Program Structure For Error Detection And Recovery," *Operating Systems. Springer Berlin Heidelberg,* pp. 171-187, 1974.

[41] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering,* vol. 1, no. 2, pp. 220-232, 1975.

[42] K.-L. Peng, C.-Y. Huang, P.-H. Wang and C.-J. Hsu, "Enhanced n-version programming and recovery block techniques for web service systems," *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices. ACM*, pp. 11-20, 2014.

[43] K. R. Scott, J. W. Gault and D. F. McAllister, "The consensus recovery block," *Proc. Total System Reliability Symp*, pp. 74-85, 1983.

[44] W. R. Elmendorf, "Fault-tolerant programming," *Proceedings of the 2nd International Symposium on Fault-tolerant Computing (FTCS-2)*, vol. 31, pp. 79-83, 1972.

[45] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, pp. 3-9, 1978.

[46] Y. Dai, M. Xie, K. Poh and S. Ng, "A model for correlated failures in N-version programming," *IIE Transactions*, vol. 36, no. 12, pp. 1183-1192, 2004.

[47] B. Randell and J. Xu, "The Evolution of the Recovery Block Concept," Software Fault Tolerance 1, 1995.

[48] "ProgrammableWeb - APIs, Mashups and the Web as Platform," [Online]. Available: http://www.programmableweb.com/.

[49] P. Mudhar, C. A. Licciardi and R. Minetti, "Service Independence, Service Components and the Network Resource Model," *Workshop on Intelligent Network*, pp. 7-19, 1994.

[50] M. Maleshkova, C. Pedrinaci, J. Domingue, G. Alvaro and I. Martinez, "Using semantics for automating the authentication of web APIs," in *International Semantic Web Conference. Springer Berlin Heidelberg*, 2010.

[51] D. Hardt, "The OAuth 2.0 Authorization Framework," *RFC 6749*, 2012.

Appendix A

Selected Source Code

Two code snippets of the prototype source code have been selected which were discussed in the thesis. A.1 shows the function which is executed when service mapping is invoked and A.2 shows the function which is executed when a new request is received by the framework.

A.1 Result Mapping

```
result: function(alt_results, original_map) {
    var traverse_map = function(node) {
        if(Array.isArray(node)) {
            var intermediary_results = [];
            var intermediary_array = true;
        } else {
            var intermediary_results = {};
            var intermediary_array = false;
        }

        for(var attr in node) {
            if(typeof(node[attr]) === 'object') {
                if(intermediary_array) {
                    intermediary_results.push(traverse_map(node[attr]));
                } else {
                    intermediary_results[attr] = traverse_map(node[attr]);
                }
            } else {
                if(node[attr].indexOf("'") != -1) {
                    intermediary_results[attr] = node[attr].replace(/'/g, "");
                } else {
                    intermediary_results[attr] = parse_map(alt_results,
node[attr]);
                }
            }
        }

        return intermediary_results;
    }

    var result = traverse_map(original_map);
    return result;
```

A.2 Service Execution

```
var execute_service = function(service, res) {
    var service_map = function() {
        //Stores results if original
```

```
        if(service.alt_number == 0 && service.status_code != SERVER_ERROR) {
            service.status_code = proxy.status_code;
            service.res_headers = proxy.headers;
            service.chunks = proxy.chunks;
        }

        //Loads next alternative
        var current_alt = service.template.template[service.alt_number];
        var proxy_service = new Service({'service': current_alt.service});

        //Transfers important parameters
        proxy_service.alt_number = service.alt_number + 1;
        proxy_service.endpoint = current_alt.endpoint;
        proxy_service.headers = service.headers;
        proxy_service.method = service.method;
        proxy_service.template = service.template;

        proxy_service.on('end', function() {
            //Maps Parameters
            if(service.alt_number == 0) {
                proxy_service.endpoint = proxy_service.endpoint + "?" +
querystring.stringify(alt_mapping.param(service.query,
JSON.parse(current_alt.template.param)));
                proxy_service.original = service;
            } else {
                proxy_service.endpoint = proxy_service.endpoint + "?" +
querystring.stringify(alt_mapping.param(service.original.query,
JSON.parse(current_alt.template.param)));
                proxy_service.original = service.original;
            }

            execute_service(proxy_service, res);
        });

        //Returns error unless connection already closed
        proxy_service.on('error', function() {
            console.log(this.error);

            if(!res.finished) {
                res.writeHead(SERVER_ERROR);
                res.write(this.error.stack);
                res.end();
            }
        });

        proxy_service.get_service();
    };

    //Loads auth key if exists
    service.auth();

    var options = {
        host: service.host,
        port: service.port,
        path: "/" + service.endpoint,
        method: service.method,
        headers: service.headers
    };

    var proxy = new Proxy();

    proxy.on('end', function() {
        if(proxy.status_code >= GOOD_REQUEST_LOWER && proxy.status_code <
```

```
GOOD_REQUEST_UPPER) {
            //Successful response but requires result mapping
            if(service.alt_number > 0) {
                //Maps new results
                var current_alt = service.template.template[service.alt_number -
1];
                var alt_results = JSON.parse(proxy.chunks.toString());
                var new_results = alt_mapping.result(alt_results,
JSON.parse(current_alt.template.result));

                //Formats results to be sent
                proxy.chunks = new Buffer(JSON.stringify(new_results), 'utf-8');
                proxy.headers['content-length'] = proxy.chunks.length;
            }

            res.writeHead(proxy.status_code, proxy.headers);
            res.write(proxy.chunks);
            res.end();
        } else {
            if(service.template) {
                //Failed response with template
                if(service.alt_number >= service.template.template.length) {
                    //No more alternatives
                    res.writeHead(service.original.status_code,
service.original.res_headers);
                    res.write(service.original.chunks);
                    res.end();

                } else {
                    service_map();
                }
            } else {
                //Failed response and no template
                res.writeHead(proxy.status_code);
                res.write(proxy.chunks);
                res.end();
            }
        }
    });

    //Returns error unless connection already closed
    proxy.on('error', function() {
        console.log(this.error);

        if(service.template) {
            if(service.alt_number == 0) {
                service.status_code = SERVER_ERROR;
                service.chunks = this.error.stack;
            }

            if(service.alt_number >= service.template.template.length) {
                //No more alternatives
                if(!res.finished) {
                    res.writeHead(service.original.status_code,
service.original.res_headers);
                    res.write(service.original.chunks);
                    res.end();
                }
            } else {
                service_map();
            }
        } else {
            if(!res.finished) {
```

```
                res.writeHead(SERVER_ERROR);
                res.write(this.error.stack);
                res.end();
            }
        }
    });

    proxy.request(options, service);
};
```

Appendix B

Sample Template Format

```
[
  {
     "template": Template Name,
     "alternatives":[
       {
          "template":{
             "result":{
               "primary_result_format":{
                  "primary_res": Cooresponding.Alt.Val
               }
             },
             "param":{
               "alt_param1”: Primary param1,
               "alt_param2”: Primary param2,
             }
          },
          "service": Service Name,
          "endpoint": Service Endpoint
       }
     ]
  }
]
```

Appendix C

Sample Execution

The following contains a sample execution of the framework on an available service. C.1 and C.2 provide the results from geocode services with no modifications applied to the response data. C.3 shows the output when the primary service C.1 fails, and the alternative is used. When the alternative is used, the results are mapped back to the specification of the primary service.

C.1 Primary Service Output – Open Street Map


```
[
  {
    "place_id":"81370563",
    "licence":"Data © OpenStreetMap contributors, ODbL 1.0.
      http:\/\/www.openstreetmap.org\/copyright",
    "osm_type":"way",
    "osm_id":"76812844",
    "boundingbox":[
      "43.8970312",
      "43.8977375",
      "-78.8584115",
      "-78.8575667"
    ],
    "lat":"43.8974669",
    "lon":"-78.8579787521436",
    "display_name":"UOIT, 61, Charles Street, Oshawa, Durham
     Region, Ontario, L1H 8B7, Canada",
    "class":"amenity",
    "type":"university",
    "importance":0.301,
    "icon":"https:\/\/nominatim.openstreetmap.org\/images\/m
     apicons\/education_university.p.20.png"
  }
]
```

C.2 Alternative Service Output – Google Map


```
{
  "results":[
    {
      "address_components":[
        {
          "long_name":"61",
          "short_name":"61",
          "types":[
            "street_number"
          ]
        },
        {
          "long_name":"Charles Street",
          "short_name":"Charles St",
          "types":[
            "route"
          ]
        },
        {
          "long_name":"Central Oshawa",
          "short_name":"Central Oshawa",
          "types":[
            "neighborhood",
            "political"
          ]
        },
        {
          "long_name":"Oshawa",
          "short_name":"Oshawa",
          "types":[
            "locality",
            "political"
          ]
        },
        {
          "long_name":"Durham Regional Municipality",
          "short_name":"Durham Regional Municipality",
          "types":[
            "administrative_area_level_2",
            "political"
          ]
        },
        {
          "long_name":"Ontario",
          "short_name":"ON",
          "types":[
```

```
        "administrative_area_level_1",
        "political"
      ]
    },
    {
      "long_name":"Canada",
      "short_name":"CA",
      "types":[
        "country",
        "political"
      ]
    },
    {
      "long_name":"L1H 4X8",
      "short_name":"L1H 4X8",
      "types":[
        "postal_code"
      ]
    }
  ],
  "formatted_address":"61 Charles St, Oshawa, ON L1H
   4X8, Canada",
  "geometry":{
    "bounds":{
      "northeast":{
        "lat":43.8977399,
        "lng":-78.85750879999999
      },
      "southwest":{
        "lat":43.8970592,
        "lng":-78.8583323
      }
    },
    "location":{
      "lat":43.8973996,
      "lng":-78.8579206
    },
    "location_type":"ROOFTOP",
    "viewport":{
      "northeast":{
        "lat":43.8987485302915,
        "lng":-78.85657156970849
      },
      "southwest":{
        "lat":43.8960505697085,
        "lng":-78.8592695302915
      }
```

```
        }
      },
      "place_id":"ChIJpxOVLdwc1YkRQd9VfiIdAOQ",
      "types":[
        "premise"
      ]
    }
  ],
  "status":"OK"
}
```


C.3 Mapped Alternative Output


```
[
  {
    "boundingbox":[
      43.8970592,
      43.8977399,
      -78.8583323,
      -78.85750879999999
    ],
    "lat":43.8973996,
    "lon":-78.8579206,
    "display_name":"61 Charles St, Oshawa, ON L1H 4X8,
     Canada"
  }
]
```