Web Scaling Frameworks

Building scalable, high-performance, portable and interoperable Web Services for the Cloud

Thomas Fankhauser

Thesis submitted in partial fulfilment of the requirements
of the University of the West of Scotland
for the award of Doctor of Philosophy

In collaboration with Stuttgart Media University

April 2016

Table of Contents

List of Figures

List of Tables

Abstract

With the emerging global trends of the social web, smart health and the Internet of Things, the Internet has become the epicenter of modern life. Through mobile smart devices and sensors, ubiquitous communication is omnipresent. Day by day, users and machines contribute inconceivable amounts of new data while operating on aggregations of existing data. Cloud computing enables platforms to deal with this enormous amount of data by offering resources on a pay per use base. This allows cloud customers to create products potentially operating world-wide with no upfront investments. The efficient utilisation of cloud computing resources, however, introduces major challenges. Web applications need to cope with complex infrastructure issues such as the distribution of messages and data between multiple machines, the dynamic provisioning of machines, the selection of the best cloud providers and services, resilient monitoring and orchestration of this manifold system. Current solutions to manage these complex tasks are typically hand crafted for specific cloud providers. A further challenge is the optimisation of data for cloud processing. Typically, web applications incorporate partial caches that store a subset of resources temporarily to improve performance. However, it is challenging to select the best items to cache as modern web applications tend to be user content driven and thereby exhibit huge numbers of resources that are custom made for users of equal importance.This thesis deals with the aforementioned challenges by proposing and investigating the following novel contributions. Firstly, in order to segregate the application and hosting logic, a novel class of Web Scaling Frameworks (WSFs) is presented with a conceptual architecture design taking over the complex matter of scaling. To optimise the distribution of the work among multiple components in WSFs, a novel request flow scheme is then designed, modelled and evaluated with real world data. Furthermore, as a modern alternative to cache eviction, an efficient resource dependency processing approach using cloud storage is analysed, modelled and evaluated. Finally, in order to overcome special customisations for cloud providers, a portable and interoperable prototypical implementation of a WSF is created and evaluated with a model determining the processing cost and storage space requirements in contrast to a traditional processing approach. The work in this thesis largely contributes to the generation of a new level of abstraction for more scalable cloud deployment and hosting. With the adoption of the proposed class of frameworks, a variety of WSF implementations can contribute to enabling future applications to focus on enhanced application logic as opposed to deploying and hosting logic. This enhanced focus in turn can be used to create a series of new generation smart services helping to unravel the true power of cloud computing.

Declaration

The research presented in this thesis was carried out by the undersigned. No part of the research has been submitted in support of an application for another degree or qualification at this or another university.

Signed:  Date: 01.04.2016

Acknowledgements

Firstly, I would like to thank my supervisors Qi Wang, Ansgar Gerlicher and Christos Grecos for their brilliant supervision efforts. Whenever I needed critical feedback or advice on the next interesting paths to explore, they provided me with their helpful guidance and experience. The positive and inspiring atmosphere during our meetings and telcos was incredibly motivating and gave me the trust to eagerly continue with the project.
Secondly, I want to express my deepest gratitude towards my family, especially my future wife Anna. I'm extremely thankful for giving me the time and space to study. I promise, from now on we can start doing something on weekends and I will not claim extra luggage for a cluster of computers in our future holidays.
Further, I want to give special thanks to Walter Kriha, who, with his inspiring lectures, projects and supervision of my bachelor and master thesis, deserves a great deal of gratitude in supporting and motivating me for the project.
Last but not least, I want to thank my current and former colleagues for their inspiration and support, especially Volker Tietz with whom I developed the initial motivation for this project.

List of Publications

Peer-reviewed Journal Papers

1. Fankhauser, T., Q. Wang, A. Gerlicher, C. Grecos and Wang, X. (2015). 'Web Scaling Frameworks for Web Services in the Cloud'. In: *IEEE Transactions on Services Computing*. In press (accepted), Jul. 2015. (DOI: 10.1109/TSC.2015.2454272 for Early Access)
2. Fankhauser, T., Q. Wang, A. Gerlicher and C. Grecos (2016). 'Resource Dependency Processing in Web Scaling Frameworks'. In: *IEEE Transactions on Services Computing*. In press (accepted). May. 2016. (DOI: 10.1109/TSC.2016.2561934 for Early Access)

Peer-reviewed Conference Papers

1. Fankhauser, T., Q. Wang, A. Gerlicher, C. Grecos and Wang, X. (2014). 'Web Scaling Frameworks: A novel class of frameworks for scalable web services in cloud environments'. In: *Proc. IEEE International Conference on Communications (ICC)*. pp. 1760–1766, Sydney, Australia, Jun. 2014.

1. Introduction

1.1 Overview

Over the last decade, the Internet has become a social place of global utmost importance. The principle of users contributing content to web platforms is omnipresent. For the current generation of users it is completely normal to have video chats with other users, buy and sell new or used products, plan trips on online maps, share pictures with friends, rate holidays and even get medical advice on the social web. Further, with the smart health trend users have started to contribute and share anonymised personal data into the web. In the future, this data can be used to predict and monitor disease spread and ultimately even prevent outbreaks completely. With the upsurge of the Internet of Things, supplementary data is contributed to the web by power grids, cars, household appliances, traffic regulators and buildings. Through mobile smart devices and sensors, a constant stream of messages between multiple endpoints is established leading to ubiquitous data exchange. This stream of messages is further increased by users having multiple mobile devices such as smart phones, tablets, laptops and smart watches that constantly collect and communicate data. All content and communication data are analysed for correlations, patterns and abnormalities with big data algorithms. This enables platforms to gather and prepare significant information aggregated from enormous collections of sources. The development of the social web is highly influenced by the cloud computing trend. Cloud computing allows provisioning computing resources and storage on demand where the costs are based on processing time and storage space. Both the handling of the increased number of messages and the processing of huge amounts of data is only possible as customers can dynamically rent and return resources as required. This allows cloud customers to create products potentially operating world-wide with no upfront investments. On the technical side, the enormous amounts of data that have to be processed as quickly as possible introduce major challenges. Systems have to deliver high performance by processing as much data per second as possible. However, a single machine provisioned from a cloud provider is not able to cope with all data. In consequence, multiple machines need to be employed and coordinated to process the data for scalability considerations. Further, systems need to interoperate multiple cloud providers to utilise multiple cloud services and benefit from different pricing models. As the amount of data that needs to be processed per second changes over time, systems need to be scaled continuously to ensure high performance. Scaling a system spanning multiple providers introduces a number of complex issues:

- To how many machines should the system be scaled minimally at each given point in time?
- How is the processing distributed between the machines?
- How fast should machines be added or removed when the amount of data changes?
- Where is the data stored that needs to be shared between the machines?
- On which geological location should the machines be placed?
- How is the application logic affected by multiple machines?
- Which cloud provider offers the best resources for the processing?
- How to ensure that data is not processed multiple times on different machines?
- Where to place the logic for machine provisioning?

As there is no clear answer for any of the issues, many customers decide to solve performance problems manually at the time they appear. As presented in the above issues, however, the solutions are quite complex and therefore take time to implement. Unfortunately, performance problems typically appear when a certain critical mass of users for the product is reached. If the product is then unavailable due to performance problems, users can not test the product and consequently might decide to abandon the product completely. Jacko, Sears and Borella (2000) find that for products that require a high level of interactivity such as social networks, the user experience massively suffers from slow response times. Therefore, web products need not only be constantly available, but also fast. A further challenge is the optimisation of data processing. In order to save valuable processing operations, typical web applications store results from the processing in a cache. When a new processing is about to be scheduled, it firstly looks into the cache to see whether the result is already present. If it is present, the processing can be skipped. The size of the cache is limited due to storage space cost, thus each processing result is evaluated for its importance where it might evict another result in the cache. Two major challenges exist with this approach. Firstly, it is a complex matter to select the correct results where the system benefits most from keeping them cached. Secondly, social web applications tend to have huge amounts of different results as every user receives content especially composed for him/her. This makes it hard to determine the importance of a cached result and can lead to highly fluctuant cache contents where many results are processed for multiple times. The aforementioned issues motivate a research effort on the automatic and optimised provisioning and composition of cloud resources with novel algorithms improving the processing performance. A further concern of this thesis is to identify possibilities to incorporate the optimisations into a common framework that can be shared by both cloud providers and

customers, so that they can easily benefit from the features that are implemented on another layer of abstraction.

1.2 Research Aim and Objectives

The overall aim of this project is to find and evaluate a novel class of frameworks taking over the complex task of automatic scaling in an optimised fashion. In contrast to existing Web Application Frameworks (WAFs) responsible for building the logic of a web application, this thesis presents and evaluates the concept of Web Scaling Frameworks (WSFs) responsible for scaling web applications in an automatic and optimised fashion. Consequently, the project is planned to fulfil the following specific objectives:

- **O1**: *To separate the concerns of application logic and scaling logic:* Find a conceptual architecture design including required modules, interfaces, parameters and components valid for all implementations of WSFs. Identify cloud architecture patterns that can be applied to support an optimised, manageable and versatile structure of web applications. Ensure that existing WAFs can be used in combination with WSFs.
- **O2**: *To distribute work to multiple components and fully benefit from a novel caching approach:* Design and implement a novel request flow scheme that routes requests efficiently through a novel composition of components in an optimised fashion. Develop an analytical model of components and their composition describing the request flow performance in comparison with a traditional approach and identify major influencing performance parameters. Find a model describing the optimal performance curve of a component with corresponding key metrics. Evaluate the model on both the component and composition level with real world data.
- **O3**: *To optimise the processing performance of the novel caching approach:* Analyse the structure of resources and their dependencies to find algorithms that can be used to optimise the processing performance. Identify key metrics of the dependencies with major influences on the processing speed and complexity. Find novel methods to generate resources with their dependencies based on existing real world application structures. Develop an analytical model describing the processing duration of resources with their dependencies and a model of a traditional approach to compare their performance. Evaluate the models with real world data.
- **O4**: *To enable multiple cloud provider systems and predict resource cost:* Design and present a prototypical implementation of an architecture that enables WSF modules and components to be migrated between cloud providers and be operated by multiple cloud providers simultaneously. Identify required

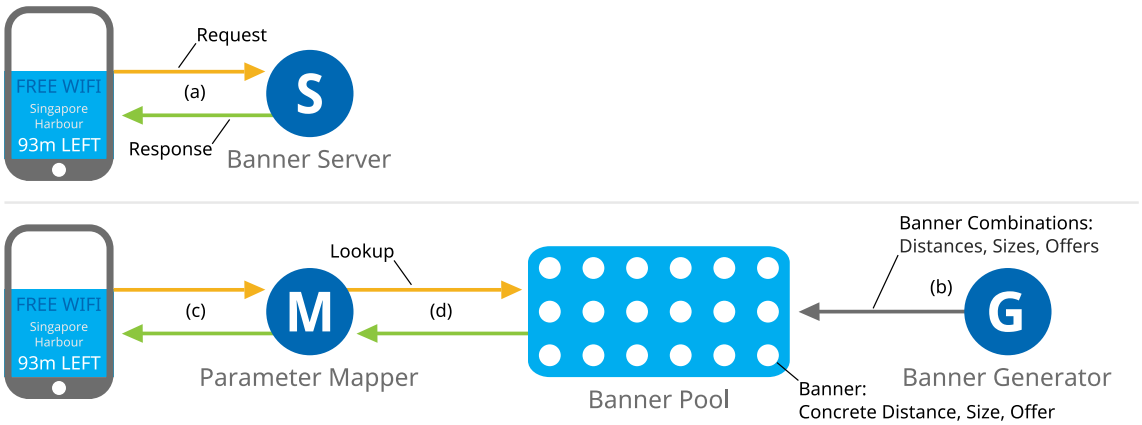


Figure 1.1: The optimisation of a mobile advertising platform served as initial motivation for the project.

steps to integrate web applications created with WAFs into a WSF. Develop analytical models describing the processing cost and storage space requirements of both a dependency based processing and a traditional processing approach. Evaluate the models with empirical data.

In summary, to fully exploit the potential of cloud computing a new layer of abstraction responsible for automatic scaling must be extracted. Further, resource update mechanisms must be found that utilise modern cloud storage capabilities to present an optimised alternative to traditional caching systems.

1.3 Project Overview and Technical Contributions

The initial motivation for this project was developed back in 2012, when the author was working as a freelancer for a mobile advertising platform in Singapore. The platform delivered dynamic mobile image banners to smart phones based on their geo location as shown in Figure 1.1 (a). With the growth of the platform, the traffic quickly increased up to a level where it was necessary to scale out the servers. The resources were of highly dynamic nature as the image banners included exact distances from the customers to the point of interests, special offer texts and images were customized for the screen sizes of the requesting devices. This led to a point where the majority of used machines were constantly reprocessing similar images, and customers had to wait a few seconds for the banners to be ready. The application of contemporary caching mechanisms did not work as every image was custom build for a single request. After some intense work, the engineering team and the author developed a scheme where all combinations of discrete distances, sizes and offer texts were preprocessed as soon as the banners were added into the system. The resulting images were stored into cloud storage as shown in Figure 1.1 (b). All banner requests were routed to a subsystem designated to retrieve image banners only. The request parameters were mapped to one of the distinct tuples referencing a preprocessed image that could be delivered instantly as shown in Figure 1.1

(c,d). This allowed the system to reduce the number of total servers by 80% while improving response times by over 95%. However, the scheme was highly customised to the problem, so consequently a first evaluation of a more abstract scheme was developed by the author in his master thesis in 2013. In the master thesis, it was evaluated if the general preprocessing of all resources can lead to performance improvements. The results of the master thesis showed that a scheme where by default all requestable resources are persisted is not reflected by current research. This identified the research gap ultimately leading to the research efforts presented in this thesis.

In the initial phase of the project, the idea of different subsystems for requests that need processing and requests that read content was tackled. The illustration of the problem and proposed solution with the subsystems received a best poster award from the UWS Institute of Creative Technologies and Applied Computing. An initial development and evaluation of a performance model with components hosted on a single machine helped to assess the general potential of the novel request flow scheme and was published by Fankhauser, Wang, Gerlicher et al. (2014) and presented at the IEEE International Conference of Communications (ICC) in Sydney.

After the positive feedback from the IEEE ICC conference publication, the novel request flow scheme was further examined. A conceptual architecture was developed and the model was enhanced significantly to describe component clusters of multiple machines. During the initial steps for the evaluation of the components, it was noticed that the performance of a component was heavily influenced by the number of concurrent requests entering the component and there exists an concurrency to requests per second optimum. This optimum was extracted into an optimal concurrency range model and a performance-concurrency-width triplet that allows a simple performance comparison between components. The triplet thereby includes hardware and software components as it describes only the relation between concurrency and request flow. Further, the model was evaluated with real world data where the model fits support the approach. The complete work was published as a journal paper by Fankhauser, Wang, Gerlicher et al. (2015) in the IEEE Transactions on Services Computing (TSC) journal. The results from the enhanced modelling identify the time that is available to process resources with their dependencies when the novel request flow is used over a traditional routing scheme.

In the third phase, this led to the question of an exact modelling of the resource dependency processing. The structure of web applications allowed defining a graph of resources as nodes with directed edges as dependencies. For an optimisation of the processing of this graph, it was possible to find algorithms in the domain of project management that allowed an order constrained processing of resources when enhanced. For the evaluation of the developed model, it was infeasible to obtain real world data for the novel concept as current web applications do not define their

dependency graph. As a solution, existing Application Programming Interfaces (APIs) of social web applications were analysed to extract key metrics. With these key metrics, it was possible to develop algorithms that create dependency graphs of arbitrary dimensions exhibiting similar key metrics. The generated graphs then were used to evaluate the models. It was further interesting to identify correlations between key metrics of dependencies and the overall performance. The correlations helped in finding the most influential metrics worth optimising. The work of the third phase has been published as a journal paper by Fankhauser, Wang, Gerlicher et al. (2016) in the IEEE Transactions on Services Computing (TSC) journal.

In the fourth and final phase of the project, a prototypical implementation of a portable and interoperable WSF was developed along with a model and empirical evaluation of the processing cost and storage space requirements. The usage of Linux containers for components facilitated the defining of containers that run on multiple cloud providers. By implementing a social application and integrating it into the prototypical implementation of the WSF, the adaptions required for the integration were found. During deployment of the application, it was necessary to initially set up a rough number of the targeted amount of processing and required storage space. This led to the development of a model that allows determining the required processing cost and storage space for applications using a WSFs with dependency processing and applications using a traditional processing approach. The evaluation of the model was executed on the prototypical implementation where the model fits the empirical data to a high degree. The work of the fourth phase will be submitted for publication.

With the completion of the project, three major concepts allow a new understanding in the field of scalable web services. Firstly, the modelling of service component performance can be simplified by abstracting complex service internals such as CPU load, memory usage and network saturation into processing delays. Cloud provided service components, such as queues or key-value databases often abstract the internals of a more complex system. Consequently, the only metric available across providers and service components is the time it takes to enqueue a message or fetch a record from a database by key. Secondly, to accurately describe the relationships between resources, the data model of a web application can be utilised. This enables building resource cache systems that are based on accurate resource dependencies in contrast to cache eviction based on approximation. Finally, the type of interaction with a web service can be used to design and scale individual subsystems for the service. Users consume and produce content in different rates for different services, such as read-heavy video on-demand services and write-heavy social messaging services. Consequently, individual subsystems dedicated to content consumation and production enable fine-grained scalability and decouple the effects of performance issues.

1.4 Research Methodology

The research project is motivated from the real world problem of scaling a mobile advertisement platform as described in the previous section. As a first step, the initial idea to create individual subsystems for scaling was formulated. Next, an intensive literature review was performed in order to find and identify existing work in the field. With the results from the literature, the idea was adapted and an initial prototype was built. In order to predict the performance, initial modelling was performed and the prototype refined to measure the predictions of the model. After further iterations spent in refining the model and prototype, data for an empirical validation was collected. Next, the empirical evaluation was designed and executed. As a result of the evaluation, the models were further refined and evaluated until the model fits suggested a defined accuracy level. This approach was repeated for all three main chapters of the project Chapter 5, Chapter 6 and Chapter 7, where the results from the previous modelling and evaluations influenced the subsequent chapters.

The proposed concepts, patterns and schemes are validated using mathematical modelling and empirical evaluation. For the empirical evaluation with multiple computers, an evaluation platform with 42 Raspberry Pi computers named *Pi-One* was built as detailed in Chapter 2. Both the performance of a Raspberry Pi computer in comparison to traditional server hardware and with different programming frameworks was evaluated to ensure the validity and transferability of results. All iterations to refine the models were carried out until the model fits suggested a defined accuracy level. The data collections used to validate the models were chosen to cover a large range of potential applications.

1.5 Thesis Outline

This thesis is organised into eight chapters as illustrated in Figure 1.2. The chapters are based on three work packages. Figure 1.2 presents the outcomes of the work packages as publications and contents of thesis chapters.

- **Chapter 1**: *Introduction* is the current chapter that introduces the motivations and structure of this thesis.
- **Chapter 2**: *Background* provides essential background information to help understanding the problem definition and advanced concepts presented throughout this thesis.
- **Chapter 3**: *Related Work* provides a critical literature review of recent relevant work. The related work is categorised into sections concerning the general concept of WSFs, an optimised request flow between components, the processing of resource dependencies, the orchestration of components and cloud portability and interoperability. This matches the subsequent chapters as

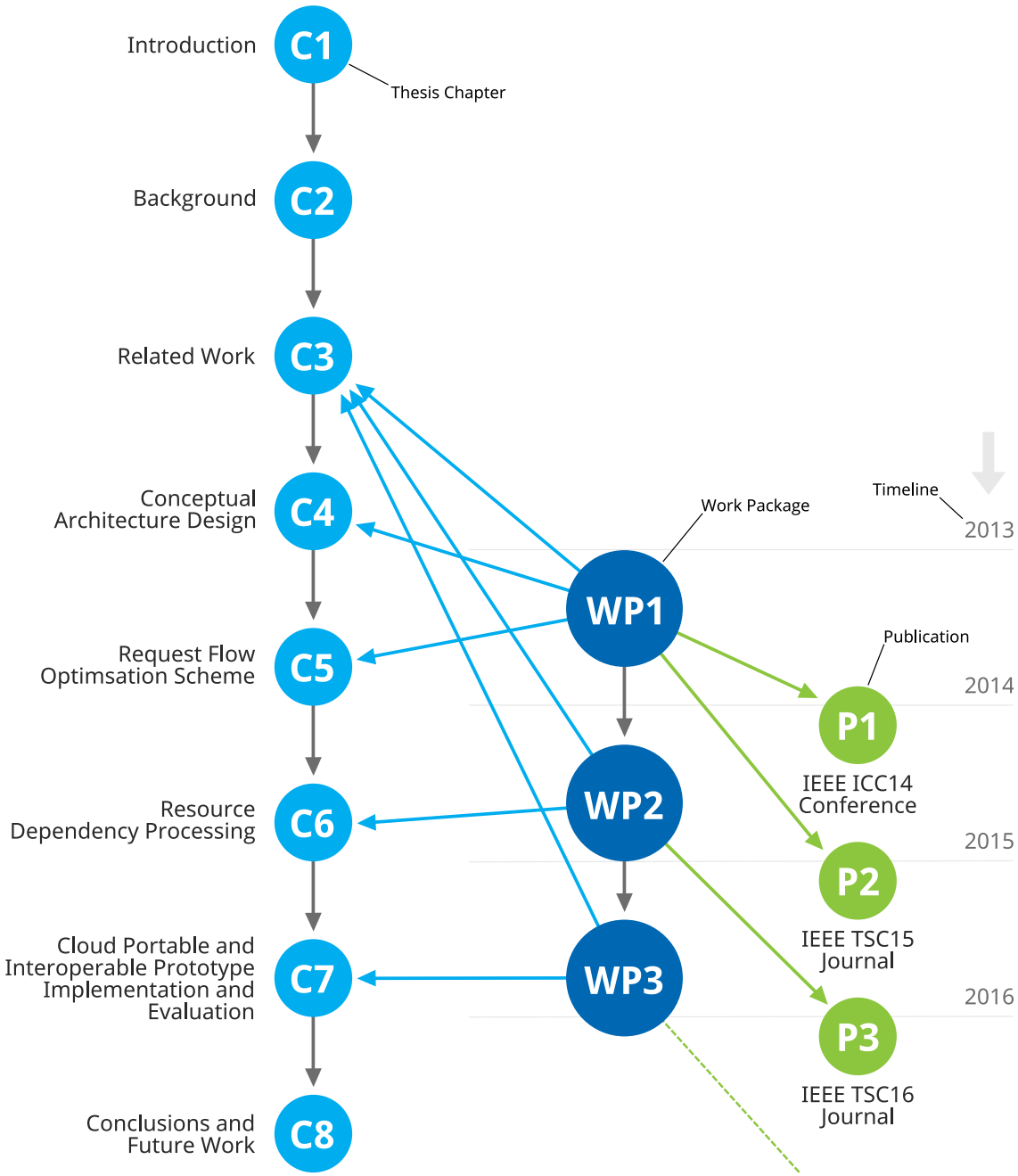


Figure 1.2: Overview of thesis outline with chapters, work packages and publications.

shown in Figure 1.2.

- **Chapter 4**: *Conceptual Architecture Design* presents a conceptual architecture design including required modules, interfaces, parameters and components valid for all implementations of WSFs.
- **Chapter 5**: *Request Flow Optimisation Scheme* presents a novel design pattern for resource storage and management, and an optimised request flow scheme between components. A Permanent Resource Storage and Management (PRSM) pattern is developed that enables all resources to be fetched without prior processing, where the processing step is shifted to a management model.

- **Chapter 6**: *Resource Dependency Processing* presents the dependency structure and key graph measurements of web resources. A longest-path algorithm using topological sort with dynamic programming is developed for efficient processing. Further, dependencies are analysed to find correlations between processing performance and graph measures.
- **Chapter 7**: *Cloud Portable and Interoperable Prototype Implementation and Evaluation* presents a cloud portable and interoperable prototype implementation of a WSF. It is shown how existing web applications can be integrated into a WSF to enhance the scalability and portability. Further, a model to calculate and compare the processing cost and storage space of resources is developed.
- **Chapter 8**: *Conclusion and Future Work* concludes this thesis based on the findings of the aforementioned chapters and presents an outlook to possible future research.
- **Appendix A** provides a list of acronyms and abbreviations used in this thesis.
- **Appendix B** attaches the awards this research project was distinguished with.
- **Appendix C** illustrates the Pi-One Evaluation Cluster mainly used for the evaluations in the project.

2. Background and Evaluation Platform

2.1 Overview

This chapter provides essential background information to help understanding the problem definition and advanced concepts presented throughout this thesis. The goal of this thesis has been the design and evaluation of a Web Scaling Framework (WSF) enabling the automated and cloud interoperable scaling of web services with advanced throughput performance. Therefore, the major characteristics of the world wide web along with common web architecture patterns and web scaling schemes are summarised here. Furthermore, the computing platform used for evaluations throughout this thesis is presented and key figures collected from the cluster are introduced at the end of the chapter.

2.2 The Web

The World Wide Web is an information space where clients can retrieve documents from servers connected to the Internet. Clients appear in many forms such as browsers on desktop computers, mobile smartphones or infotainment systems in cars. Servers provide documents often stored in databases for a better organisation. Documents are uniquely identified by a Uniform Resource Identifier (URI) that specifies the protocol scheme, the host, the port and the path of a resource. Document contents are formatted in hypertext. Hypertext is structured text that contains hyperlinks to other documents identified by their URI.

2.2.1 Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is an application protocol standardised by the Internet Engineering Task Force (IETF) to exchange hypertext between clients and servers. In order to exchange a document, a client expresses its intent to retrieve a document by sending a request to a server.

2.2.1.1 HTTP Request

A HTTP request has a standardised form containing a header and a body. The request header is used to pass information such as the HTTP method, the URI, the HTTP version, accepted content-type for the response and information about the client to the server. The request body can optionally be used to transfer data to the server. The HTTP methods (IETF RFC2616, 1999) presented in Table 2.1 indicate the action to be performed by the server. By definition of IETF RFC2616 (1999) safe methods do nothing but retrieve content without any side effects. Idempotent

Table 2.1: HTTP methods indicating the action to be performed on a server with safe and idempotent classifications.

Method	Action	Safe	Idempotent
OPTIONS	Request communication options	yes	yes
GET	Retrieve content	yes	yes
HEAD	Retrieve information header only	yes	yes
POST	Create information provided in request body	no	no
PUT	Update information provided in request body	no	yes
DELETE	Destroy document	no	yes

methods leave the server in the same state regardless of the number of times the request is processed. All other methods strictly have side effects and therefore must be ensured to be processed only once. This distinction is important for this thesis as the request routing flow presented in Chapter 5 utilises the properties of safe methods. The following example illustrates a complete HTTP request for a document initiated by a client:

```
GET /index.html HTTP/1.1
Host: webscalingframeworks.org
User-Agent: curl/7.43.0
Accept: */*
```

2.2.1.2 HTTP Response

Analog to the request, a HTTP response has a header and a body. The response header contains information about the result such as the HTTP status code, the HTTP version, the content-type and content-length. The response body contains the content of the requested document. Please refer to IETF RFC2616 (1999) for a conclusive list of header values and options. The following example illustrates the complete HTTP response received for the above request (comments are denoted with //):

```
// Header
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 154

// Body
<!DOCTYPE html>
<html>
<head>
  <title>Web Scaling Frameworks</title>
</head>
<body>
```

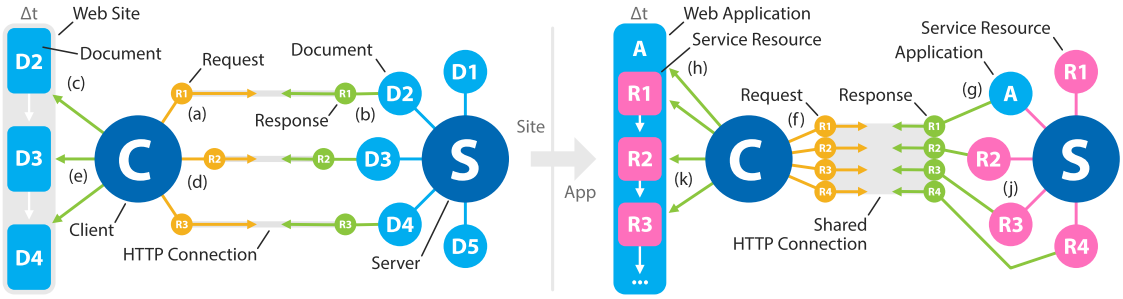


Figure 2.1: Evolution of the Web from document based web sites to interactive web applications requesting resources from web services.

```
  <h1>Web Scaling Frameworks</h1>
  <p>Welcome!</p>
</body>
</html>
```

2.3 Web Sites

A web site is a related set of hypertext documents typically served for a single host name such as `http://webscalingframeworks.org`. Figure 2.1 illustrates the exchange of documents for a web site. In Figure 2.1 (a) the client creates and sends a request through a new HTTP connection to a server. The server responds at Figure 2.1 (b) with the requested document. The client then presents the document to the user in Figure 2.1 (c). As the user follows a link in document D2 to D3, the client creates a new HTTP connection to request document D3 in Figure 2.1 (d). Finally in Figure 2.1 (e), the client throws away document D2 and presents D3 to the user. As for HTTP/1.1, connections are not automatically closed after the server responds but can be reused. According to IETF RFC2616 (1999), "A single-user client SHOULD NOT maintain more than 2 connections...", however modern browsers are known to maintain up to 6 connections to improve performance.

2.4 Web Applications

Web applications fundamentally extend the notion of a web of hyperlinked documents. Instead of the exchange of mere documents, full applications are served to clients while servers implement complex application logic. The right side of Figure 2.1 illustrates the exchange of an application and its requested resources for a web application. In Figure 2.1 (f) the client requests to load an application from the server at Figure 2.1 (g). Once the client retrieves the application in Figure 2.1 (h), it executes and presents the application to the user. At any time the application decides to need more content, it requests new resources from the server over a shared HTTP connection as shown in Figure 2.1 (j). A resource can be of any kind such as lists of raw data, images or videos where the next section elaborates on further

details. Finally, the resources are processed in the client application at Figure 2.1 (k) and optionally presented to the user.

2.4.1 Back End

The application back end is implemented on a single server or multiple, interconnected servers. In contrast to traditional web sites, web application back ends provide enhanced functionality such as user authentication and authorisation, dynamic content management and data aggregation. As managing application logic is a complex matter, Web Application Frameworks (WAFs) take over and simplify common tasks involved in creating application logic.

2.4.2 Front End

The application front end is transferred to the client and presented in the browser. A front end application differs from web site documents. It can offer interactive user interfaces with instant response as needed in web mail, word processing, online auction, video on demand, instant messaging or graphic design applications. WAFs dedicated to the front end simplify the complex tasks of application development and simplify common problems such as browser diversity.

2.4.3 Web Evolution

Figure 2.1 illustrates the evolution of the web from document-based web sites to interactive web applications requesting resources from web services. Today, pure web sites that exchange only documents with a server are increasingly replaced by web applications. Many hybrid solutions exist where multiple smaller applications are loaded into documents, e.g. `http://facebook.com`, `http://twitter.com` or `http://amazon.com`. The hybrid approach is often selected in order to preserve the ability to be indexed by search engines. Pure web applications, such as `http://netflix.com`, `http://mail.google.com` or `http://docs.google.com` are often closed systems by design, where linking of application state is achieved only within the application domain. This thesis considers both types of web sites and web applications, where the current trend points towards an increased use of web applications.

2.5 Web Services

Web services provide resources and functionality on the server back end. In contrast to documents provided by a server, they expose small units of application logic. The information is exchanged in machine readable data formats, such as Extensible Markup Language (XML) or JavaScript Object Notation (JSON). This enables, as defined by the World Wide Web Consortium (W3C), an "... interoperable machine-to-machine interaction over a network" (W3C Web Service Architecture, 2004).

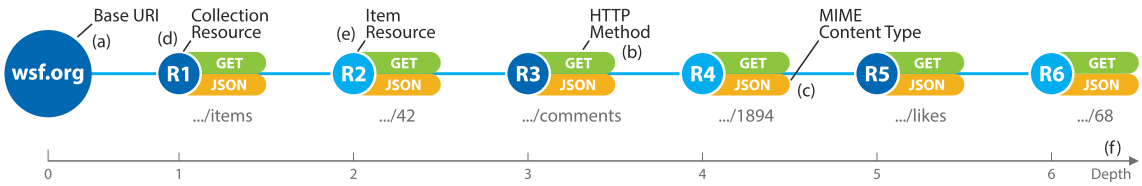


Figure 2.2: Representational State Transfer (REST) architectural style.

2.5.1 Service Oriented Architecture

A Service Oriented Architecture (SOA) is a distributed systems architecture of web services as defined by W3C Web Service Architecture (2004). In a SOA, web application front ends communicate with web application back ends using web services as shown in Figure 2.1 (j-k). Additionally, web services in a SOA enable a web application back end to communicate with other web application back ends. This enables the creation of complex web applications on highly distributed systems as presented throughout this thesis.

2.5.2 Architectural Styles

The authors in W3C Web Service Architecture (2004) identify two major classes of web services: Web services with an application specific architectural style and web services complying to the Representational State Transfer (REST) architectural style.

2.5.2.1 Application Specific

As shown in Table 2.1, HTTP defines methods to indicate desired behaviour of web service actions. However, many design decisions remain undefined by HTTP and thereby open to the application (Fielding, 2000). Consequently, web services can be incompatible unless following a uniform architectural style.

2.5.2.2 Representational State Transfer

The REST architectural style proposed by Fielding (2000) constraints HTTP to uniform interface semantics. This enables web services to be formed as compatible components communicating with each other. The use of compatible components is a fundamental approach essential to this thesis, where a novel component architecture is presentend in Chapter 4. Servers providing a RESTful Application Programming Interface (API) typically follow a URI naming scheme as illustrated in Figure 2.2. The base URI at Figure 2.2 (a) identifies a root resource name. Using an HTTP method at Figure 2.2 (b), a resource identifies the action it performs, such as Create, Read, Update or Delete (CRUD). In Figure 2.2 (c), a resource declares the Multipurpose Internet Mail Extensions (MIME) content type for its current representation independently from the URI. A collection resource at Figure 2.2 (d) bundles multiple item resources at Figure 2.2 (e), where an item resource represents content for

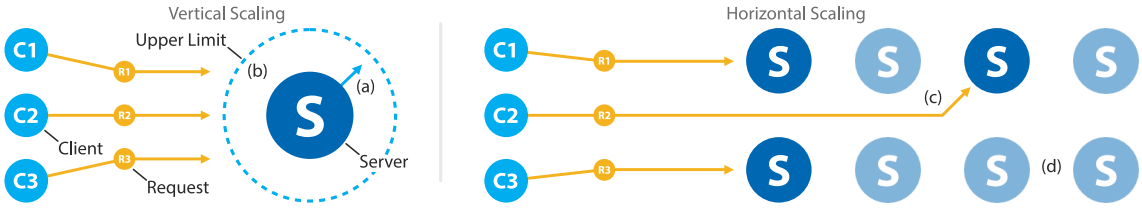


Figure 2.3: Horizontal and vertical web scaling of servers for multiple requesting clients.

a single entity. An item resource can link to dependent collection resources linking to further item resources. This hierarchy and its depth (Figure 2.2 (f)) is essential to the resource dependency processing approach presented in Chapter 6.

2.6 Web Scaling

The processing power of servers is limited by their hardware. As a result, servers are only able to handle a limited number of requests and responses per second. In this thesis, the number of requests per second is denoted as load, where the number of requests and responses per second is denoted as throughput. If the load applied from clients is higher than the maximum load a server can handle, the server can not accept further requests which leads to a service outage. In order to prevent service outages, the server needs to be scaled.

2.6.1 Horizontal and Vertical Scaling

Figure 2.3 illustrates two possibilities to scale a web service. For both possibilities, the clients are represented as C1, C2 and C3 that send requests R1, R2 and R3 to a server, or cluster of servers. In Figure 2.3 (a), the server processing power is increased by adding a faster Central Processing Unit (CPU), more main memory or additional network interfaces. In Figure 2.3, this is represented by increasing the size of the circle. This approach however has an upper limit at Figure 2.3 (b), which is determined by the maximum hardware performance available. If the load increases above the upper limit, further service outage occurs, hence limiting the field of application for vertical scaling. In contrast, horizontal scaling increases the number of servers and distributes portions of the load as shown in Figure 2.3 (c). The servers coloured in dark-blue represent existing servers for the current amount of requests. The ligh-blue coloured servers are servers that can optionally be provisioned if the number of requests increases. This approach has virtually no upper limit, although it introduces complexity through the increased management effort of multiple servers. A hosting with traditional servers is expensive as the number of servers has to be increased to fit the maximum load ever applied to the web application.

2.6.2 Cloud Computing

In the Infrastructure as a Service (IaaS) model (Fehling, Leymann, Retter et al., 2014), cloud computing enables on-demand provisioning of processing power and further resources on a pay-per-use basis. This allows a web application to scale dynamically using a horizontal scaling approach. During low load periods only few servers are provisioned to keep the infrastructure cost at minimum. When the load increases, further servers can be added to the cluster to prevent service outages (Figure 2.3 (d)). As dynamically adapting the number of servers is a complex matter, this thesis proposes a system to evaluate the required number of servers for individual components in Chapter 4.

2.7 Web Architecture Patterns

Web architecture patterns (Fehling, Leymann, Retter et al., 2014) describe common approaches to split up web applications into components. According to Fehling, Leymann, Retter et al. (2014), web applications are divided into three logical tiers: A presentation tier, a business logic tier and a data tier. The presentation tier supplies all user interface related functionality. The business logic tier implements the processing of application related functionalities. The data tier stores and provides content for the business logic tier. Two-tier applications combine the presentation and business logic tier. Three-tier applications have separate tiers for each category of functionality.

2.7.1 Monolithic Architecture

Using a monolithic architecture pattern, a web application is developed as a single component. The component contains all application functionality and typically connects directly to a database. Due to the fact that all functionality is bundled into a single component, development, testing and deployment is simplified (Fowler, 2014). According to Fowler (2014), with increasing complexity of applications, it becomes hard to structure and scale monolithic applications.

2.7.2 Microservice Architecture

In the microservice architectural pattern, an application is structured as a suite of small web services (Fowler, 2014; Namiot and Sneps-Sneppe, 2014). Each service component offers an autonomous functionality and typically has a self-reliant service database. Small services allow simplified development and testing of components and enable fine-grained scalability. Deployment becomes more complex as services have to be discovered, monitored and maintained. This thesis uses the microservice architecture pattern for the architecture in Chapter 4. Applications managed by a WSF however are not constrained to a specific architecture pattern.

2.7.3 Traditional Approach

This thesis uses a traditional two-tier application as performance baseline. In the Traditional Processing (TP) approach, HTTP requests enter the system through a load-balancer component. The load-balancer distributes the requests to one of many application servers. An application server parses the requests and checks if the response it is asked to deliver is present in a cache component. If the response is present, it fetches the response from the cache component and returns it via the load-balancer to the client. If the response is not present, it creates the response by fetching entries from a database component and converting the entries into the resource representation format the client asked for. Finally, the response is sent to the client via the load-balancer. For this approach, the application server implements logic with the help of a WAF that typically supports the implementer with simplified fetching of records from the database and the rendering of resource representations. A key point of this approach is that every request sent to the service is handled by the application server, which increases the performance demands for the cluster of application servers. A detailed illustration of this approach is given in Figure 5.2 and corresponding performance modelling is presented in Chapter 5, Chapter 6 and Chapter 7.

2.8 Pi-One: Evaluation Platform

For all performance evaluations, this thesis uses a custom-built cluster of 42 Raspberry Pi computers named *Pi-One* (Appendix 8.6). To ensure the results can be transferred to other implementations and hardware, the evaluation system is tested for influences of different programming frameworks and hardware.

2.8.1 Hardware and Software Construction

The cluster illustrated in Appendix 8.6 is built from 42 Raspberry Pi 1 Model B+ computers. The computers are mainly selected due to financial limitations of the project, where the cost for a single computer including all required accessories was 50 EUR. Each of the computers has a 700 MHz ARMv61 single-core CPU, 512 MB of memory, 16 GB of flash storage and a 100 Mbit/s Ethernet interface. All computers are directly connected to a HP ProCurve 2810–48G 48 port Ethernet switch. An additional Raspberry Pi computer is connected to the switch that orchestrates all other computers. It serves as a DNS server for the computers and central gateway to the cluster. Each computer is individually powered by a dedicated power supply. All computers run Arch Linux in the ARM edition and are accessible via Secure Shell (SSH). The evaluation tasks are individually started via SSH by the master computer. Therefore, scripts are created on the master, which automatically build and deploy programs to the target computers, execute and repeat the evaluations and collect the results. Further, for the evaluations in Section 7.3.3, each computer

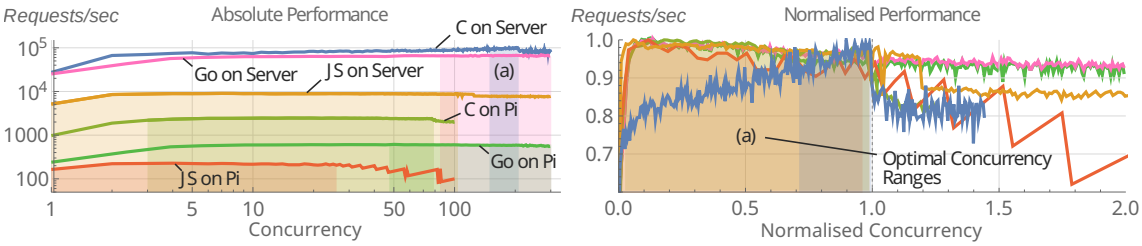


Figure 2.4: Absolute and normalised request per second and concurrency comparison of a C, JavaScript and Go implementation on server hardware and Pi computers.

runs a Docker (2013) daemon that is used by a Docker client running on the master computer to deploy containers.

2.8.2 Evaluation Setup

For the evaluation of the platform, a request generator sends a request to a server, waits for the response and then immediately sends the next request. The generator measures the throughput for 10 minutes excluding the first 2 minutes and then records the average requests per second achieved within the 8 minutes period. Then, both the request generator and the server are restarted to begin with the next evaluation where two requests are started at the same time, thereby increasing the concurrency to two. With increasing concurrency, this is continued until the measured requests per second reach a maximum and then drop down to at least 25% of that maximum.

2.8.3 Influence of Programming Framework

To compare the influence of the programming framework on all measured results, three implementations are tested: An implementation using the C programming language with the libuv (2011) library, a JavaScript implementation on node.js (2009) and a Go (2009) implementation using Go's native HTTP server. The complexity of all all implementations varies, where the C implementation is most complex due to the hardware-related programming such as manual memory and concurrency management. It is followed by the Go implementation that completely abstracts the hardware layer but requires manual concurrency management via channels. The most simple implementation is the JavaScript implementation that automatically manages concurrency with an asynchronous event loop and callbacks. The Go implementation is between 5%-15% slower than the C implementation as shown in Figure 2.4. The C implementation is noticed to be one order of magnitude faster than the JavaScript implementation.

2.8.4 Influence of Hardware

Typically, cloud providers do not use Raspberry Pi computers for their hosting services. Consequently, it is necessary to evaluate the effect of different hardware on the results to ensure their transferability. For this, all implementations from the

previous section are tested on the following platforms: Server hardware using a 2.6 GHz Intel Core i7 and a Raspberry Pi computer where both are running Arch Linux. For network switching, a HP ProCurve 2810–48G Ethernet switch is employed whose ports operate at 100 Mbit/s. The server hardware is noticed to be approximately one order of magnitude faster than the Pi computers.

2.8.5 Results

The data supports that the results of the evaluation are transferable among different implementations and hardware. Figure 2.4 illustrates the normalised and absolute performance curves of all implementations on both hardware. All normalised curves show the same performance pattern where the performance increases to a maximum, stays close to the maximum for a while and then breaks down as the system is overloaded. The performance of the C implementation using libuv (2011) on the server hardware increases notably slower than all the other implementations. Therefore, the C implementation is the implementation most sensitive to concurrency, where both other implementations exhibit performance close to the optimum in a wider range of concurrencies as shown in Figure 2.4 (a). Chapter 4 analyses this optimal concurrency range in detail. Based on the results, where the C implementation is most sensitive to concurrency, complex to implement and for these reasons sparsely used in web development, further implementations of prototypes in this thesis are implemented using Go and JavaScript.

2.9 Summary

This chapter has introduced essential technology and terminology for the presented research. Static web sites progressively evolve to complex web applications, where web services provide functionality in machine readable formats. Due to increasing numbers of requests, web services have to be scaled to prevent service outages. Applying common architecture patterns for scaling, such as the microservice architecture increases web application complexity by an enormous extend. Hence, this thesis proposes and elaborates on a framework enabling automatic scaling with improved request throughput performance in Chapters 4, 5, 6 and 7. Finally, the computing cluster *Pi-One* used for performance evaluations of the traditional approach and prototypes proposed in this thesis has been presented and evaluated. The data supports that *Pi-One* results collected in this thesis are transferable among different implementations and hardware.

3. Related Work and Theoretical Foundations

3.1 Overview

This chapter provides a critical literature review of recent relevant work and presents theoretical foundations of the project. The related work is categorised into sections concerning the general concept of WSFs, an optimised request flow between components, the processing of resource dependencies, the orchestration of components and cloud portability and interoperability. Table 3.1, Table 3.2, Table 3.3 and Table 3.4 give an overview of the reviewed work classified into sub-categories. Each section identifies open research questions that are addressed in the Chapters 4, 5, 6 and 7. Finally, Section 3.6 presents an overview of open research questions in reference with their dedicated chapter and Section 3.8 provides a final summary of the literature review.

3.2 Web Scaling Frameworks

In order to bundle the complex matter of scaling into a dedicated, reusable and maintainable logical unit, the general concept of a WSF is proposed in detail in Chapter 4. Cloud providers offer infrastructure and services that are used by applications to implement logic. As an example, a storage service of a cloud provider offers virtually unlimited storage capacity to an application, a scalable queueing service allows multiple logical modules of an application to communicate with each other in a decoupled fashion and the dynamic provisioning of multiple servers enables an application to adapt the requests that can be processed per second to the number of incoming requests per second created by users of the application. The role of a WSF is to define and automate how the offered services interact with the application logic. A possible implementation of a WSF could be hosted on a server of a cloud provider and ensure that the application logic that runs on other servers has enough queueing and storage resources by provisioning optimised amounts from the cloud provider. The full spectrum of features required to efficiently scale web applications is detailed in Chapter 4 and includes modules that collect metrics from components, provision components, watch and store metrics, implement resilient behaviours, deploy applications and trigger component actions such as moving an application to another cloud provider. The following sections identify recent relevant work presenting related concepts and approaches.

Table 3.1: Categorisation of work related to Web Scaling Frameworks.

Web Scaling Frameworks (3.2)	**References**
Platforms and Frameworks	Addo, Do, Ge et al. (2015), Fehling, Leymann, Retter et al. (2014), Zareian, Veleda, Litoiu et al. (2015), Krintz (2013), Wolke and Meixner (2010), Tung, Chaw, Xie et al. (2012), Mao and Humphrey (2011) and Marshall, Keahey and Freeman (2010)
Auto Scaling Features	Amazon Web Services (2006), Google Cloud Platform (2008), RackSpace (1998), IBM Bluemix (2014) and DigitalOcean (2011)
Cloud Application Design Patterns	Fehling, Leymann, Retter et al. (2014), Fowler (2009), Young (2010), Fowler (2011) and Facebook Flux (2014)

3.2.1 Platforms and Frameworks

A scaling platform or framework provides functionalities to improve web application scaling, deployment, maintenance, monitoring and performance. Addo, Do, Ge et al. (2015) present an architectural design for building and sustaining large-scale social media intelligence solutions that can exhibit high-scalability, high-availability and improved performance attributes. The proposed architecture design is specialised in social media intelligence solutions, where a traditional three-tier architecture (Fehling, Leymann, Retter et al., 2014) is combined from a collection of IaaSs and Platform as a Services (PaaSs) in the cloud. The *Knowledge-Feed* platform proposed by Zareian, Veleda, Litoiu et al. (2015) facilitates monitoring of web applications, creates performance and cost models and executes machine provisioning to optimise performance and infrastructure cost. The *AppScale* Cloud Platform as proposed by Krintz (2013), is a distributed software system that implements a PaaS that allows deployment of cloud applications. The *TwoSpot* PaaS (Wolke and Meixner, 2010), enables hosting multiple, sandboxed Java compatible applications and has a focus on the prevention of vendor lock-in. Tung, Chaw, Xie et al. (2012) propose a highly resilient systems architecture for the cloud with a focus on failures and fallback handling. The auto-scaling algorithm with job deadlines presented by Mao and Humphrey (2011) optimises resource utilisation. In their *ElasticSite* platform, Marshall, Keahey and Freeman (2010) extend non-cloud resources by cloud resources with a focus on different launch policies. This thesis presents a framework that considers the automatic scaling, monitoring and maintaining of components along with an optimised resource utilisation, service composition architecture and caching strategy. None of the related approaches covers the full spectrum that is required to build a WSF as presented in Chapter 4.

3.2.2 Auto Scaling Features of Cloud Providers

Cloud providers offer scaling features directly integrated into their services as illustrated in Figure 3.1. Amazon Web Services (2006) *Auto Scaling* allows to create alarm conditions based on metrics collected by Amazon Web Services (2006) *CloudWatch*. If a metric breaches the threshold of an alarm, actions, such as scaling up or down can be triggered. The Google Cloud Platform (2008) supports the creation of autoscaling groups, where instances can be scaled up and down using the Google Cloud Platform (2008) *Autoscaler*. *Autoscaler* decisions are based on CPU, load or monitoring metrics where the scaling decisions can be configured from multiple policies. Microsoft Azure (2010) cloud services offer autoscaling based on CPU and traffic demands. RackSpace (1998) offers autoscaling features via their *Cloud Control Panel* and API. IBM Bluemix (2014) enables customers to scale horizontally and vertically based on collected metrics. The DigitalOcean (2011) API offers manual scaling features, where metrics such as CPU utilisation can be accessed to program custom policies. All auto scaling features are strictly limited and custom tailored to cloud providers. This thesis identifies and applies a common approach to scaling through the monitoring of performance metrics as shown in Figure 3.1 (a). On a breach of a metric threshold in Figure 3.1 (b), the alarm triggers a provisioning action at Figure 3.1 (c) that is executed by a cloud provider service in Figure 3.1 (d).

3.2.3 Cloud Application Design Patterns

In addition to the fundamental web architecture patterns presented in Section 2.7, further patterns exist to compose cloud applications and information systems. Fehling, Leymann, Retter et al. (2014) identify different types of components, such as stateful, stateless, user interface, processing, batch processing, data access, data abstractor, idempotent processor, transaction-based and timeout-based components for cloud applications. Additionally, the authors classify management components to follow either a provider adapter, managed configuration, elasticity manager, elastic load balancer, elastic queue or watchdog pattern. The management components defined by Fehling, Leymann, Retter et al. (2014) are utilised in management processes, such as the update transition, standby pooling, elasticity, feature flag and resilience management process. Fowler (2009) proposes a pattern called Eager Read Derivation for information systems where the domain logic is split into validations, consequences

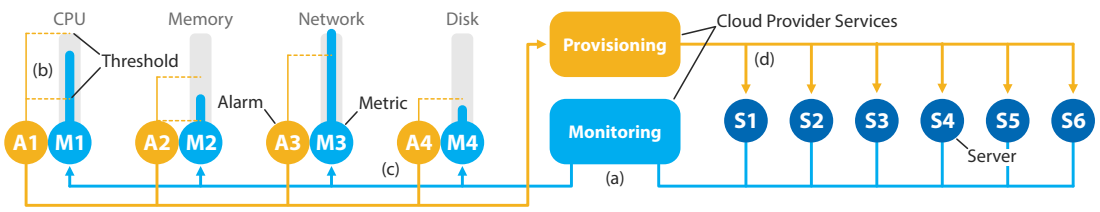


Figure 3.1: Alarm-based auto scaling mechanism as implemented by cloud providers.

Table 3.2: Categorisation of work related to Request Flow.

Request Flow (3.3)	**References**
Caching Strategies and Policies	Le Scouarnec, Neumann and Straub (2014), Qin, Zhang, Wang et al. (2011), Negrão, Roque, Ferreira et al. (2015), Bangar and Singh (2015), Sarhan, Elmogy and Ali (2014), Pettersen, Valvag, Kvalnes et al. (2014), Bocchi, Mellia and Sarni (2014) and Han, Lee, Shin et al. (2012)
Performance Modelling	Han, Ghanem, Guo et al. (2014), Espadas, Molina, Jiménez et al. (2013) and Jiang, Lu, Zhang et al. (2013)
Event Stream Processing	Kroß, Brunnert, Prehofer et al. (2015), Kalashnikov, Bartashev, Mitropolskaya et al. (2015), Wu and Tan (2015), Hummer, Satzger and Dustdar (2013), Apache Samza (2012), Apache Kafka (2012), Apache Hadoop YARN (2011), Apache Storm (2015), Apache Spark (2014), Amazon Kinesis Streams (2014), Amazon Lambda (2014) and Google Cloud Dataflow (2015)

and derivations. The pattern introduces reporting databases to separate query data from the domain model. On this basis, the Command Query Responsibility Segregation (CQRS) pattern identified by Young (2010) and Fowler (2011) splits the conceptual representation of an information system domain into separate models for update and display. The authors compare it to a common Create Read Update and Delete (CRUD) approach, where a single model is used to store and retrieve data. With increasing complexity through the addition of multiple representations of data, the CRUD approach gets more complicated. Hence, Young (2010) and Fowler (2011) propose to split up the model into separate query and command models in order to reduce complexity. Facebook Flux (2014) is an architecture pattern related to CQRS that utilises a unidirectional data flow in order to ensure maintainability of application components. This thesis applies all design patterns reviewed in this section, where the conceptual architecture is presented in Chapter 4.

3.3 Request Flow

In Chapter 5, an optimised request flow based on the convention of a full caching strategy is proposed. This section identifies work related to caching strategies and policies, performance modelling and event stream processing.

3.3.1 Caching Strategies and Policies

As shown in Figure 3.2, caching strategies and policies determine the eviction of cache objects to minimise the cache size and cost. In Figure 3.2 (a), new cache objects are stored in the cache, where due to limited size, other cache objects have to be evicted at Figure 3.2 (b). Eviction is either based on timeouts (Le Scouarnec, Neumann and Straub, 2014), access frequency (Le Scouarnec, Neumann and Straub, 2014) or access patterns (Qin, Zhang, Wang et al., 2011) that are based

on machine-learning algorithms. The *SACS* system proposed by Negrão, Roque, Ferreira et al. (2015) adds a spatial dimension to the cache replacement process by measuring the distance between objects in terms of the number of links necessary to navigate from one object to another and replaces objects distant from the most recently accessed pages. Bangar and Singh (2015) develop a recommendation system based on a K-means algorithm analysing the proxy access log containing entire navigations of web pages by a targeted user and use it to predict upcoming web URIs for pre-fetching. By adding the internal requests generated in each web site as factors to the Least Frequently Used (LFU) and Weighting Replacement Policy (WRP), the replacement approaches proposed by Sarhan, Elmogy and Ali (2014) strengthen the performance of web object caching. The *Jovaku* platform (Pettersen, Valvag, Kvalnes et al., 2014) is a generic database caching layer that relays database operations through the Domain Name System (DNS) protocol utilising the proximity to clients. Bocchi, Mellia and Sarni (2014) and Han, Lee, Shin et al. (2012) find cloud storage prices to continue to drop, hence storing resources in the cloud can be cheaper than the recurring processing of resources on a cache miss. In contrast to the aforementioned approaches, this thesis proposes to explicitly declare all dependencies between resources as shown in Figure 3.2 (c). All requestable web objects are stored in a persistent resource database instead of a volatile cache. Instead of the application of an recommendation system influenced LFU eviction mechanism, a precise update mechanism as shown in Chapter 6 is proposed to keep the Resource Database (RDB) in sync with the data.

3.3.2 Performance Modelling

This section identifies related work in the modelling of load and throughput performance. The elastic scaling approach as modelled by Han, Ghanem, Guo et al. (2014) makes use of cost-aware criteria to detect and analyse the bottlenecks within multi-tier cloud-based applications. The approach is based on monitors that measure the current workload and scale up or down based on the performance. Espadas, Molina, Jiménez et al. (2013) propose the *itesm-cloud* that establishes a formal measure for under- and over-provisioning of virtualised resources in cloud infrastructures specifically for Software as a Service (SaaS) platform deployments and propose new resource allocation mechanisms based on tenant isolation, Virtual Machine (VM) instance allocation and load balancing. Jiang, Lu, Zhang et al. (2013) propose an

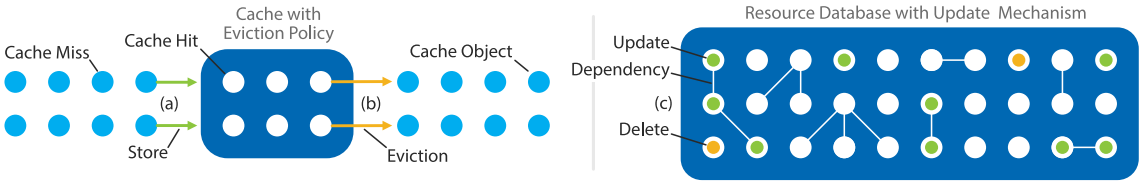


Figure 3.2: A traditional cache with an eviction policy compared to the resource database update mechanism as proposed in this thesis.

optimal VM-level auto-scaling scheme with cost-latency trade-off. The scheme predicts the number of requests based on history data and then gives instructions for service provisioning. In this thesis, the performance modelling is based on request throughput and processing delays. This enables an abstraction of the underlying hardware. CPU or memory boundaries are not present for cloud service components as providers manage service scaling for the customer in the background. Hence, the only universally present measures for performance are the request throughput and the processing delay.

3.3.3 Event Stream Processing Platforms and Frameworks

Event stream processing is a programming paradigm where a stream of events is processed by one or multiple components. This section identifies platforms and frameworks related to the optimised processing flow of a stream of requests, as presented in Chapter 5. Kroß, Brunnert, Prehofer et al. (2015) provide a stream processing platform for Lambda architectures to efficiently use resources and reduce hardware investments. The *Carrera* platform proposed by Kalashnikov, Bartashev, Mitropolskaya et al. (2015) enables to build stream processing workflows that run computations on Microsoft Azure (2010) and export data in real time. *Chrono-Stream* (Wu and Tan, 2015) is a distributed system designed for elastic stateful stream processing in the cloud offering dynamic vertical and horizontal scaling. Hummer, Satzger and Dustdar (2013) provide an overview of the key concepts of stream processing in databases, with special focus on adaptivity and cloud-based elasticity. The reviewed approaches provide general, scalable stream processing platforms that offer the ability to create programmable workflows. This thesis presents not a general platform, but a framework with workflows explicitly dedicated to web scaling. All presented approaches, utilise one or more of the following stream processing frameworks: Apache Samza (2012) is a distributed stream processing framework that uses the distributed publish-subscribe framework Apache Kafka (2012) for messaging, and Apache Hadoop YARN (2011), a framework for distributed computing of large data sets, to provide fault tolerance, processor isolation, security, and resource management. Apache Storm (2015) and Apache Spark (2014) both are distributed, scalable engines for large-scale data processing including stream processing. Amazon Kinesis Streams (2014) is a cloud service to collect and process large streams of data records in real time, where the service automatically scales in the background. Amazon Lambda (2014) is a cloud service that enables to run code on a function level without provisioning or managing servers. Functions can be triggered from incoming requests or a stream of events. Similarly, Google Cloud Dataflow (2015) is a fully-managed cloud service and programming model for batch and streaming big data processing. An implementation of a WSF as proposed in this thesis can utilise any of the reviewed stream processing platforms in order to implement components and request flows.

Table 3.3: Categorisation of work related to Resource Dependency Processing.

Resource Dependency Processing (3.4)	**References**
Job and Workflow Scheduling	Malcolm, Roseboom, Clark et al. (1959), Kelley and Walker (1959), Abrishami, Naghibzadeh and Epema (2012), Chanas and Zieliński (2001), Masdari, ValiKardan, Shahi et al. (2016), Maheshwari, Jung, Meng et al. (2016), Pang, Wang, Cheng et al. (2015), Haeupler, Kavitha, Mathew et al. (2012), Ajwani and Friedrich (2010) and Bellman (1954)
Graph Processing	Batarfi, Shawi, Fayoumi et al. (2015), Ching, Edunov, Kabiljo et al. (2015) and Guo, Biczak, Varbanescu et al. (2014)
Reactive Programming	Salvaneschi, Margara and Tamburrelli (2015), Margara and Salvaneschi (2014) and Salvaneschi and Mezini (2014)
Web Service Measures	Du and Wang (2015), Songwattana, Theeramunkong and Vinh (2014), Rajabi and Wong (2014), Poggi, Carrera, Gavalda et al. (2014), Meusel, Vigna, Lehmberg et al. (2014) and Ramachandran, Kim and Chaintreau (2014)
Traffic Modelling	Dick, Yazdanbaksh, Tang et al. (2014), Chen, Ghorbani, Wang et al. (2014), Zukerman, Neame and Addie (2003), Chen, Addie, Zukerman et al. (2015), Donthi, Renikunta, Dasari et al. (2014), Katsaros, Xylomenos and Polyzos (2012) and Visala, Keating and Khan (2014)

3.4 Resource Dependency Processing

Chapter 6 presents algorithms for optimised resource dependency processing. This section identifies related work in the categories of scheduling problems, graph processing platforms, the reactive programming paradigm, the modelling of web service parameters and traffic modelling.

3.4.1 Job and Workflow Scheduling

Chapter 6 identifies the processing of dependencies as an optimisation problem in the domain of job and workflow scheduling. Job scheduling is an optimisation problem where multiple jobs connected through constraints need to be processed in a valid and fast manner as shown in Figure 3.3. The Program Evaluation and Research Task (PERT) model by Malcolm, Roseboom, Clark et al. (1959) characterises the Research and Development program of the U.S. Navy as a network of interrelated events to be achieved in proper ordered sequence where the duration of the events are time estimates from responsible technical persons. The Critical Path Method by Kelley and Walker (1959) develops a topological representation of a project where each job of the project is represented by an arrow and annotated with an execution time. If the maximum time available for a job equals its duration the job is called critical as it causes a comparable delay in the project completion time. Figure 3.3 shows a processing problem where the dependencies are directed from

left to right and the processing starts at Figure 3.3 (a). When all dependent jobs are processed while maintaining the correct order as shown in Figure 3.3 (a,b,c), the critical path highlights the maximum duration. Abrishami, Naghibzadeh and Epema (2012) propose a new QoS-based workflow scheduling algorithm for utility Grids based on a concept called Partial Critical Paths (PCP), trying to minimize the cost of workflow execution while meeting a user-defined deadline. Chanas and Zieliński (2001) identify that in practice, the activity duration times for critical path planning are not always deterministic and presents two efficient methods of calculation of the path degree of fuzzy criticality. In the domain of workflow scheduling, Masdari, ValiKardan, Shahi et al. (2016) present a comprehensive survey and analysis of scheduling schemes in cloud computing, where a scheduling scheme tries to map the workflow tasks to multiple virtual machines based on different functional and non-functional requirements. Maheshwari, Jung, Meng et al. (2016) present a scheduling strategy that maps workflow tasks to multiple clusters and clouds with optimised balancing. In this thesis, the PERT/Critical Path Problem is transferred from the project management context and applied to dependency processing of web resources. The processing of dependencies are the activities and the critical path duration is the maximum dependency processing duration. This thesis does not apply the concepts of fuzzy criticality or job deadlines as the approximation model presented in Chapter 6 exhibits promising model fits. From workflow scheduling, the basic structures such as a Directed Acyclic Graph (DAG) are used to declare dependent tasks. The major objective in this thesis however, is not to distribute work to multiple virtual machines as presented by Masdari, ValiKardan, Shahi et al. (2016) and Maheshwari, Jung, Meng et al. (2016), but find an optimal forest of processing trees from a dependency graph. The trees are then used to process dependencies in an optimised fashion and calculate the maximum processing duration as critical path. Pang, Wang, Cheng et al. (2015), Haeupler, Kavitha, Mathew et al. (2012) and Ajwani and Friedrich (2010) show that the topological sorting of jobs for correct orderly scheduling can be calculated incrementally in linear time. The dynamic programming method presented by Bellman (1954) solves complex problems by breaking them down into a collection of simpler subproblems if the problems exhibit the properties of an optimal substructure and are overlapping subproblems. In Chapter 6, this thesis combines both topological sorting with dynamic programming on a DAG structure to determine the critical paths of processing.

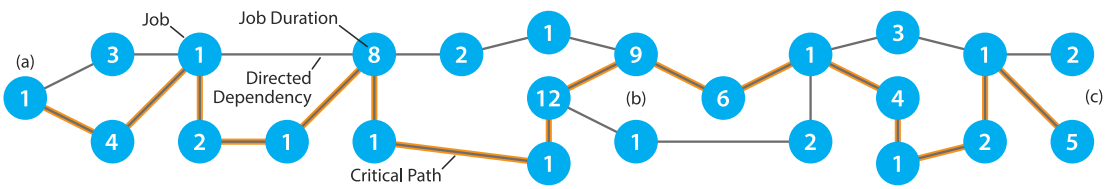


Figure 3.3: A critical path job scheduling problem where jobs run for a specified duration and must run in a constrained order.

3.4.2 Graph Processing Platforms

Large scale graph processing platforms are distributed applications that implement common graph processing algorithms in a scalable fashion. Batarfi, Shawi, Fayoumi et al. (2015) provide a comprehensive survey over the state of the art of large scale graph processing platforms and an experimental study of the *GraphChi*, *Apache Giraph*, *GPS*, *GraphLab* and *GraphX* systems evaluating five common graph processing algorithms. Facebook (Ching, Edunov, Kabiljo et al., 2015) improves the usability, performance and scalability of the *Apache Giraph* system in order to use it on Facebook-scale graphs of up to one trillion edges. Guo, Biczak, Varbanescu et al. (2014) propose and apply an empirical method to obtain a comprehensive performance study of the *Apache Hadoop*, *YARN*, *Stratosphere*, *Apache Giraph*, *GraphLab*, and *Neo4j* platforms. Additionally, the authors implement general statistics, breadth-first search, connected component, community detection and graph evolution algorithms to evaluate them for multiple application domains with sizes up to 1.8 billion edges and tens of GB of stored data. This thesis proposes to store the dependency graph in one of the aforementioned large scale graph processing platforms and use algorithms developed in this thesis to extract a forest of individual dependency trees for each resource. Unfortunately, datasets from Batarfi, Shawi, Fayoumi et al. (2015), Ching, Edunov, Kabiljo et al. (2015) and Guo, Biczak, Varbanescu et al. (2014) can not be used in this thesis as they describe relationships between data entities, e.g. users but not service resources as required for dependency processing. Thus, this thesis utilises custom created service datasets.

3.4.3 Reactive Programming

The reactive programming paradigm presented by Salvaneschi, Margara and Tamburrelli (2015) is oriented around data flows and the propagation of changes and thereby fits the requirements for dependency processing. The *DREAM* middleware by Margara and Salvaneschi (2014) introduces and implements precise semantics and consistency guarantees for reactive programming in distributed environments and studies the overheads introduced by different semantics. Salvaneschi and Mezini (2014) propose a conceptual framework to improve reactive applications in the object-oriented setting which traditionally use event systems and the observer pattern. This thesis uses and applies reactive programming with eventual consistency to automatically update resource dependencies with a middleware approach observing incoming changes through requests.

3.4.4 Web Service Measures

In order to model and evaluate a resource dependency processing algorithm, this section collects and presents related web services measures. Du and Wang (2015) and Songwattana, Theeramunkong and Vinh (2014) improve the cache Hit-Miss

Ratio (HMR) with machine learning approaches and define typical values between 35% and 75% cache hits. Rajabi and Wong (2014) and Poggi, Carrera, Gavalda et al. (2014) characterise the processing time of web applications as a Hyper-Erlang, Weibull, generalised Pareto or Lomax distribution. Meusel, Vigna, Lehmberg et al. (2014) and Ramachandran, Kim and Chaintreau (2014) classify the graph structure in the web to be heavy-tailed and study the in-degree and out-degree distributions of a large web crawls. In this thesis, the cache hit-miss ratio and processing duration is modeled according to Du and Wang (2015), Songwattana, Theeramunkong and Vinh (2014), Rajabi and Wong (2014), Poggi, Carrera, Gavalda et al. (2014), Meusel, Vigna, Lehmberg et al. (2014) and Ramachandran, Kim and Chaintreau (2014). However, the work on graph structures in the web studies the dependencies between multiple services, where this thesis considers the dependencies within a single service. Thus, Chapter 6 presents an analysis of existing service structures.

3.4.5 Traffic Modelling

For the evaluation of a resource dependency processing algorithm, different traffic modelling techniques are reviewed and described in the following section. Dick, Yazdanbaksh, Tang et al. (2014), Chen, Ghorbani, Wang et al. (2014), Zukerman, Neame and Addie (2003), Chen, Addie, Zukerman et al. (2015) and Donthi, Renikunta, Dasari et al. (2014) find that request arrival times exhibit self-similarity and use the following random processes for modelling: Fractionally Autoregressive Integrated Moving-Average (FARIMA), Fractional Brownian Motion (FBM), Poisson Pareto Burst (PPB), Poisson Lomax Burst (PLB) and Circulant Markov-Modulated Poisson (CMMP). For resource popularity as identified by Katsaros, Xylomenos and Polyzos (2012) and Visala, Keating and Khan (2014), the popularity distribution can be modelled using a Zipf distribution with parameter ranges between 0.64 and 0.84. This thesis uses all proposed random processes (FARIMA, FBM, PPB, PLB, CMMP) for modelling the request arrival times and a Zipf distribution for modelling the selection of resources.

3.5 Cloud Portability and Interoperability

In Chapter 7, a cloud portable and interoperable prototype of a WSF is presented. This section identifies work related to the concepts of cloud portability, interoperability and orchestration.

3.5.1 Portability and Interoperability

The National Institute of Standards and Technology (NIST) specifies that the cloud computing model has five essential characteristics: On-demand self-service, broad network access, resource pooling, rapid elasticity and measured service (NIST SP800-145, 2011). To manage the entire lifecycle of infrastructure and applications in

Table 3.4: Categorisation of work related to Cloud Portability and Interoperability.

Cloud Portability and Interoperability (3.5)	**References**
Portability and Interoperability	NIST SP800-145 (2011), Amazon CloudFormation (2011), Google Cloud Deployment Manager (2015), OpenStack Heat (2014), Zhang, Wu and Cheung (2013), Di Martino, Cretella and Esposito (2015) and OpenGroup Cloud Computing Portability and Interoperability (2004)
Deployment and Management Platforms	Binz, Breitenbücher, Kopp et al. (2014), TOSCA v1.0 (2013), OASIS (1993), Fehling, Leymann, Retter et al. (2014), Katsaros, Menzel, Lenk et al. (2014), Qanbari, Li and Dustdar (2014), Haupt, Leymann, Nowak et al. (2014), CAMP v1.1 (2014), Binz, Breitenbücher, Haupt et al. (2013), Kostoska, Gusev and Ristov (2014), Inzinger, Nastic, Sehic et al. (2014), Petcu, Macariu, Panica et al. (2013), Petcu, Martino, Venticinque et al. (2013), Andrikopoulos, Gómez Sáez, Leymann et al. (2014), Loulloudes, Sofokleous, Trihinas et al. (2015) and Ahn, Kim, Seo et al. (2015)
Containers and Cloud Orchestration	Docker (2013), LXC (2008), OpenVZ (2005), Merkel (2014), Amazon EC2 Container Service (2014), Google Container Engine (2014), IBM Containers for Bluemix (2014), Docker Swarm (2015), Apache Mesos (2012) and Google Kubernetes (2014)

the cloud, cloud providers offer interfaces such as Amazon CloudFormation (2011), Google Cloud Deployment Manager (2015) or OpenStack Heat (2014). However, the infrastructure and access to each of the providers is not standardised. Zhang, Wu and Cheung (2013) identify that this prevents both the creation of portable applications that can easily be migrated between providers, and services that are dynamically hosted by multiple cloud providers as shown in Figure 3.4. Di Martino, Cretella and Esposito (2015), OpenGroup Cloud Computing Portability and Interoperability (2004) and NIST SP800-145 (2011) classify between data, system and application portability, and further service, application and platform interoperability. Figure 3.4 shows an overview of the concepts of cloud portability in Figure 3.4 (a,b) and interoperability in Figure 3.4 (c). Data portability enables the transfer of data objects between different cloud platforms. System portability enables the migration of virtual machine instances, machine images and services from one cloud provider to another. Application portability enables the reuse and migration of applications and components across cloud providers. Service interoperability is defined as the ability to use services across multiple cloud platforms through a unified management interface. Application interoperability is defined as the ability of cloud-enabled applications to collaborate across different platforms. Platform interoperability is defined as the ability of platform components to interoperate. The WSF prototype developed in Chapter 7 is both cloud portable and interoperable.

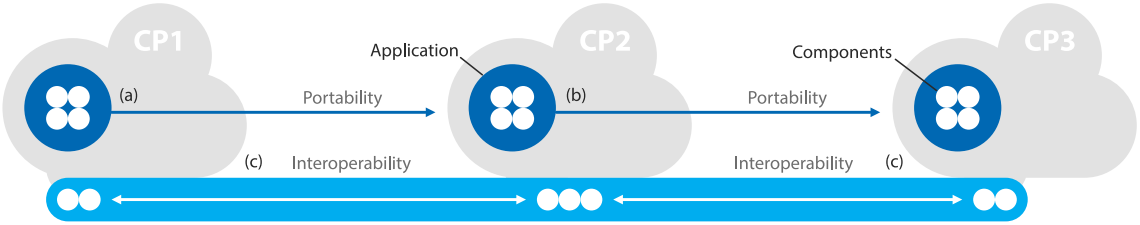


Figure 3.4: The concept of cloud portability and interoperability.

On the service level, a standardised container engine is used where Chapter 7 implements the provider adapter pattern for the component interface as defined in Chapter 4. On the application level, the universally applicable application interface defined in Chapter 4 is implemented in Chapter 7 to ensure application portability.

3.5.2 Cloud Application Deployment and Management Platforms

Cloud application deployment and management platforms enable the management and provisioning of cloud infrastructure. The Topology and Orchestration Specification for Cloud Applications (TOSCA) by Binz, Breitenbücher, Kopp et al. (2014) and TOSCA v1.0 (2013) is a standard from the OASIS (1993) non-profit consortium. It provides a standardised, well-defined, portable, interoperable and modular exchange format for the structure of application components, the relationships among them, and their corresponding management functionalities. As identified by Fehling, Leymann, Retter et al. (2014), the specification is a cloud application management pattern. According to Katsaros, Menzel, Lenk et al. (2014), Qanbari, Li and Dustdar (2014) and Haupt, Leymann, Nowak et al. (2014), the specification defines a holistic way for cloud portability, interoperability and orchestration. The OASIS (1993) Cloud Application Management for Platforms (CAMP) specification (CAMP v1.1, 2014) is designed specifically for the PaaS service model where services for different programming languages and application frameworks are offered. Binz, Breitenbücher, Haupt et al. (2013), Katsaros, Menzel, Lenk et al. (2014) and Kostoska, Gusev and Ristov (2014) have successfully implemented runtimes for TOSCA containers to test the actual interoperability and portability features. With *MADCAT* by Inzinger, Nastic, Sehic et al. (2014), the authors introduce a methodology enabling the structured creation of cloud applications, addressing the complete application development lifecycle. The Open Source API and Platform for Multiple Clouds project (mOSAIC) by Petcu, Macariu, Panica et al. (2013) and Petcu, Martino, Venticinque et al. (2013) proposes a vendor-agnostic and language-independent set of open APIs supporting portability between clouds. Andrikopoulos, Gómez Sáez, Leymann et al. (2014) create a technology-agnostic formal framework that models, verifies and generates alternative scenarios for the distribution of an application stack across cloud offerings and evaluate the optimal scenario given the application needs in terms of operational expenses. Loulloudes, Sofokleous, Trihi-

nas et al. (2015) present an open source Cloud Application Management Framework (CAMF) based on the Eclipse Rich Client Platform which facilitates cloud application lifecycle management in a vendor-neutral way. Ahn, Kim, Seo et al. (2015) extract a set of requirements from the examination of the existing rules and policies in the public sector. This thesis does not provide an application management platform for the cloud. It however depends on standards, such as TOSCA in order to implement component interfaces in a reusable, portable and interoperable fashion.

3.5.3 Containers and Cluster Orchestration Frameworks

Linux containers enable a portable and efficient deployment of components that must be orchestrated over multiple hosts or cloud services. Container implementations, such as Docker (2013), LXC (2008) and OpenVZ (2005) provide a virtualisation on the operating system level. In contrast to virtual machines, a container does not need a hypervisor to virtualise hardware and all containers run in the same kernel (Merkel, 2014). This makes containers an efficient alternative to virtual machines. Cloud providers offer container services, such as Amazon EC2 Container Service (2014), Google Container Engine (2014) and IBM Containers for Bluemix (2014) that are able to run containers in a universally defined format with cloud portable and interoperable APIs. In this thesis, Docker (2013) is used to implement the prototype of a WSF in Chapter 7. In order to orchestrate a multitude of containers, Docker Swarm (2015) is capable of turning a group of Docker engines on multiple hosts into a single, virtual Docker engine. This enables to start containers as if they were running on a single host and provides a unified API for orchestration. Apache Mesos (2012) abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to easily be built and run effectively. It supports applications such as Apache Hadoop YARN (2011), Apache Spark (2014) and Apache Kafka (2012), allows to run custom Docker containers and provides a unified API for resource management. Google Kubernetes (2014) is an open source orchestration system for Docker containers that handles scheduling onto nodes in a compute cluster and actively manages workloads. This thesis uses Docker Swarm (2015) in the implementation of the WSF prototype in Chapter 7, as the increased orchestration capabilities provided by Apache Mesos (2012) and Google Kubernetes (2014) are not needed for the prototype. In production however, all cluster orchestration frameworks can be used.

3.6 Open Research Questions

Table 3.5 presents the open research questions extracted from the literature review in this chapter. The research questions are categorised by the sections in this chapter that also represent the research chapters in this thesis. Each research question is

Table 3.5: Summary of open research questions.

ID	**Web Scaling Frameworks** (3.2)
R4.1	What are the design goals for a conceptual architecture design?
R4.2	Which cloud architecture design patterns can be applied?
R4.3	Which modules are needed to enable automatic scaling?
R4.4	Which parameters are needed to manage scaling?
R4.5	Which minimal set of interfaces needs to be designed and implemented?
ID	**Request Flow** (3.3)
R5.1	Which design pattern can be used for optimised resource management?
R5.2	How can components be composed and requests be flowed efficiently?
R5.3	Which interface extension is required to enable the optimised scheme?
R5.4	How can the performance be modelled analytically?
R5.5	Which metric enables to operate an application with optimal load?
R5.6	How is the real-world application performance of the proposed scheme compared to a traditional approach?
ID	**Resource Dependency Processing** (3.4)
R6.1	How can resource dependencies be measured and stored?
R6.2	What algorithm can be used to optimise the performance of processing?
R6.3	What effects have dependency graph measures on the performance?
R6.4	How can resource dependencies be generated?
R6.5	How well can the dependency processing duration be modelled?
R6.6	How is the performance compared with a typical traditional processing approach?
ID	**Cloud Portability and Interoperability** (3.5)
R7.1	How can modules and components be designed in a portable and interoperable fashion?
R7.2	What needs to be done in order to integrate a traditional app into a WSF?
R7.3	How well can the processing cost and storage space required for dependency processing be modelled?
R7.4	How is the performance trade-off between processing cost and processing duration when using a dependency processing approach?

assigned a unique ID, where the number preceding the period represents the chapter in which the research question is considered.

3.7 Theoretical Foundations

The novel contributions and ideas presented in this thesis are primarily based on three theoretical foundations: Parallel computing, queueing theory and graph theory. The parallel computing paradigm is based on the principle that a large computing problem can be simplified by being divided into pieces and then individually solved in parallel. For the scaling of web services this means that a stream of requests that is too large to compute for a single server is divided into multiple streams, where each server can operate on an individual, smaller sub-stream. Chapter 5 presents a further enhancement to web service scalability by dividing the streams of requests based on the type of the requests, which allows a parallel and decoupled execution

of read and write requests. The modelling in Chapter 5 has its foundations in the queueing theory. Little's Law (Little, 1961) states that the average number of customers in a queue equals the arrival rate multiplied by the time spend in the queue. For the performance modelling, where cloud provider services are abstracted as components, this means that the average number of requests in a component denoted as concurrency equals the average processed requests per second denoted as flow multiplied by the processing delay of the component. Finally, Chapter 6 uses graph theory for modelling the resource dependencies. The resources are represented by nodes and the dependencies by edges. Further, shortest-path and longest-path algorithms are used to evaluate properties of the dependencies and a topological sort of the graph is used to determine the correct processing order.

3.8 Summary

This chapter has presented a literature review covering a large number of recent relevant works and theoretical foundations of the project. It was first shown, that none of the reviewed platforms and frameworks for scaling covers the full spectrum of features required to efficiently scale web applications in a reusable fashion. All reviewed auto scaling features are strictly limited and custom tailored to cloud providers impeding the creation of reusable scaling workflows. None of the reviewed caching strategies and policies considers an explicit declaration of dependencies between resources, although research suggests decreasing cloud storage prices. In regard to performance modelling, the reviewed works use CPU, memory, networking and input/output measures to evaluate performance, where this measures are not exposed by managed services of cloud providers. Event stream processing frameworks provide general platforms to distribute computing on huge clusters, however there are no standardised frameworks that enable the processing of web requests. In terms of resource dependencies, none of the reviewed work presents an approach where dependencies within a single service are explicitly evaluated, defined and processed in an optimised fashion. There exists a range of large scale graph processing platforms that enable a distributed and scalable processing of dependency graphs. None of the reviewed works applies a reactive programming model to automatically update resource dependencies. Finally, the concepts of cloud portability and interoperability have been discussed. While all of the reviewed platforms and frameworks enable the creation of reusable workflows to orchestrate infrastructure, none of them provides interfaces to directly operate web applications nor has knowledge of an optimised request flow.

4. Conceptual Architecture Design

4.1 Overview

In this chapter, a conceptual architecture design including required modules, interfaces, parameters and components valid for all implementations of WSFs is proposed. In order to ensure feasibility of the presented conceptual design, the subsequent Chapters 5, 6 and 7 propose and evaluate implementations of WSF key mechanisms. However, the conceptual design in this chapter is intended to serve as a basis for a novel class of frameworks for scalable web services in cloud environments, where different implementations can be optimised for special use cases. The literature review in Chapter 3 identified five open research questions related to the conceptual architecture design of WSFs:

- **R4.1**: What are the design goals for a conceptual architecture design?
- **R4.2**: Which cloud architecture design patterns can be applied?
- **R4.3**: Which modules are needed to enable automatic scaling?
- **R4.4**: Which parameters are needed to manage scaling?
- **R4.5**: Which minimal set of interfaces needs to be designed and implemented?

Each of the subsequent sections considers one of the open research questions. Finally, Section 4.7 provides answers to the aforementioned research questions and Section 4.8 concludes the chapter and puts it in a greater context.

4.2 Proposed Architecture Overview and Design Goals

This section gives an initial overview of the proposed architecture and is further dedicated to the first open research question **R4.1**: *What are the design goals for a conceptual architecture design?*

4.2.1 Design Goals

The major design goal is to create an architecture that enables to build maintainable, automatable, scalable, resilient, portable and interoperable implementations of WSFs. Firstly, components have to automatically adapt to dynamic workloads. Fehling, Leymann, Retter et al. (2014) identify this as elastic management process. Secondly, all communication with cloud providers needs to be abstracted in order to ensure portability and enable interoperability. Thirdly, components have to be checked for failures and replaced automatically. Fehling, Leymann, Retter et al.

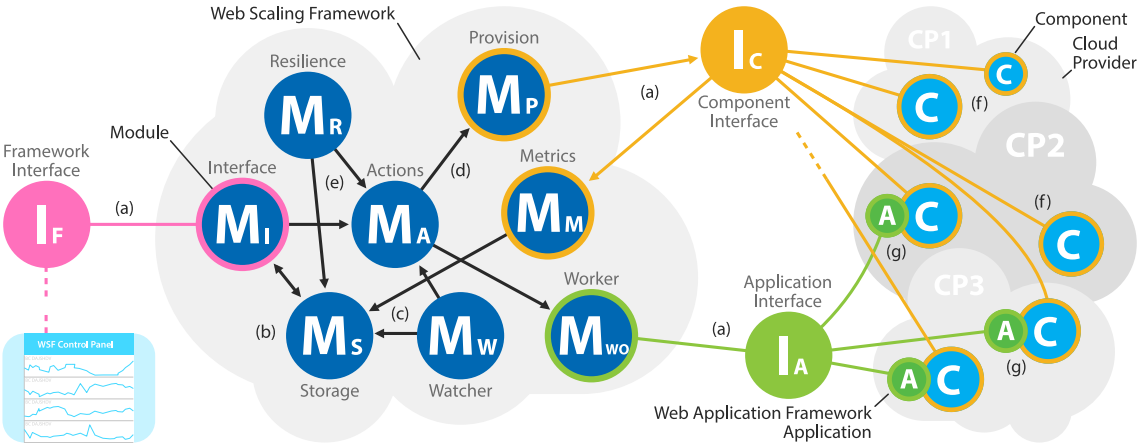


Figure 4.1: Architecture overview of the proposed WSF that manages multiple components and applications hosted by different cloud providers.

(2014) identify this as resiliency management process. Finally, a WSF implementation needs to be maintainable and scalable.

4.2.2 Proposed Architecture Overview

Figure 4.1 presents an initial overview of the proposed architecture that is used throughout this chapter. The design patterns, modules and interfaces shown in Figure 4.1 are elaborated on in the subsequent Sections 4.3, 4.4 and 4.6. The cloud on the left side of Figure 4.1 presents the logical structure of a WSF, where the modules within this cloud implement the core functionality of a WSF. The right side of Figure 4.1 shows the components that are managed by a WSF. The components provide services and functionalities needed to operate a full web application. One type of component is the worker component at Figure 4.1 (g), which hosts the application logic that is implemented with the help of a WAF. The worker component joins a WSF with a WAF, where the worker logic is implemented by the WSF and the application logic is implemented by the WAF.

4.3 Applied Cloud Architecture Design Patterns

The literature review in Chapter 3 identified multiple cloud architecture management patterns (Fehling, Leymann, Retter et al., 2014) and architectural styles (Young, 2010; Fowler, 2011). In combination with the aforementioned design goals, this opens up research question **R4.2**: *Which cloud architecture design patterns can be applied?*

4.3.1 Provider Adapter Pattern

Figure 4.1 (a) shows how modules implement interfaces that connect to modules and components outside a WSF. This is done in order to provide a unified interface to other modules inside a WSF. The underlying design pattern is identified as provider adapter pattern (Fehling, Leymann, Retter et al., 2014). For each used cloud

provider component, a provider adapter needs to be created. The provider adapter transforms unified commands specified in the interface, so they can be executed with the cloud provider's API. Regarding the design goals for the architecture, this ensures the portability and interoperability of the framework. Further details on interfaces are given in Section 4.6.

4.3.2 Managed Configuration Pattern

Figure 4.1 (b) shows the storage of a managed configuration as identified by Fehling, Leymann, Retter et al. (2014). In order to configure components in an automatable fashion, the configuration is persisted in a central storage. The configuration is used to provision new components as well as to adapt existing components. Regarding the design goals for the architecture, this enables a full automation of provisioning. Further details on the store module are given in Section 4.4.

4.3.3 Elastic Manager Pattern

In Figure 4.1 (c), the application of both the elastic manager pattern (Fehling, Leymann, Retter et al., 2014), and the observer pattern is shown. Elastic management describes the continuous provisioning of components based on utilisation metrics. Through the continuous observation of these utilisation metrics as shown in Figure 3.1, algorithms in the manager determine the optimal number of component instances. Based on the results of the algorithms and the current provisioning status, a necessary change in provisioning is calculated. The manager itself does however not carry out the changes. Regarding the design goals for the architecture, this adapts components automatically to dynamic workloads. This thesis presents an elastic management algorithm based on the optimal request flow in Chapter 5. Further details on the watcher module are given in Section 4.4.

4.3.4 Command Query Responsibility Segregation and Flux Pattern

Figure 4.1 (d) shows the application of the CQRS pattern as identified by Young (2010) and Fowler (2011). Additionally, Figure 4.1 (d) shows the application of the CQRS-related Flux pattern as presented by Facebook Flux (2014). In the CQRS pattern, the command and the query model are segregated in order to decouple domains. This allows to use different models for updating and querying to maximise performance and scalability, provide a higher data model flexibility and simplify complex data structures. A query model can provide much more data with complex calculated data than the corresponding command model. This allows simple updates with fewer data while providing detailed data on query at the same time. The proposed architecture at Figure 4.1 (d) has segregated domains for retrieving data through the metrics module, query data through the storage module and execute commands through the actions module. The Flux pattern (Facebook Flux, 2014) defines a unidirectional data flow where commands always enter the system as

actions that are dispatched through the system. The proposed architecture exhibits the same feature by allowing all provisioning actions strictly through the actions module. Regarding the design goals for the architecture, this ensures the maintainability of a WSF implementation. Further details on the action module are given in Section 4.4.

4.3.5 Watchdog Pattern

In Figure 4.1 (e), the application of both the watchdog pattern (Fehling, Leymann, Retter et al., 2014), and the observer pattern is shown. The watchdog pattern ensures that in case of a failure or unavailability of a component the error is handled automatically without human intervention. Based on monitored metrics, actions to reprovision, reload or rollback components are triggered. Regarding the design goals for the architecture, this ensures the resiliency of a WSF implementation. Further details on the resilience module are given in Section 4.4.

4.3.6 Microservice Architecture Pattern

Figure 4.1 shows the application of the microservice architecture pattern as identified by Fowler (2014) and Namiot and Sneps-Sneppe (2014). The presented architecture proposes to structure all modules as a suite of small web services. Each service component implements its autonomous module functionality that can utilise a self-reliant service database. Regarding the design goals for the architecture, this ensures the scalability of a WSF implementation. Further details on the implementation of service modules are given in Chapter 7.

4.4 Modules Specification

The implementation of the aforementioned applied architecture design patterns opens up research question **R4.3**: *Which modules are needed to enable automatic scaling?* Hence, this section proposes a minimal module structure specification and elaborates on the interplay in relation to the design goals. Table 4.1 provides an overview of all proposed modules required to implement a WSF with the specified design goals, however it can be extended by individual implementations of WSFs.

4.4.1 Storage Module

The storage module M_S provides a shared data access component (Fehling, Leymann, Retter et al., 2014). The storage is used to store component management configurations, metrics retrieved by the metrics module M_M and parameters set by the framework interface module M_I. Additionally, it provides read access for the watcher module M_W that observes component metrics and the framework interface module M_I that presents the current status of the system. The implementation can be done using an IaaS compute unit with a custom database, or via a SaaS such

Table 4.1: Modules of a Web Scaling Framework.

Module	Description
Storage Module (M_S)	Provides a shared data access for component management configurations, component metrics and framework parameters.
Metrics Module (M_M)	Continuously collects metrics such as performance, failures and availability from components.
Watcher Module (M_W)	Observes performance metrics and triggers provisioning actions.
Resilience Module (M_R)	Observes failure and availability metrics and triggers actions to maintain component state.
Actions Module (M_A)	Defines available framework actions for component provisioning and application deployment.
Provision Module (M_P)	Provides a unified interface for cloud provider communication and adapts cloud provider APIs.
Interface Module (M_I)	Implements the framework interface to allow a unified management from external sources.
Worker Module (M_{Wo})	Processes requests with the help of the web application that is implemented using a WAF.

as Amazon Web Services (2006) *DynamoDB*, Amazon Web Services (2006) *Relational Database Service*, Google Cloud Platform (2008) *Datastore* or Google Cloud Platform (2008) *SQL*.

4.4.2 Metrics Module

The metrics module M_M continuously collects metrics from the components and passes them to the storage module M_S for persistence. In order to be able to communicate with a multitude of cloud providers, the module implements the parts of the component interface I_C that consider metrics retrieval. The module contains a scheduling part that orchestrates collection frequencies and multiple provider adapters (Fehling, Leymann, Retter et al., 2014) that connect to cloud providers. The implementation can be created with a custom IaaS compute unit that connects with an existing SaaS such as Amazon Web Services (2006) *CloudWatch* or Google Cloud Platform (2008) *Monitoring*.

4.4.3 Watcher Module

The watcher module M_W observes the components metrics retrieved from the storage module M_S and triggers provisioning actions via the actions module M_A. The algorithm in the module continuously compares component provisioning metrics, such as the number of machines in a component, with the current performance metrics and targeted available performance metrics determined by the framework configuration. One implementation of such an algorithm is given in Chapter 5 of this thesis. Available parameters for configuring the targeted performance are elaborated in Section 4.5. For the implementation, a custom component hosted by a IaaS needs

to be created as no provider IaaS implementations exist at the time of writing.

4.4.4 Resilience Module

The resilience module M_R observes the components failures, availability and metrics it retrieves from the storage module M_S. If a component is not reachable by network, shows suspiciously high processing delays, reports hardware errors or is detected to be misconfigured, the resilience module detects and acts on these system changes to ensure a continuous operation of the system. Therefore, it automatically triggers actions via the actions module M_A, such as restarting a component, moving a component to another cloud provider or deploying another version of a component with a different configuration that is known to work from a previous operation. For the implementation of the module, a custom component hosted by an IaaS needs to be created as no cloud provider IaaS implementations exist with the required functionality at the time of writing.

4.4.5 Actions Module

The actions module M_A serves as a single origin for component provisioning (Facebook Flux, 2014; Young, 2010; Fowler, 2011) and the application update transition process as identified by Fehling, Leymann, Retter et al. (2014). Only the watcher module M_W, the framework interface module M_I and the resilience module M_R can trigger actions. Actions are defined as a set of API calls and include actions to:

- Increase or decrease component performance by x units
- Start and stop entire component
- Move parts of or entire component to cloud provider y
- Deploy or upgrade application z
- Start and stop collection of performance metrics

The actions module dispatches component related actions to the provision module M_P and deployment related actions to the worker module M_W. For the implementation, a custom framework component needs to be created and hosted by a PaaS.

4.4.6 Provision Module

The provision module M_P implements the provisioning part of the component interface I_C. It provides a unified interface for cloud provider communication by implementing a multitude of provider adapters as identified by Fehling, Leymann, Retter et al. (2014). For the implementation, a custom component deployed as IaaS compute unit can be used. The component can utilise existing SaaS such as Amazon Web Services (2006) *CloudFormation* or Google Cloud Platform (2008) *Deployment Manager*.

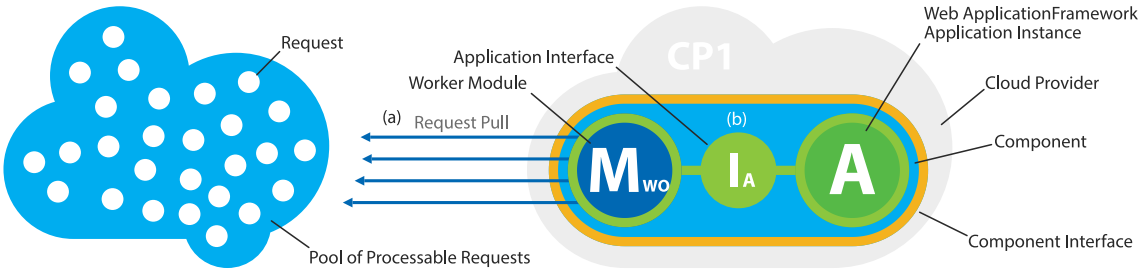


Figure 4.2: Overview of the worker component that is joining the worker module and the web application.

4.4.7 Interface Module

The interface module M_I implements the framework interface I_F to allow a unified management of the framework from external sources. The module allows to create, read, update and delete component configurations, framework parameters and failure handling mechanisms via the storage module M_S. Additionally, it is able to dispatch actions to the actions module M_A for optional manual triggering of component provisioning and application deployment. For the implementation, a custom component deployed as IaaS compute unit can be used.

4.4.8 Worker Module

The worker module M_P runs inside a component and implements the application interface I_A as illustrated in Figure 4.2. At Figure 4.2 (a), the module pulls several request from a pool of requests awaiting processing, e.g. a request queue. It then processes the requests at Figure 4.2 (b) using an instance of the web application that is implemented using a traditional WAF. The number and concurrency of requests to be processed is determined according to the optimal performance range elaborated in Chapter 5. For the implementation, a Linux container component in conjunction with PaaS cloud processing platforms such as Amazon Web Services (2006) *Container Service* or Google Cloud Platform (2008) *Container Engine* can be used.

4.5 Scaling Parameters

Algorithms in a WSF base their scaling decisions on various parameters. This opens up research question **R4.4**: *Which parameters are needed to manage scaling?* This section identifies three types of parameters: Component parameters, system parameters and traffic parameters, where Table 4.2 provides an overview of all three categories.

4.5.1 Component Parameters

Component parameters describe the performance metrics, provisioning state, failures and availability data that is collected for each component, respectively. Exemplary

Table 4.2: Framework parameters to manage a Web Scaling Framework.

Component Parameters	**Description**
Machines	The current number of machines in a component.
Target Machines	The targeted number of machines in a component.
Request Flow/s	The number of requests that flow through the component per second.
Target Flow/s	The desired target number of requests that flow through the component per second.
Concurrency	The current number of parallel component requests that are not responded yet.
Optimal Concurrent Range	The concurrency range for which the component delivers optimal flow.
Processing Delay	The average time it takes the component to process a request.
System Parameters	**Description**
Sensitivity	Influences the aggressiveness of the scaling actions.
Headroom	The percentage of over-provisioning available for load spikes.
Machines	The current number of total machines in all components.
Maximum Machines	The maximum number of total machines in all components.
Traffic Parameters	**Description**
Requests/s	The current number of incoming requests per second.
Target Flow/s	The desired target number of incoming requests per second.
Concurrency	The current number of parallel incoming requests that are not responded yet.

components are load-balancers, databases, queues, event propagation systems, graph processing systems and the aforementioned worker component that contains web application logic. As shown in Table 4.2, component parameters keep track of the current number of component machines and provide storage for the watcher module M_W to save the targeted number of machines. The request flow through the component describes the number of requests that enter and leave a component per second. Similar to the target number of machines, the target request flow describes the desired number of requests the component should be able to handle in order to trigger provisioning actions. In addition to a concurrency parameter, a component needs to provide the optimal concurrency range that is described in detail in Chapter 5. The mean processing delay is needed in the watcher module M_W to provision the component and in the resilience module M_R to ensure availability and failure states. Chapter 5 further elaborates on the component parameters with a detailed explanation of their influences on performance calculation and management.

Table 4.3: Minimum viable interfaces for a Web Scaling Framework.

Component Interface	**Description**
Metrics Interface	Collects performance, failure and availability data.
Provision Interface	Adds/removes machines or increases/decreases Input/Output Operations Per Second (IOPS).
Framework Interface	**Description**
Configuration Interface	Creates and updates components and their configurations.
Parameter Interface	Manages system parameters for scaling.
Action Interface	Lists and triggers available framework actions.
Application Interface	**Description**
Deployment Interface	Manages the lifecycle of a web application.
Request Flow Interface	Sets the origin for requests and target for responses.

4.5.2 System Parameters

System parameters influence the overall behaviour of a WSF. Due to the dynamic nature of web traffic, load often exposes bursts that heavily influence the short time trend of the load. Thus, scaling actions can be delayed for a configurable amount of time so the system is not over-provisioned. The sensitivity parameter influences this aggressiveness of the provisioning actions. Further, as shown in Table 4.2, the headroom parameter describes the percentage of over-provisioning. Similar to the sensitivity parameter, this is done to be able to process requests with some buffer that can be used for instantaneous load bursts. The machines and maximum number of machines parameters are used to keep track of the current total number of machines and set a limit on the maximum number of machines in the case of an unexpected increase of traffic.

4.5.3 Traffic Parameters

Traffic parameters keep track of the total number of incoming requests per second and the total number of requests that are currently processed. Additionally, they allow to specify a target flow the system needs to be able to handle. This can be used to prepare the infrastructure in the case of a planned increase of traffic, e.g. when a certain social event is taking place.

4.6 Minimum Viable Interfaces

In order to communicate with the framework, components and the application, a minimal set of interfaces is proposed. An instance of a WSF has the obligation to implement at least the minimal set of interfaces. If the minimal set is not implemented, the WSF is not viable for scaling. Thus, this thesis refers to the minimal set as the Minimum Viable Interfaces (MVI), where Table 4.3 provides an overview of all interface categories. The MVI however, can be extended in order to include special functionality that differentiates WSFs from another. This opens up research

question **R4.5**: *Which interfaces need to be designed and implemented?* The interfaces are designed to follow the REST architectural style as defined in Chapter 2 and the exchanged format is JSON. This allows all modules implementing the interfaces to be created as independent microservices that can be managed and scaled individually.

4.6.1 Component Interface

As shown in Figure 4.1, the component interface I_C abstracts communication between cloud provider components and framework modules. Consequently, the component interface is further subdivided into a metrics interface that is implemented by the metrics module M_M and the provision interface that is implemented by the provision module M_P.

4.6.1.1 Metrics Interface

The metrics module M_M collects performance, failure and availability data from the components and implements a provider adapter pattern as identified by Fehling, Leymann, Retter et al. (2014). Thus, the metrics interface defines the actions the metrics module M_M needs to implement and adapt to the provider. For the WSF proposed in this thesis, the minimal list of components required includes the load-balancer LB, the dispatcher D, the resource storage RS, the queue Q, the worker W and the event sytem ES components. For other implementations of WSFs however, this list can be expanded. The proposed action and sample response to collect performance data is defined as follows:

```
GET /:componentId/performance
{
  "lastUpdated": 1452252926,
  "machines": 3,
  "iops": 10000,
  "requestsPerSecond": 8563,
  "concurrency": 480,
  "meanProcessingDelayInMs": 23,
  "optimalConcurrencyRange": [65, 925]
}
```

The machines and IOPS field depend on the type of component. A PaaS component often can be provisioned using IOPS to abstract the actual number of machines that run in the background while achieving the desired throughput. The optimal concurrency range is a metric to identify the optimal load curve of a component. It is elaborated in Chapter 5. The rest of the parameters are strictly based on the component parameters defined in the previous section.
To collect failures, the proposed interface action is defined as:

```
GET /:componentId/failures
{
  "lastUpdated": 1452252926,
  "notifications": [
    {
      "id": "1buysgfut",
      "date": 1452252302,
      "type": "os.update.available",
      "message": "Kernel update 4.3 is available, 4.25 is installed."
    },
    ...
  ],
  "errors": [
    {
      "id": "ut82v3v",
      "date": 1452252568,
      "type": "memory.full",
      "message": "15.98GB of 16GB of Memory is used."
    },
    ...
  ],
}
```

The response illustrates a list of notifications and errors. Notifications are informative messages that can be acted on. Errors are severe failures that must be acted on. Each failure type helps to identify the type of the occured incidence, where the message gives details in a human readable form.
To collect the availability of the component, an action representing a time series of heartbeats is proposed as follows:

```
GET /:componentId/availability
{
  "lastUpdated": 1452252926,
  "delayInMs": 13,
  "responded": true,
  "timeseries": [
    {
      "id": "8sfydg",
      "date": 1452252926,
      "delayInMs": 13,
      "responded": true
    },
```

```
    {
      "id": "7g87gs",
      "date": 1452252925,
      "delayInMs": 19,
      "responded": true
    },
    {
      "id": "7gf7g7",
      "date": 1452252924,
      "delayInMs": 0,
      "responded": false
    },
    ...
  ]
}
```

The delay defines the healthiness of the component. This is achieved by sending a heartbeat to the component and measuring the time it takes the component to respond. If the component does not respond within a defined timeout, it sets the responded field to false in order to highlight the unavailability of the component.

4.6.1.2 Provision Interface

The provision module M_P manages the performance by adding/removing machines or increasing/decreasing the IOPS that control the provisioned throughput of certain PaaS, such as Amazon Web Services (2006) *DynamoDB* or Google Cloud Platform (2008) *Storage* with on-demand I/O. The proposed action with the parameter body sent to provision a component is defined as follows:

```
PUT /:componentId/provision
{
  "machines": 4,
  "iops": 20000,
}
```

As for the metric collection, the machines and IOPS field depend on the type of component. For a PaaS-provisioned component the IOPS field is the key metric and for machine based components, the machines field is used. The provisioning is executed via an extra resource that reflects the current provisioning target. The target can deviate from the metrics collected via the metrics interface, as provisioning does not happen instantaneous but is a process that takes a certain amount of time.

4.6.2 Framework Interface

As shown in Figure 4.1, the framework interface I_F enables a unified communication and management of the framework from external sources. The interface exposes the current state of all component configurations, allows to set system parameters that influence the scaling and enables to manually trigger actions to provision and deploy applications. Consequently, the framework interface is further subdivided into a configuration interface, a parameter interface and an action interface.

4.6.2.1 Configuration Interface

The storage module M_S persists metrics and machine configurations for all components. As to the metrics interface, the minimal list of components for the WSF proposed in this thesis includes the load-balancer LB, the dispatcher D, the resource storage RS, the queue Q, the worker W and the event sytem ES components, where other implementations can expand on this list. To create or update a component the interface action is defined as follows:

```
GET/PUT /components/:componentId
{
  "id": "f78g236ffsd",
  "type": "loadbalancer",
  "name": "Europe Load Balancers",
}
```

Each component has a unique id and type field that expresses the component category. Additionally, each component has a name field to make it identifiable by humans. In order to retrieve all registered components, the interface action is proposed as follows:

```
GET /components
[
  {
    "id": "f78g236ffsd",
    "type": "loadbalancer",
    "name": "Europe Load Balancers",
    "configuration": {
      "type": "service",
      "provisioning": "iops",
      "provider": "aws.elasticloadbalancer"
    },
    "metrics": {
      "lastUpdated": 1452252926,
      "machines": 0,
```

```
      "iops": 10000,
      "requestsPerSecond": 8563,
      "concurrency": 480,
      "meanProcessingDelayInMs": 23,
      "optimalConcurrencyRange": [65, 925]
   }
  },
  {
    "id": "9o7sgeryyuf",
    "type": "worker",
    "name": "Northamerica Workers",
    "configuration": {
      "type": "container",
      "provisioning": "machines",
      "provider": "gcp.containerengine"
    },
    "metrics": {
      "lastUpdated": 1452252945,
      "machines": 9,
      "iops": 0,
      "requestsPerSecond": 34087,
      "concurrency": 8876,
      "meanProcessingDelayInMs": 104,
      "optimalConcurrencyRange": [30, 4320]
   },
   ...
  }
]
```

The metrics object represents exactly the metrics defined in the component interface I_C. The configuration object represents exactly the configuration interface that is defined as follows:

```
GET/PUT /:componentId/configuration
{
  "type": "service",
  "provisioning": "iops",
  "provider": "aws.elasticloadbalancer"
}
```

The type field specifies the kind of the service and the provisioning field specifies whether the component is provisioned with machines or IOPS. The provider field identifies the cloud provider and service the component is running on.

4.6.2.2 Parameter Interface

The parameter interface exposes actions to read and manage system parameters as defined in Section 4.5. The interface actions to read and update are proposed as follows:

```
GET/PUT /parameters
{
  "sensitivity": 1,
  "headroom": 5,
  "maxTotalMachines": 80
}
```

4.6.2.3 Action Interface

The action interface provides a list of actions from the actions module M_A that can be triggered using the interface module I_M. To list all available actions, the interface is proposed as follows:

```
GET /actions
[
  {
    "id": 16ffksd,
    "description": "Updates the number of machines for a component"
    "uri": "/actions/provisionComponentMachines",
    "parameters": {
      "componentId": "string",
      "machines": "integer"
    }
  },
  {
    "id": hjf6g3,
    "description": "Pulls the latest version of an application"
    "uri": "/actions/deployApplication",
    "parameters": {
      "componentId": "string",
      "applicationId": "string"
    }
  },
  ...
]
```

Each action provides a description for human identification, e.g. in a user interface. The URI field points to the resource that executes the action. The parameter field provides the necessary parameters needed to trigger the action.

In order to trigger one of the listed actions, e.g. the deployment of an application, the interface is defined from the list as follows:

```
POST /actions/deployApplication
{
  "componentId": "7g2783g",
  "applicationId": "98g237j"
}
```

4.6.3 Application Interface

As shown in Figure 4.1, the application interface I_A abstracts communication between the worker module M_{WO} and the web application that is created using a traditional WAF. Additionally, it defines actions for the deployment of the application. Consequently, the application interface is further subdivided into a deployment and a request flow interface that both are implemented by the worker module M_{WO}.

4.6.3.1 Deployment Interface

Each worker module M_{WO} that runs in a component manages exactly one instance of a web application. The deployment interface defines all actions needed to control the lifecycle of the application. In order to deploy an application to the latest version, the interface proposes an action as follows:

```
POST /deploy
{
  "version": "1.3.2.7",
  "hash": "1530b1ff31fa4b16299470f88d1279483ad06fdd",
  "repository": "ssh://deploy@repos:app1.git"
}
```

The body specifying the exact version, hash of a Version Control System (VCS) commit and repository location is optional and allows a flexible deployment to update and test different application versions. In the case of a failed deploy, the interface proposes an action to rollback to the previous working application version as follows:

```
POST /rollback
```

4.6.3.2 Request Flow Interface

As shown in Figure 4.2, the worker module M_{WO} pulls requests from a pool of processable requests, has them processed using its designated web application and finally redirects the response so it can be returned to the client. The proposed action sets the URIs for the requests and the responses as follows:

```
PUT /flow
{
  "requests": {
    "componentId": "g28d397sik",
    "uri": "/pool/of/requests",
    "type": "aws.elasticqueue"
  },
  "responses": {
    "componentId": "8f7gliugdy",
    "uri": "/channel/for/responses",
    "type": "aws.eventsystem"
  }
}
```

Both, the requests and the responses objects define the components that are used to retrieve and put data with a URI and a type that identifies the kind of the component. This allows the worker to implement multiple mechanisms for pulling and pushing requests between components.

4.7 Discussion

In this chapter, a novel conceptual architecture design valid for all implementations of WSFs has been presented. With respect to **R4.1**: *What are the design goals for a conceptual architecture design?*, the major identified design goal was to create an architecture that enables to build maintainable, automatable, scalable, resilient, portable and interoperable implementations of WSFs. With respect to **R4.2**: *Which cloud architecture design patterns can be applied?*, the provider adapter pattern, the managed configuration pattern, the elastic manager pattern, the command query responsibility segregation and Flux pattern, the watchdog pattern and the microservice architecture pattern were applied to the conceptual architecture. Each of the applied module-level patterns is dedicated to provide a single required functionality that is implemented by a module of the framework. In consequence, each module is required to implement the identified design goals of a maintainable, automatable, scalabe, resilient, portable and interoperable implementation of a WSF. With respect to **R4.3**: *Which modules are needed to enable automatic scaling?*, the architecture was designed to use a storage module, a metrics module, a watcher module, a resilience module, an actions module, a provision module, an interface module and a worker module in order to separate the concerns of implementation. With respect to **R4.4**: *Which parameters are needed to manage scaling?*, parameters were divided into component parameters, system parameters and traffic parameters to configure and manage scaling. With respect to **R4.5**: *Which minimal set of interfaces needs to be designed and implemented?*, a minimal viable set of interfaces

has been presented that includes component interfaces, framework interfaces and application interfaces.

4.8 Summary

In conclusion, it was shown that the complex matter of web application scaling can be solved by many possible implementations. The presented conceptual architecture is proposed to serve as a framework for all implementations that can share a common understanding of modules, components, parameters and interfaces. The interface definitions in this chapter provide a minimal definition of interfaces that are at least required. For enhanced WSF functionalities, the interfaces need to be extended as presented in Chapter 5 and Chapter 6 of this thesis, where the worker interface is extended to provide access to an optimised resource dependency processing mechanism. Additionally, this chapter does not make any assumptions on the setup and interplay of framework components, nor an optimal routing of requests between the components. Thus, in Chapter 5 an optimised approach to setup components and route requests is presented.

5. Request Flow Optimisation Scheme

5.1 Overview

In this chapter, a novel design pattern for resource storage and management, and an optimised request flow scheme between components is presented. The Permanent Resource Storage and Management (PRSM) pattern enables all resources to be fetched without prior processing, where the processing step is shifted to a management model. The flow scheme presents a novel composition of components, enabling a performance optimised routing of requests. A mathematical model used for performance rating is developed and evaluated on the Raspberry Pi computing cluster *Pi-One* presented in Chapter 2. Traffic traces from over 25 million real-world applications are analysed and evaluated on the cluster to compare the WSF performance with a traditional scaling approach. A resource interface used to declare the existence and dependencies between resources is designed to extend the worker interface presented in Chapter 4. The literature review in Chapter 3 identified six open research questions related to the composition of components and optimised routing of requests between them:

- **R5.1**: Which design pattern can be used for optimised resource management?
- **R5.2**: How can components be composed and requests be flowed efficiently?
- **R5.3**: Which interface extension is required to enable the optimised scheme?
- **R5.4**: How can the performance be modelled analytically?
- **R5.5**: Which metric enables to operate an application with optimal load?
- **R5.6**: How is the real-world application performance of the proposed scheme compared to a traditional approach?

The remainder of the chapter is organised as follows: In Section 5.3, a novel design pattern for optimised resource management is proposed, while in Section 5.4 a concrete scheme for composing components and routing requests is derived. Section 5.5 develops the models for the scheme and Section 5.6 evaluates the models with the computing cluster. Section 5.7 outlines the results and provides answers to the aforementioned research questions. Finally, Section 5.8 concludes the chapter with a perspective on the subsequent chapters.

5.2 Motivations and Objectives

The conceptual architecture presented in the previous Chapter 4 proposes an abstract framework specification with modules, interfaces, parameters and compon-

ents. It however makes no assumptions on the composition of the components that process and serve the requests, nor the sequence of components the requests are flowing through. In order to adapt to the complex application requirements and increasing load generated by mobile devices, a series of architectural styles and design patterns have been identified in Chapter 2, 3 and 4. The microservice architectural style (Fowler, 2014; Namiot and Sneps-Sneppe, 2014) divides complex web application into smaller, manageable units. The CQRS pattern as identified by Young (2010) and Fowler (2011) segregates the domain model of an application into a query model and a command model to decrease complexity. The Facebook Flux (2014) pattern defines a unidirectional data flow where commands always enter the system at a single location and are then dispatched through the system in order to reduce complexity. Consequently, all of the aforementioned patterns help to reduce the complexity of applications. In order to reduce the amount of requests that have to be processed by a web application, a caching layer is introduced as presented in Chapter 2. By caching responses for requests, a response matching a request description can be delivered to the client without prior processing. As the contents of a web application can grow to large amounts of data, not all possible responses are cached. The caching strategies and policies identified in Chapter 2 consider the complex matter of finding the best responses to keep in the cache, so requests must be processed as seldom as possible. When the cache is full, most dispensable responses are evicted from the cache. This cache eviction implies two major issues: Response times are unpredictable and responses must be calculated multiple times although the underlying data has not changed. At the same time, Bocchi, Mellia and Sarni (2014) and Han, Lee, Shin et al. (2012) identify that prices in cloud storage are expected to decrease. This leads to the conclusion, that storage of resources will become cheaper than the recurrent processing and purging when using partial caching. Consequently, this chapter describes the design, implementation and evaluation of the novel PRSM design pattern. It addresses the aforementioned issues by examining the research questions given in the first section of this chapter.

5.3 Permanent Resource Storage and Management Pattern

Resources of web applications are constantly read, created, updated and deleted. With increasing load, the create, update and delete mechanisms however incur the major processing power. This leads to major scaling problems and opens up research question **R5.1**: *Which design pattern can be used for optimised resource management?*

5.3.1 Motivation

In this section, the novel Permanent Resource Storage and Management (PRSM) (read *prism*) pattern is presented. It is based on the three major ideas and assump-

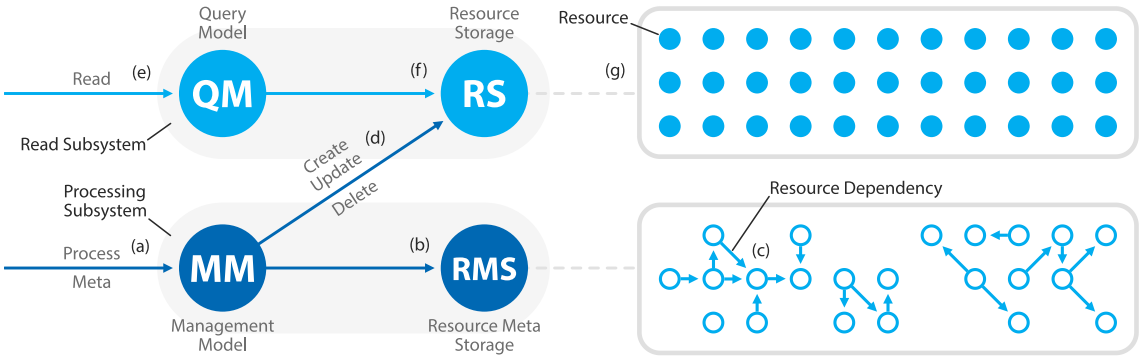


Figure 5.1: Overview of the proposed Permanent Resource Storage and Management (PRSM) pattern.

tions:

1. A read of a resource is more time critical than an update
2. Resources are much more often read than updated or deleted
3. Storage pricing is low and expected to further decrease

The first idea stems from the fact that a user requesting a resource expects a timely response with new information. When a user updates or deletes a resource, the new desired state of the resource is already present in the client. This allows a client to optimistically update the representation the user sees, while waiting for confirmation from the web service. The second assumption depends on the type of an application, but is expected to be true for most web applications. As most traffic traces from applications are kept private, there is only sparse data present to support this assumption. However, all traffic traces examined in the subsequent sections of this chapter present read-heavy traffic characteristics. Additionally, Section 5.5 and 5.6 show up to which degree of read versus processing requests an implementation of the PRSM pattern is viable. The third assumption is primarily identified by Bocchi, Mellia and Sarni (2014) and Han, Lee, Shin et al. (2012) and a further supported by a study of current storage prices of major cloud providers such as Amazon Web Services (2006), Google Cloud Platform (2008), Microsoft Azure (2010) and IBM Bluemix (2014).

5.3.2 Proposed Pattern

Figure 5.1 illustrates the proposed pattern. When a request arrives that needs processing so it can create, update or delete a resource it is first handled by the management model shown in Figure 5.1 (a). The management model checks the resource meta storage at Figure 5.1 (b,c) for related resources. It then performs the updates of all affected resources in the correct order and stores the results in the resource storage at Figure 5.1 (d). The resource meta storage at Figure 5.1 (c) stores resource dependencies that have to be updated when a resource is updated. The management of the resource meta storage is executed by the management model.

When a request arrives that needs to read a resource, it is handled by the query model at Figure 5.1 (e). The query model looks up the current representation of the resource from the resource storage at Figure 5.1 (f,g) and returns it without further processing. Generally, the proposed pattern is an extension to the CQRS pattern where the query model simply returns current representations of resources. The management model implements the CQRS command model that keeps the resources up-to-date in the background. The update mechanism is based on the concept of eventual consistency. A change in the system is not necessarily reflected immediately in a subsequent read, however is guaranteed to be reflected at some future point in time.

5.3.3 Advantages

The proposed pattern exhibits the following advantages over traditional caching approaches:

1. Read response times are constant for all requests
2. Only requests that need processing are sent to the web application
3. Systems can be divided into distinct, individually scalable read and processing subsystems.
4. Changes are processed only exactly once for the whole system

In contrast to the first advantage, with a traditional caching approach it is not known whether a response is cached or not. This leads to unpredictable response times as some responses might need complex and time-consuming reprocessing on the fly. Additionally, it directly influences the user experience with the current total processing performance. When the processing is done in the background as proposed with the PRSM pattern, total processing performance only influences the time changes need to be reflected in the resource storage. With a dedicated resource storage component, the time to request the unchanged representations remains constant. In contrast to the second advantage, traditionally the web application implements a caching mechanism. This means that requests where responses are cached, have a negative influence on the processing performance as they are handled by the same application. With the PRSM pattern, all read requests are persisted by convention so they can directly be responded from the storage and not the application. In contrast to the third advantage, traditionally all requests are handled by the same application so the scaling of individual subsystems is not possible. With the PRSM pattern, both the read and processing capacity can individually be scaled. Additionally, it is possible to migrate or even switch off the processing subsystem while the read subsystem remains unconcerned. In contrast to the fourth advantage, caching mechanisms regularly evict objects from the cache due to storage space constraints. This means that resources that have been processed and cached previously need to

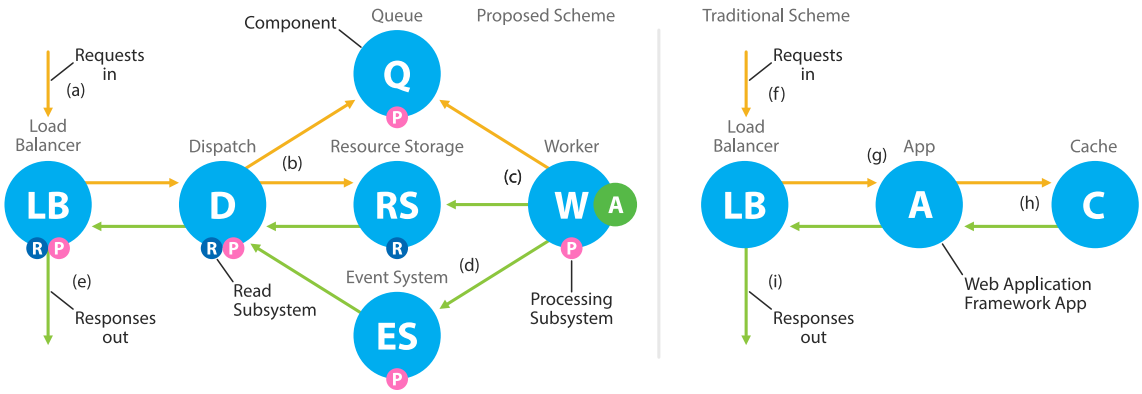


Figure 5.2: Proposed and traditional component composition and request flow scheme.

be reprocessed. During the lifecycle of an application, this can happen frequently where the exact same processing action is repeated each time. With the proposed PRSM pattern, changes are guaranteed to be processed only once as no resource eviction mechanisms are applied. The remainder of this chapter models and evaluates a concrete implementation of the proposed PRSM pattern, where in Chapter 6 mechanisms for an efficient processing subsystem are presented.

5.4 Proposed Scheme Implementation

The proposed PRSM pattern opens up research question **R5.2**: *How can components be composed and requests be flowed efficiently?* Consequently, this section presents a component composition and request flow scheme that implements the PRSM pattern. The proposed scheme is able to calculate and optimise the overall throughput of a web service. In order to compare the performance of the scheme, a traditional composition and flow approach is used. The performance of both approaches is modelled and evaluated in the subsequent Section 5.5 and 5.6.

5.4.1 Traditional and Proposed Scheme Comparison

As reference architecture the traditional scheme S_T is composed from the two-tier pattern Fehling, Leymann, Retter et al. (2014) that was introduced in Section 2.7.3. Figure 5.2 shows the utilised components with a load balancer LB, an application A for the presentation and business logic tier and a cache C for the data tier. The application component A runs an implementation of a web application that is created using a WAF. The processing of requests follows the graph with the edges $S_T = \{(LB, A), (A, C), (C, A), (A, LB)\}$. The proposed scheme S_T presented in Figure 5.2 uses more components and a different composition than the traditional scheme S_T. It implements the PRSM pattern to combine a WSF with a WAF. Figure 5.2 (a-e) illustrates the components the proposed scheme adds to the traditional scheme S_T: A request queue Q, a dispatcher D, an event system component ES and a worker component W that embeds an implementation of a web application A.

5.4.2 Request Flow

The illustrated composition of components (Figure 5.2 (a-e)) is created with two major design goals: optimised performance and enhanced scalability. The approach to optimising the performance is to minimise the request flow graph for every request by implementing the PRSM pattern. Figure 5.2 (a-e) illustrates the detailed flow of a request through the components: Requests enter the system through multiple load balancers LB (Figure 5.2 (a)). The load balancers LB forward the request to one of the dispatchers D. The dispatcher D now decides whether the request is a read request R_R or a processing request R_P. Read requests R_R are determined by the HTTP methods that guarantee to be safe as described in Section 2.2.1 (GET, HEAD and OPTIONS do nothing but retrieve content without side effects). Processing requests R_P are requests with all other HTTP methods that by definition change content on the server. Read requests R_R are routed to the read subsystem SR which is a graph with the directed edges $SR = \{(LB, D),\ (D, RS),\ (RS, D),\ (D, LB)\}$ (Figure 5.2 (a,b). Processing requests R_P are routed to the processing subsystem SP where $SP = \{(LB, D),\ (D, Q),\ (Q, W),\ (W, A),\ (A, W),\ (W, ES),\ (ES, D),\ (D, LB)\}$ (Figure 5.2 (a-e)). This is only possible as by definition all deliverable resources are stored in the resource storage RS. Therefore, for a read request R_R the dispatcher D looks up the storage key including possible fragments and delivers the response. Processing requests R_P always need to be processed, so the dispatcher D puts them in the queue Q and listens for the response from the event system ES. At Figure 5.2 (c), a worker W with free resources eventually pops the request from the queue Q. The worker W has the request processed by the app A and then publishes the response to the event system ES (Figure 5.2 (d)) where the dispatcher D is waiting for it. Finally at Figure 5.2 (e), the dispatcher D delivers the response back to the client (Figure 5.2 (h)). With this implementation, both subsystems can be scaled independently on a component level.

5.4.3 Resource and Dependency Processing Scheme

In order to use the proposed scheme, all deliverable resources need to initially be put in the resource storage RS. This requires the creation of an index of all available resources before the system can go into operating state. In Chapter 7, this initial storage fill is elaborated further. To maintain an updated resource state, a management model as proposed by the PRSM pattern is implemented.

5.4.3.1 Synchronous and Asynchronous Processing Phase

Section 5.3 proposes that the management model works in the background and thereby only eventual consistency can be guaranteed. However, for a web service action that processes a resource and afterwards reads this resource it is not guaranteed that the resource is already processed and updated in the storage. Depending

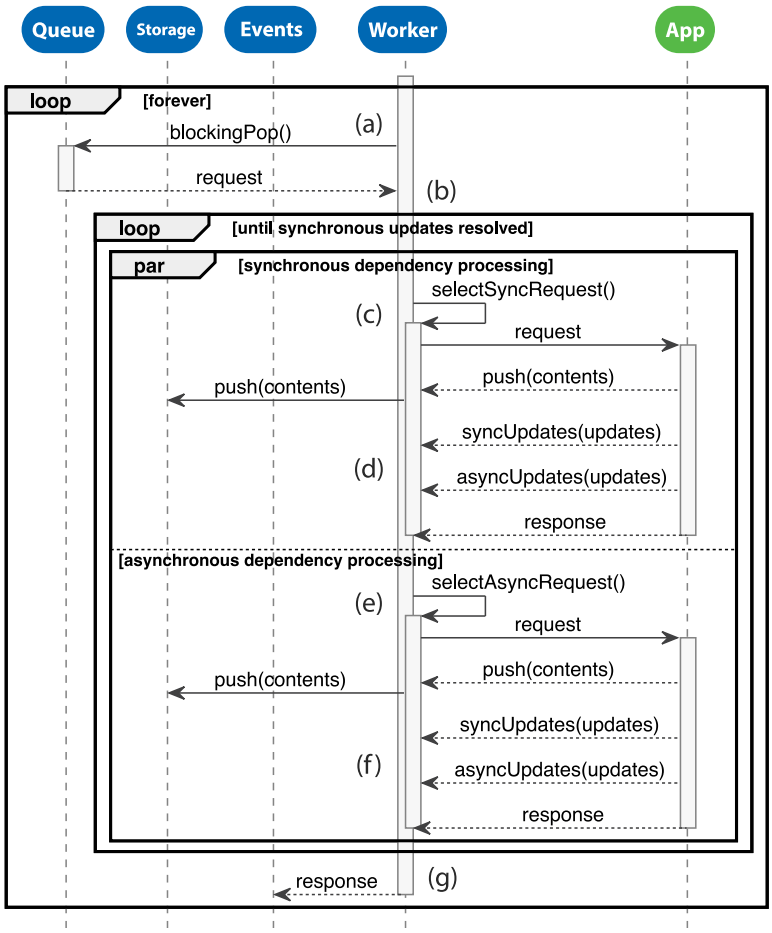


Figure 5.3: Sequence diagram of the resource and dependency processing mechanism implementing the management model of the PRSM pattern.

on the current load of the management model, the change can take more or less time to process. To address this, the proposed scheme defines a synchronous and an asynchronous phase in dependency processing. In the synchronous phase, the worker ensures to stall the response until all dependencies are processed. The asynchronous phase runs in parallel and has no effect on the response time. With this approach, actions that require strong consistency because they subsequently read an updated response can define synchronous dependencies. Resources that must be updated but do not need to be finished by the time of the response can be defined as asynchronous dependencies.

5.4.3.2 Processing Scheme

Figure 5.3 presents a sequence diagram of the processing scheme. The dispatcher D puts requests that need processing into the queue Q. If a worker has available resource to process a request, it pops a request from the queue at Figure 5.3 (a) and processes it synchronously at Figure 5.3 (b). In the parallel processing section Figure 5.3 (c-f), the worker W selects a request that needs processing by the app A (Figure 5.3 (c,e)) and initiates the request. The app A then processes the request. During processing, it pushes updated contents to the storage. Additionally, it de-

clares its synchronous and asynchronous updates at Figure 5.3 (d,f) before it returns the response to the worker. This is repeated until all synchronous updates are processed. Finally, the response to the initial request is put into the event system at Figure 5.3 (g). With this approach, the asynchronous dependencies are decoupled from the original request and processed when the system has available resources. A general goal to minimise the response time for a request is to keep the number of synchronous updates as low as possible. This allows the processing of dependencies to happen in the background, while the response to the initial request can be sent out immediately.

5.4.4 Resource Interface

In the conceptual architecture design at Section 4.6, a set of minimal viable interfaces has been designed. The interfaces however do not specify how resources and their dependencies can be registered or stored which opens up research question **R5.3**: *Which interface extension is required to enable the optimised scheme?* Consequently, this section provides an interface extension to the proposed worker interface that allows to manage the resources. The resource interface is essential to the composition and request flow scheme proposed in this chapter, however not required for other implementations of WSFs. The proposed resource interface is divided into a storage interface and a meta interface that are presented in the subsequent sections.

5.4.4.1 Storage Interface

As shown in Figure 5.3, an application needs to be able to put resource into the storage via the worker. Hence, the storage interface proposes an action as follows:

```
PUT/DELETE /store/:resourceId
// resource content line 1
// resource content line 2
// resource content line 3
// ...
```

The resource id field can be of any form, e.g. an URI. The body of the request simply contains the contents to store or update. If a resource does not exist, it needs to be created, if it exists it needs to be updated. On delete, no request body needs to be specified.

5.4.4.2 Meta Interface

Based on the PRSM pattern, the worker stores the meta information about resources and their dependencies in order to manage the changes correctly. Hence, the dependency interface proposes an action as follows:

```
PUT/DELETE /meta/dependency/:resourceId
```

```
[
  { "resourceId": "/sitemap", "type": "async" },
  { "resourceId": "/posts", "type": "async" },
  { "resourceId": "/posts", "type": "sync" },
  ...
]
```

The array in the body defines the direct dependencies of the resource id with their type. From this, the worker defines a dependency graph which is examined in detail in Chapter 6. Additionally, the application can manually define dependencies that are considered only for the active request. For this, the interface action is proposed as follows:

```
POST /meta/update/:resourceId
[
  { "resourceId": "/sitemap", "type": "async" },
  { "resourceId": "/posts", "type": "async" },
  { "resourceId": "/posts", "type": "sync" },
  ...
]
```

The worker merges these dependencies into the dependency graph and processes the requests according to Figure 5.3.

5.5 Analytical Performance Modelling

A model that allows to calculate performance based on certain parameters enables to analyse the differences between a traditional scheme and the scheme proposed in this chapter. This opens up research question **R5.4**: *How can the performance be modelled analytically?* Consequently, in this section mathematical performance models for a component, the composition of multiple components, the performance comparison of two compositions of components, and the performance optimisation of a component are developed.

5.5.1 Performance Goals

The general performance goals for the proposed scheme are either the reduction of the number of total machines M needed to satisfy a desired target request flow F or in reverse the increase of the request flow F with a given number of total machines M. Thus, models for the traditional scheme S_T and the proposed scheme S_P are developed. Generally, it is aimed to achieve that $F_P > F_T$ with equal M or $M_P < M_T$ with equal F.

Table 5.1: Component parameters that are used to describe and model the performance of a single component x.

Goal oriented	**Variable**	**Description**
Request Flow/Second	$f_x \in [0, \infty]$	The requests that flow through a component in one second.
Target Flow/Second	$t_x \in [0, \infty]$	A desired target f_x for a component.
Performance based	**Variable**	**Description**
Network Delay	$d_{n,x} \in [0, \infty]$ in s	The time it takes a request to travel the full network stack.
Network Delay Gain	$d_{g,x} \in [0, \infty]$ in s	The linear or quadratic time factor by which $d_{n,x}$ increases.
Lookup Delay (S_T)	$d_{l,x} \in [0, \infty]$ in s	The time it takes the app A to lookup a resource in the resource storage RS.
Processing Delay	$d_{p,x} \in [0, \infty]$ in s	The time it takes a component to process a request.
Dependency Processing Delay	$d_{dp,x} \in [0, \infty]$	The time needed to process dependencies a request.
Size Delay	$d_{s,x} \in [0, \infty]$ in s	The time a single kilobyte adds to the delay.
Workload based	**Variable**	**Description**
Concurrent Users	$c_x \in [1, \infty]$	The number of concurrent users or connections.
Size	$s_x \in [0, \infty]$ in kB	The size of a single request-response round-trip.
Deployment based	**Variable**	**Description**
Machines	$m_x \in [1, \infty]$	The number of machines that are used for one component.

5.5.2 Component Models

As a first step, the performance of a single component is modelled that later is composed to a larger system.

5.5.2.1 Parameters

For the model, each component has its own set of parameters denoted in Table 5.1. Component parameters are not valid for the whole composition as they are influenced by the individual routing and processing of a component. By convention, lower case variables are used whenever a parameter or model belongs to a component. In a composition, a component is identified by the subscript x which serves a placeholder for a component abbreviation such as LB, D or W.

5.5.2.2 Delay Factors

A request flowing through a component x is delayed by different factors. The model takes this into account by calculating the processing-, request size- and network

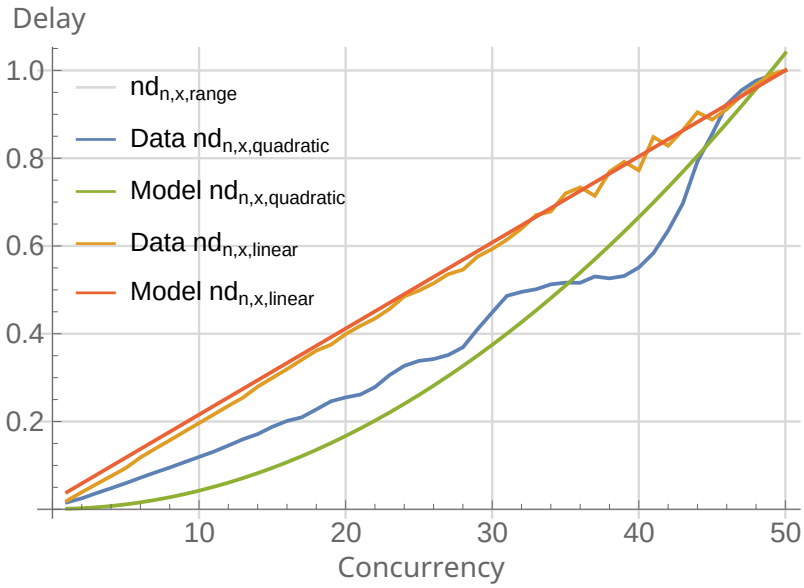


Figure 5.4: Normalised measurements and model of the linear and quadratic network delay.

delays:

$$d_{P,x} = d_{p,x} + d_{dp,x} \tag{5.1}$$

$$d_{S,x} = d_{s,x} \cdot s_x \tag{5.2}$$

For the network delay two different developments are observed from measured data that is illustrated in Figure 5.4:

$$d_{N,x} = \begin{cases} \dfrac{c_x \cdot d_{g,x}}{m_x} + d_{n,x}, & \text{if } \textit{linear} \quad (5.3a) \\ \dfrac{c_x^2 \cdot d_{g,x} + d_{n,x}}{m_x}, & \text{if } \textit{quadratic} \quad (5.3b) \end{cases}$$

For the measurements, requests with increasing concurrency are issued to both a resource storage component RS and an application component A. For the resource storage component a simple ping command is sent, where the resource storage responds with a pong command. For the application component a simple empty request is sent, where the application replies with an empty response. This is done to eliminate any processing and size related delays. The results show that for the resource storage component RS a linear development of the delays can be observed (Figure 5.4), while for the application component A a quadratic development of the delays can be observed. This stems from the increased complexity of the application component A, while the resource storage component RS provides a constant lookup time through distributed hashing. The data lines in Figure 5.4 are averages from 20 test runs for both the linear and the quadratic development. Despite the increased complexity, both models are developed and allow the user to select the appropriate accuracy. If simplicity is chosen over accuracy, the linear version can be used only.

Otherwise, the delay model is selected by the model fit.

5.5.2.3 Maximum Request Flow

To calculate the requests that can flow through a component per second, the concurrency is divided by the sum of all component delay factors. Adding more machines to the component increases the flow f_x by m_x to a maximum:

$$f_x = \frac{c_x \cdot m_x}{d_{P,x} + d_{S,x} + d_{N,x}} \tag{5.4}$$

As the performance improvement is not linear with m_x, $d_{N,x}$ increases with m_x and thereby degrades the performance.

5.5.2.4 Machines for Target Flow

To satisfy a target flow $f_x = t_x$, the number of machines a component uses needs to be adapted. It can be calculated by solving f_x from (5.4) for m_x in the linear case l and the quadratic case q, where the processing, size and network delays are substituted with their corresponding full representations from (5.1), (5.2), (5.3a) and (5.3b). Please note, that a and b represent substitutions for display purpose only:

$$\begin{aligned}
a_l &= d_{p,x}t_x + d_{dp,x}t_x + d_{s,x}s_x t_x \\
b_l &= \sqrt{t_x(4c_x^3 d_{g,x} + 4c_x d_{n,x} + (d_{p,x} + d_{dp,x} + d_{s,x}s_x)^2 t_x)} \\
a_q &= d_{n,x}t_x + d_{p,x}t_x + d_{dp,x}t_x + d_{s,x}s_x t_x \\
b_q &= \sqrt{t_x(4c_x^2 d_{g,x} + (d_{n,x} + d_{p,x} + d_{dp,x} + d_{s,x}s_x)^2 t_x)} \\
m_{t,x} &= \left\lceil \frac{1}{2c_x}(a_l + b_l) \right\rceil, \qquad \text{or} \qquad \left\lceil \frac{1}{2c_x}(a_q + b_q) \right\rceil
\end{aligned} \tag{5.5}$$

For the traditional scheme $S_T : d_{dp,x} = 0$, as the traditional scheme does not apply any dependency processing.

5.5.3 Composition Models

The composition models compose the individual components into a larger system.

5.5.3.1 Parameters

The parameters denoted in Table 5.2 are valid for a whole composed system of components. By convention, capital notation is used whenever a parameter or model belongs to the whole composition.

5.5.3.2 Components and Subsystems

Two compositions of components are proposed: The traditional scheme S_T and the proposed scheme S_P. The traditional scheme S_T uses the components $C_T =$

Table 5.2: Composition parameters that are used to describe and model the performance of the composition of multiple components.

Goal oriented	**Variable**	**Description**
Request Flow/Second	$F \in [0, \infty]$	The requests that flow through all components in one second.
Target Flow/Second	$T \in [0, \infty]$	A desired target F for the whole system.
Workload based	**Variable**	**Description**
Read/Processing Ratio	$RPR \in [1, 0]$	The relation between read requests R_R and processing requests R_P.
Cache Hit/Miss Ratio (S_T)	$HMR \in [0, 1]$	The relation between cache hits and cache misses.
Deployment based	**Variable**	**Description**
Machines	$M \in [1, \infty]$	The number of total machines that are used for the whole system.
Machine-Quantity Tuple	$MQT = (m_x \mid x \in C_X)$	Lists number of machines for each component in a composition X.

(LB, A, C). The proposed scheme S_P uses the components $C_P = (LB, D, RS, Q, W, ES)$. For the model the components need to be separated by read subsystem SR and processing subsystem SP as illustrated in Figure 5.2:

$$C_{T_{SR,SP}} = (LB, A) \tag{5.6}$$
$$C_{T_{SR}} = (C) \tag{5.7}$$
$$C_{P_{SR,SP}} = (LB, D) \tag{5.8}$$
$$C_{P_{SR}} = (RS) \tag{5.9}$$
$$C_{P_{SP}} = (Q, W, ES) \tag{5.10}$$

5.5.3.3 Maximum Request Flow

The maximum request flow F predicts the maximal throughput of S_T or S_P for a machine-quantity tuple MQT. In the traditional scheme S_T, requests can be either served (cache hit) or need to be processed (cache miss) depending on the cache C's cache hit/miss ratio HMR:

$$d_{p,A} = (RPR \cdot d_{l,A}) + (d_{p,A} - RPR \cdot HMR \cdot d_{p,A}) \tag{5.11}$$

The concurrency at the cache C depends on the number of machines and concurrency of the application A and the read/processing ratio RPR:

$$c_C = m_A \cdot c_A \cdot RPR \tag{5.12}$$

The maximum flow is determined by the slowest component of a composition:

$$F_T = \min\{f_x \mid x \in C_{T_{SR,SP}}, \frac{f_x}{RPR} \mid x \in C_{T_{SR}}\} \tag{5.13}$$

For the proposed scheme S_P, both the read subsystem SR and the processing subsystem SP have to be considered:

$$F_P = \min\{f_x \mid x \in C_{P_{SR,SP}}, \frac{f_x}{RPR} \mid x \in C_{P_{SR}}, -\frac{f_x}{-1+RPR} \mid x \in C_{P_{SP}}\} \tag{5.14}$$

5.5.3.4 Machines for Target Flow

The machines equation calculates the machine-quantity tuple MQT for a certain target flow $F = T$ in a composition. For S_T, the app delay $d_{p,A}$ is used as shown in (5.11) and the cache C is only hit by read requests R_R:

$$t_C = RPR \cdot T \tag{5.15}$$

The quantities are calculated for every component:

$$M_{T,T,MQT} = (m_x \mid x \in C_T) \tag{5.16}$$
$$= (m_{LB}, m_A, m_C) \tag{5.17}$$
$$M_{T,T} = \sum_{m_x, x \in C_T} m_x \tag{5.18}$$

The tuple (5.17) is turned to a scalar by summing its individual components (5.18). The sum $M_{T,T}$ is the total number of machines needed for target T. For instance, if the load-balancer uses three machines, the application six machines and the cache a single machine, the tuple looks like this:

$$M_{T,T,MQT} = (3, 6, 1) \tag{5.19}$$
$$M_{T,T} = 10 \tag{5.20}$$

The number of total machines for the target T then is the sum of all component machines, in this example 10. For the proposed scheme S_P, the target T is split up by the processing subsystem SP and the read subsystem SR:

$$t_x = \begin{cases} T, & \text{if } x \in C_{SR,SP} \quad (5.21a) \\ RPR \cdot T, & \text{if } x \in C_{SR} \quad (5.21b) \\ (1 - RPR) \cdot T, & \text{if } x \in C_{SP} \quad (5.21c) \end{cases}$$

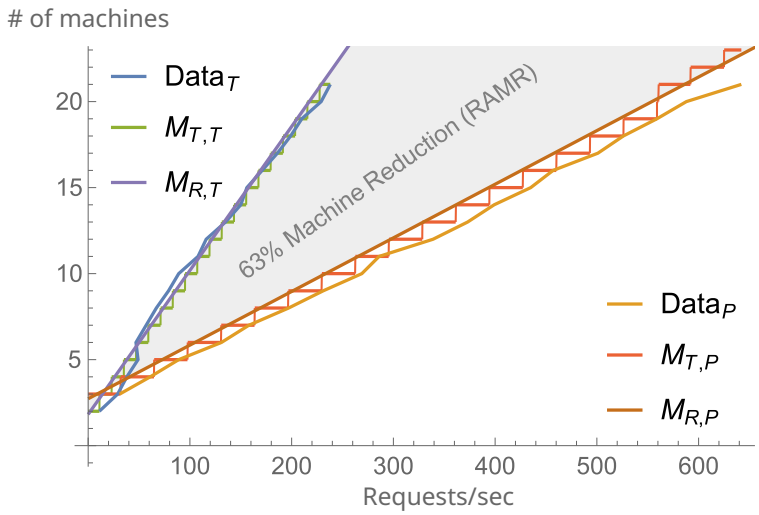


Figure 5.5: A comparison of the total machines for target model M_T, the linear total machines regression M_R and measured data for the proposed scheme S_P and traditional scheme S_T.

The machine-quantity tuple MQT is generated from all components:

$$M_{T,S,MQT} = (m_x \mid x \in C_P) \tag{5.22}$$

$$= (m_{LB}, m_D, m_{RS}, m_Q, m_W, m_{ES}) \tag{5.23}$$

$$M_{T,S} = \sum_{m_x, x \in C_P} m_x \tag{5.24}$$

As for the traditional scheme S_T, the tuple (5.23) is turned to a scalar by summing its individual components (5.24).

5.5.3.5 Linear Regression for Machines for Target Flow

The linear regression for the number of machines provides a simpler approximation of the performance (Figure 5.5). The machine reduction in Figure 5.5 stems from a finer-grained allocation of machines, where in the traditional scheme the full application has to be scaled, where in the proposed scheme both the read and processing subsystem can be scaled individually. For the traditional scheme S_T, the app delay $d_{p,A}$ is as shown in (5.11). The slope can be calculated as a unit of increasing total machines per flow-quantity:

$$M_{R,T,s} = \sum_{x \in C_{T_{SR,SP}}} \frac{1}{f_x} + \sum_{x \in C_{T_{SR}}} \frac{RPR}{f_x} \tag{5.25}$$

The full regression equation multiplies the slope with the target T and adds the minimal number of component machines $|C_T|$:

$$M_{R,N} = T \cdot M_{R,N,s} + |C_T| \tag{5.26}$$

For the proposed scheme S_P, the slope needs to consider both the read subsystem SR and the processing subsystem SP:

$$M_{R,P,s} = \sum_{x \in C_{P_{SR,SP}}} \frac{1}{f_x} + \sum_{x \in C_{P_{SR}}} \frac{1}{RPR \cdot f_x} + \sum_{x \in C_{P_{SP}}} \frac{1 - RPR}{f_x} \tag{5.27}$$

$$M_{R,P} = T \cdot M_{R,P,s} + |C_P| \tag{5.28}$$

5.5.4 Performance Comparison

To be able to compare the performance of the traditional scheme S_T and the proposed scheme S_P, comparison metrics are developed.

5.5.4.1 Relative Average Machine Reduction

When the proposed scheme S_P needs fewer machines for the same load than the traditional scheme S_T, the delta can be expressed as a factor of machine reduction. The relative average machine reduction is calculated with the slopes of the linear total machines regressions of both schemes (Figure 5.5).

$$RAMR = 1 - \frac{M_{R,P,s}}{M_{R,T,s}} \tag{5.29}$$

If the proposed scheme S_P needs five machines and the traditional scheme S_T needs six machines for the same load, the RAMR equals $(1 - (5/6)) = 0.17$. This shows, that the proposed scheme S_P needs 17% fewer machines than the traditional scheme S_T. If both schemes use the same number of machines, the $RAMR$ is zero.

5.5.4.2 Break-Even Point for Dependency Processing

In the traditional scheme S_T, the app A is responsible for the cache updates and invalidation. Therefore, the cost of updates is added to the app delay $d_{p,A}$. In the proposed scheme S_P, the dependency processing delay $d_{dp,W}$ is explicitly defined as the time it takes to process the dependencies of a request. When comparing both schemes, an interesting metric is the time the proposed scheme S_P has available for the dependency processing $d_{dp,W}$. The dependency processing delay $d_{dp,W}$ where both schemes deliver the same performance is the break-even point $d_{dp,W,BEP}$. It can be calculated by equalising the linear regressions of the traditional scheme S_T and proposed scheme S_P and solving the equation for the dependency processing delay $d_{dp,W}$:

$$C_{P_{SP}} = C_{P_{SP}} \setminus \{W\} \tag{5.30}$$

$$\begin{aligned} d_{dp,W,BEP} &= (M_{R,T} = M_{R,S}), \quad \text{solve for } d_{dp,W} \\ &= c_w \cdot m_W \cdot (M_{R,T} - M_{R,P}) - (1 - RPR) \cdot (d_{T,W} + d_{P,W} + d_{p,W}) \end{aligned} \tag{5.31}$$

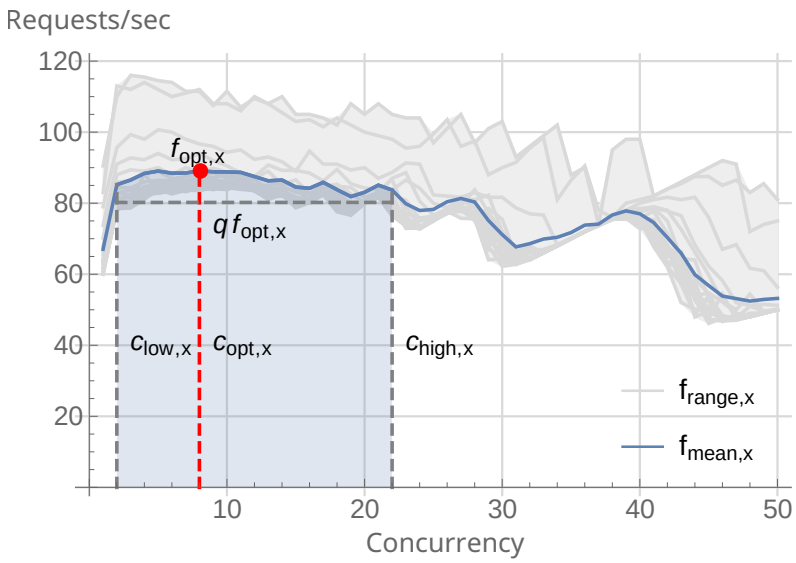


Figure 5.6: Optimal Concurrency Range $ocw_{0.9,x} = [2, 22]$ with a performance-concurrency-width triplet $pcw_{0.9,x} = (89, 8, 20)$ for the average normalised request flow f_x of 20 machines.

For the break-even calculation, the worker component W is excluded from the processing components $C_{P_{SP}}$ as shown in (5.30). The worker component delays without the dependency processing delay are specifically considered in the last term of (5.31).

5.5.5 Performance Optimisation

In Section 2.8 it was identified that the hardware and implementation of a component is critical to the performance. This opens up research question **R5.5**: *Which metric enables to operate an application with optimal load?* Consequently, this section considers optimal load and implementation specific metrics.

5.5.5.1 Optimal Concurrency Range

In Section 2.8 and further machine-normalised measurements for 1 to 22 machines as $f_{range,x}$ (Figure 5.6) it is observed that the request flow of a component has an optimal range. In Figure 5.6, $f_{mean,x}$ shows that with increasing concurrency c_x, a component has a performance optimum $c_{opt,x}$ where it delivers the maximal request flow $f_{opt,x}$. An algorithm that controls the flow of requests to a component aims to load the component with the optimal concurrency $c_{opt,x}$. A key metric for a component, however, is the sensitivity around the optimal concurrency. If the performance degradation is low around the optimum, algorithms can operate with higher tolerance. This allows the systems to be more insensitive to dynamic concurrency values, which leads to fewer scaling actions. To describe this broadness of possible concurrency values with respect to a percentage request flow loss q, the optimal concurrency range $ocr_{q,x}$ is introduced. An $ocr_{0.95,x} = [5, 20]$ means that a component is able to handle concurrencies between 5 and 20 while operating at 95% of the performance optimum $f_{opt,x}$. The optimal concurrency range $ocr_{q,x}$ can be calculated

from a series of performance $data(c_x)$ that is measured with increasing concurrency values c_x. The following is representative pseudocode for finding the lowest- and highest concurrency values:

1: $f_{opt,x} \leftarrow \max(data)$
2: $c_{low,x} \leftarrow c_{high,x} \leftarrow c_{opt,x} \leftarrow \max^{-1}(f_{opt,x})$
3: **while** data$(c_{low,x}) \geq q \cdot f_{opt,x}$ **do**
4: $\quad c_{low,x} \leftarrow c_{low,x} - 1$
5: **end while**
6: **while** data$(c_{high,x}) \geq q \cdot f_{opt,x}$ **do**
7: $\quad c_{high,x} \leftarrow c_{high,x} + 1$
8: **end while**
9: **return** $[c_{low,x}, c_{high,x}]$

The optimal concurrency width $ocw_{q,x}$ of an optimal concurrency range $ocr_{q,x}$ expresses the general sensitivity of the component to concurrency:

$$ocw_{q,x} := \max(ocr_{q,x}) - \min(ocr_{q,x}) \tag{5.32}$$

5.5.5.2 Performance-Concurrency-Width Triplet

In order to optimise the performance, the Performance-Concurrency-Width triplet $pcw_{q,x}$ is proposed as a metric that allows comparing different implementations of components with each other.

$$pcw_{q,x} = (f_{opt,x}, c_{opt,x}, ocw_{q,x}) \tag{5.33}$$

The $pcw_{q,x}$-triplet includes the most important performance parameters for a component. It allows building a performance delta triplet $\Delta pcw_{q,x} = pcw_{q,Z} - pcw_{q,Y}$ that shows the performance differences between implementations Y and Z. A delta triplet $\Delta pcw_{q,x} = (12, 0, 8)$ presents Z as a superior implementation to Y. With the same concurrency, implementation Z's request flow f is 12 requests bigger. At the same time it is 8 concurrency values more insensitive to load.

5.6 Empirical Performance Evaluation

After the modelling of the performance in the previous section, this section empirically evaluates both schemes with multiple machines on the component and composition level. For the evaluation, the evaluation cluster *Pi-One* that was introduced in Section 2.8 is used. Performance testing of systems with real-world application data further strengthens modelled assumptions and improves trust in the model. This opens up research question **R5.6**: *How is the real-world application performance of the proposed scheme compared to a traditional approach?* Consequently, this section additionally compares the performance of three real-world applications for

Table 5.3: Isolated evaluation data for the delay factors of the component models.

Delay	Parameter	Value	R^2	RMSE	Fit
Network	$d_{n,x}$	8.32×10^{-4}	0.975	1.7×10^{-2}	0.973
	$d_{g,x}$	3.96×10^{-4}			
Size	$d_{s,x}$	1.26×10^{-4}	0.998	1.3×10^{-3}	0.975
Process	$d_{p,x}$	2.07×10^{-10}	—	—	—

both schemes.

5.6.1 Component Models Evaluation

To evaluate the component models, a single component with multiple machines is tested. The central equation, the maximum request flow f_x of a component is composed of the number of machines m_x, the network delay $d_{n,x}$, the request size delay $d_{s,x}$ and the process delay $d_{p,x}$. Since the evaluation of the whole equation is complicated, the delay factors are isolated and evaluated individually. For each delay factor, a customised test bed is set up that sends requests to the component. Each test bed eliminates the influence of other delays, such as different sizes and processing delays by setting them to zero. Further, each test bed varies the number of machines and applied concurrency to compare the predicted delays from the models with the measured delays. For the network delay evaluation the goal is to measure the network delay $d_{n,x}$ and determine the network delay gain $d_{g,x}$ by fitting it to the results of the evaluation. For the size delay $d_{s,x}$ evaluation the goal is to measure the influence a single kilobyte has on the delay by finding a slope that matches all measured values best. The processing delay $d_{p,x}$ can be measured and depends on the type of processing that occurs inside the component. The results from the evaluations including determined values valid for the *Pi-One* evaluation platform are given in Table 5.3.

5.6.1.1 Metrics

To quantify the results of the evaluation and compare the model to the data, the Coefficient of Determination (R^2), Root-Mean-Square Error (RMSE), Normalised RMSE (NRMSE) and model fit are calculated. The RMSE shows the absolute error without relating it to the range of observed values. The normalised version of the RMSE relates to the observed values so that $NRMSE = RMSE/(y_{\max} - y\min)$ where $y_{\max}$ and $y_{\min}$ represent the maximum and minimum of all observed values y. This allows expressing the model fit $Fit = 1 - NRMSE$ as a percentage where 1.0 is a perfect fit and 0.0 is no fit.

5.6.1.2 Network Delay

In the first step, the network delay $d_{n,x}$ is isolated by setting $d_{s,x} = d_{p,x} = 0$. For the model, the known parameters are the number of machines m_x and the

concurrency c_x. The tests are run for all possible (m_x, c_x) combinations where the number of machines m_x is in the range of $(1 \dots 20)$, and the number of concurrent requests c_x is in the range of $(1 \dots 50)$. The network delay $d_{n,x}$ can be retrieved from the results as the smallest measured delay. The network delay gain $d_{g,x}$ can either be formulated as a quadratic- or linear optimisation problem on the network delay $d_{n,x}$. It is solved using the *NonlinearModelFit* function from Mathematica (1988) which automatically picks a best fitting linear or quadratic delay using one of the *ConjugateGradient, Gradient, LevenbergMarquardt, Newton, NMinimize* or *QuasiNewton* regression methods. The results support the network delay model $d_{n,x}$ as it fits the data by 97.3%.

5.6.1.3 Request Size Delay

The size delay $d_{s,x}$ is isolated by setting the processing delay $d_{p,x} = 0$ and the concurrency and number of machines $c_x = m_x = 1$. According to the Internet Archive (1996), the average individual response size depends on the content type but is smaller than 108 kB. Tested sizes s range from 0 to 400 to cover the average response sizes listed in Internet Archive (1996) well. The determination of the size delay $d_{s,x}$ can be formulated as a linear optimization problem with $d_{s,x}$ and is computed using Mathematica (1988). The results in Table 5.3 support the size delay model $d_{s,x}$ as it fits the data by 97.5%.

5.6.1.4 Processing Delay

The processing delay $d_{p,x}$ can simply be measured where our results are listed in Table 5.3. For example a component that guarantees a constant lookup time $O(1)$ is expected to have a constant processing delay $d_{p,x}$.

5.6.2 Composition Models Evaluation

As a next step, the composition evaluation is used to analyse the interplay of components. For the evaluation, a chained and distributed composition of components is conceived. The proposed scheme model makes the following assumptions that can be formulated as hypotheses:

- **H5.1**: The maximum request flow is determined by the slowest component in the chain (chained composition).
- **H5.2**: The maximum request flow is relative to the distribution of the traffic to the components (distributed composition).

5.6.2.1 Chained Composition

In a chained composition multiple components are stringed through a single connection. The evaluation is run with $(2 \dots 10)$ machines stringed together. One random machine in the chain introduces a processing delay of $d_{p,x} = 0.05$. Measured at the

end of the chain, the expected maximum delay is:

$$F = (d_{n,x} + d_{p,x})^{-1} = 19 \tag{5.34}$$

The measurements of all chains show a request flow F of 19 which supports **H5.1** with empirical data.

5.6.2.2 Distributed Composition

In the distributed composition one component distributes the requests to many others. One component X dispatches the traffic to two other components (Y, Z) with 10 different ratios $r_{disp} \in (0.0, 0.1, \ldots, 1.0)$. X introduces a processing delay of $d_{p,X} = 0.01$ and Y a processing delay of $d_{p,Y} = 0.1$. The expected maximum for each ratio is:

$$F = ((r_{disp} \cdot d_{p,X}) + ((1 - r_{disp}) \cdot d_{p,Y}))^{-1} \tag{5.35}$$

All test cases combined have a total $RMSE = 4.11$ and a total prediction fit of $Fit = 0.942$. This allows to support **H5.2** as the data supports the model with confidence of 94.2%.

5.6.3 Real-World Application Evaluation

After the evaluation of components and their composition, the whole model is evaluated. Therefore it is interesting to see how the model behaves when traffic with the parameters of three real-world application traces is applied. For each application, both the traditional scheme S_T and the proposed scheme S_P are implemented. The collected data series of each application in both schemes is evaluated in two dimensions: The prediction fits of the proposed model and the machine reductions. The data series measure the achieved request flow F for the total number of machines $M_{T,N}$. The measurements are starting with the minimum number of machines as determined by the components for the traditional scheme $|C_T|$ and the proposed schemed $|C_P|$. Further machines are added up to a maximum of 20, as half of the 42 machines are needed for load generation. As a scaling decision, the new machine of the next run is always added to the slowest component that introduces the bottleneck. The evaluation considers three application traces that are presented in the subsequent sections.

5.6.3.1 Trip Planner

The trip planner T_{trip} is a web service that allows users to plan a journey all over the world. It calculates the itinerary between two or more destinations and enhances it with local information, e.g. restaurants and hotels. The service has no social features that allow the sharing of trips or recommendations. The traffic resembles

Table 5.4: Trace parameters extracted from real-world applications.

Application	n	RPR	$d_{p,x}$	s
trip	10 million	0.849	1.41 s	2 kB
social	0.6 million	0.571	0.48 s	161 kB
soccer	14 million	0.998	1.0 s	5 kB

an application of an intermediate update nature as trip indices can be calculated offline, but user input has to be handled. The $|T_{trip}|$ trace contains 10 million traces that are available from webscalingframeworks.org/traces (2016).

5.6.3.2 Social Network

The social network T_{social} is a platform we implemented and set up on campus as traffic traces of social-networks were unavailable. The platform provides a subset of the features of a typical social-network platform and is built to be as similar to Facebook as possible. In addition to the management of people and their friendships, it shows a news feed with status messages from friends and allows exchanging private messages between friends. Missing features are photo- and video-sharing and the creation and management of groups. The platform is hosted on a university server and only accessible from the university test network. Over a time period of two months we recorded user requests and replayed them ten times against the platform. The traffic resembles traffic of a social nature as resources are constantly changed by users. The $|T_{social}|$ trace contains 0.6 million traces that are available from webscalingframeworks.org/traces (2016).

5.6.3.3 FIFA Soccer Worldcup 98 Website

Traces of the 1998 soccer World Cup website T_{soccer} between April 30, 1998 and July 26, 1998. The website resembles an application of a more static nature as no social features and few processing requests are issued. The $|T_{soccer}|$ trace contains 14 million traces that are available online from Hewlett-Packard (1999).

5.6.3.4 Extracted Application Metrics

Table 5.4 lists the traffic trace parameters extracted from the evaluated T_{trip}, T_{social} and T_{soccer} applications. The n parameter shows the total number of analysed traces. For the read/processing ratio RPR, the number of safe versus unsafe HTTP methods is counted. The mean processing delay $d_{p,x}$ is extracted by calculating the time difference between request and response. The request size for each trace is given where the evaluation uses the mean of all sizes. As the T_{soccer} trace does not contain processing delays $d_{p,x}$, it is manually set to a rounded average processing delay $d_{p,x} = 1$ derived from the TP_{trip} and TP_{social} traces.

Table 5.5: Results of real-world application evaluation for both schemes.

	Prediction Fit				Machine Reduction			DP BEP
App	$RMSE_T$	$RMSE_P$	Fit_T	Fit_P	OMR	PMR	$RAMR$	$d_{dp,x}$
trip	0.707	0.858	0.962	0.954	0.630	0.604	0.627	2.69
social	0.806	0.858	0.957	0.954	0.325	0.355	0.466	1.19
soccer	1.048	2.790	0.944	0.853	0.924	0.943	0.903	26.04

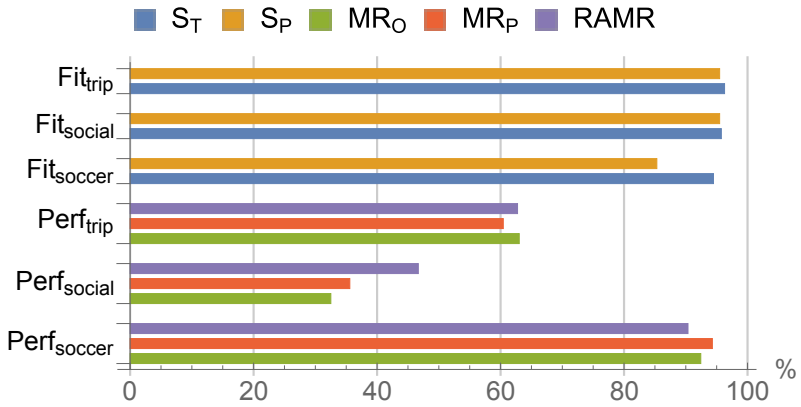


Figure 5.7: Predictions fits, observed and predicted machine reductions and relative average machine reduction for both schemes and all evaluated real-world applications.

5.6.3.5 Results

Fig. 5.7 and Table 5.5 illustrate the results of the real-world application evaluation. The average prediction fit of 93.7% for all three applications further supports the proposed scheme models. Table 5.4 shows the observed- and predicted machine reductions OMR and PMR with the relative average machine reduction $RAMR$ as modelled in Equation (5.29). The machine reduction compares the composite number of machines needed by both schemes with equal target request flows T. All applications in Table 5.5 need fewer machines (63%, 32%, 92%) if they use the proposed scheme. The proposed scheme S_P explicitly needs time to process dependencies while the traditional scheme S_T spends the time to manage its cache implicitly in the measured application delay $d_{p,A}$. The break-even point for dependency processing $d_{dp,W,BEP}$ calculates the delay where both schemes exhibit equal performance. The column DP BEP in Table 5.5 shows that the proposed scheme S_P has 2.69, 1.19 and 26.04 seconds to process all dependencies before the performance of both schemes is equal.

5.7 Discussion

In this chapter, a novel design pattern for resource storage and management and an optimised request flow scheme between components has been presented. With respect to **R5.1**: *Which design pattern can be used for optimised resource management?*, the novel Permanent Resource Storage and Management pattern divides resource models into individually scalable, manageable and decoupled read and pro-

cessing subsystem that guarantee constant response times for all read requests, releases applications from avoidable load and ensures that changes are processed exactly once for the whole system. With respect to **R5.2**: *How can components be composed and requests be flowed efficiently?*, an implementation of the PRSM pattern was presented that provides an efficient and scalable composition of components. Additionally, a mechanism to synchronously and asynchronously process dependencies in order to enforce eventual or strong consistency of resources was proposed. With respect to **R5.3**: *Which interface extension is required to enable the optimised scheme?*, a resource interface was presented as extension to the worker interface. The resource interface provides actions to manage the storage and meta information such as dependencies of resources. With respect to **R5.4**: *How can the performance be modelled analytically?*, component and composition parameters and models were developed that allow the analytical evaluation of request flow and total machines performance. Additionally, models to compare the performance between a traditional scheme and the proposed scheme were presented. With respect to **R5.5**: *Which metric enables to operate an application with optimal load?*, metrics and models were developed that are able to measure and optimise the performance of components by operating them in the optimal concurrency range. With respect to **R5.6**: *How is the real-world application performance of the proposed scheme compared to a traditional approach?*, results showed that all evaluated real-world applications need fewer machines (63%,32%,92%) with the proposed scheme than the traditional composition and flow scheme. Additionally, the results showed that the average time available to process dependencies was positive for all applications with 2.69, 1.19 and 26.04 seconds. The component, composition and mean prediction fit for all applications further support the proposed models.

5.8 Summary

In conclusion, it was shown that shifting the complex and time intensive processing of resource to the background allows creating a decoupled foreground layer with constant processing times. The application of the presented PRSM pattern furthermore can enhance the total performance of a system in terms of request throughput per second and total machines needed. By having a worker pull requests from a pool of processable requests, applications can be operated within their optimal concurrency range. The traditional push mechanism has no knowledge of the application's processing capabilities in terms of current and optimal number of requests per second. As modern web applications often keep state at the client, an eventual consistency of resources is an acceptable option. The state changes initiated by the user can optimistically and immediately be shown to the user while the actual processing is performed in the background. As presented in Section 5.3, the management model processes resource changes in the background. However, this chapter did not con-

sider detailed resource update mechanisms but calculated the time available for the updates. Consequently in the following Chapter 6, algorithms to optimise the resource dependency processing are elaborated in detail.

6. Resource Dependency Processing

6.1 Overview

In this chapter, the dependency structure and key graph measurements of web resources are examined. A longest-path algorithm using topological sort with dynamic programming is presented for efficient processing. Further, dependencies are analysed to find correlations between processing performance and graph measures. Two algorithms that base their parameters on six real-world web service structures, such as the Facebook Graph API are developed to generate dependency graphs. Further, a model is developed to estimate processing performance based on resource parameters. Finally, four series of graphs with increasing graph measures are analysed for the effects of the graph structure on the performance. The literature review in Chapter 3 identified six open research questions related to the analysis and optimisation of resource dependency processing:

- **R6.1**: How can resource dependencies be measured and stored?
- **R6.2**: What algorithm can be used to optimise the performance of processing?
- **R6.3**: What effects have dependency graph measures on the performance?
- **R6.4**: How can resource dependencies be generated?
- **R6.5**: How well can the dependency processing duration be modelled?
- **R6.6**: How is the performance compared with a typical traditional processing approach?

The remainder of the chapter is organised as follows: Section 6.3 identifies the structure and measures of resources and dependencies. Section 6.4 evaluates suitable graph processing algorithms, whilst Section 6.5 finds correlations between the processing duration and other graph measures. Section 6.6 introduces dependency graph and traffic generation algorithms and Section 6.7 develops the models for the approximation model where Section 6.8 evaluates both the performance and model fits with the evaluation cluster. Section 6.9 outlines the results and provides answers to the aforementioned research questions. Finally, Section 6.10 concludes the chapter with a perspective on the subsequent chapters.

6.2 Motivations and Objectives

The optimised request flow implementation based on the PRSM pattern presented in the previous Chapter 5 helps to distribute requests to multiple servers. The optimal

number of necessary servers has to be adapted to the encountered load continuously by scaling up and down. The scaling actions can occur based on the model presented in Chapter 5. Simultaneously to scaling, the high throughput of requests needs to be ensured with minimal processing delays in order to minimise waiting times for the users. Thus, finding an efficient RDB update mechanism is a key requirement to build a scalable web service architecture with optimised request routing. In Chapter 5, it was identified that the read/processing ratio has a major influence on the performance of the request flow. This stems from the increased processing overhead the proposed PRSM pattern exhibits over a traditional approach. By definition, the PRSM pattern trades a more complex processing for increased read performance. Hence, with a slow update performance, only read-driven applications can benefit from the RDB which limits the field of applications. In Chapter 5, a model to calculate the optimal number of machines with the highest request flow was presented. The model represents the time it takes to update changes with the dependency processing delay as parameter. For the scenarios evaluated in Chapter 5, the calculated dependency processing delay highlighted how much time there is available for dependency processing before the traditional scheme exposes the same performance as the proposed scheme. In this chapter, the constant dependency processing delay parameter is extended with an exact RDB update model. The model enables predicting the exact performance and allows evaluating the effectiveness of an optimised request flow approach for a distinct application. Further, it helps to optimise an existing application structure for fast processing. The remainder of this chapter puts a focus on optimising the structure and processing of web resource dependencies. Firstly, a graph is identified as the structure for expressing resources and its dependencies in addition to significant properties and measures. Different graph processing algorithms are evaluated in a shortest-path and longest-path problem domain, where a topological sort with dynamic programming is used to efficiently compute a valid dependency processing order alongside a model to calculate the processing duration. A linear correlation of the processing duration with the dependency depth and the cluster size is found and evaluated with a model and empirical data collected from the evaluation cluster presented in Section 2.8. Secondly, a simplified approximation model is developed allowing for precise and cost-efficient computation of the processing duration, the duration delta and relative performance improvement compared to traditional methods. The constant dependency processing delay from Chapter 5 becomes variable and enables a full calculation of the expected performance. Finally, a comprehensive performance comparison between a dependency processing approach and a traditional approach is conducted. The test applications are created from a service based graph algorithm that is developed using the APIs of six real-world applications and a fuzzy graph algorithm randomly creating applications based on certain measures.

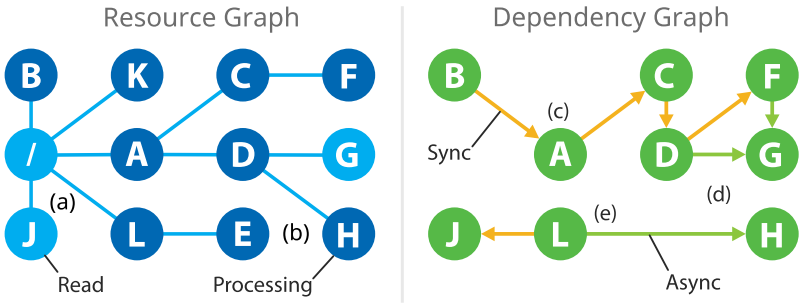


Figure 6.1: Resource graph with logical dependencies and resource dependency graph with synchronous and asynchronous dependencies

6.3 Resource Dependency Measurements

In order to minimise the expensive processing of requests, the declaration of resource dependencies that have to be processed for each request modifying data on the server is presupposed. Resulting from the processing, resources are stored in a distributed, cloud based resource database RDB. This opens up research question **R6.1**: *How can resource dependencies be measured and stored?*

6.3.1 Resource Vertices

A web service exposes multiple routes delivering different resources such as markup, structured data, images or videos to the consumer. The resource graph on the left side of Figure 6.1 illustrates the logical structure of a RESTful API with any base URL before / as root. The sub-resources $A...L$ are accessible via their unique URI , where (A, B, J, K, L) are direct descendants of the base URL offered as $/A...L$. Sub-resources nested deeper in the hierarchy, such as H, are offered by traversing through the graph: $/A/D/H$. The resource graph on the left side of Figure 6.1 represents the logical structure of a service and does not define dependencies between the resources. On the right side of Figure 6.1 explicit dependencies between resources are defined, where dependencies declare that a resource contains content of another resource. Table 6.1 presents a detailed list of different graph types used in this work.

6.3.1.1 Processing & Read Vertices

The standard HTTP methods (e.g., GET, PUT, POST, or DELETE) are used to act on a resource. Each resource vertex is uniquely identified by a (HTTP method, URI) tuple, so (GET, $/A$) is distinct from (POST, $/A$). As Figure 6.1 shows, each resource vertex is assigned to be either a processing vertex at Figure 6.1 (b) or a read vertex at Figure 6.1 (a). Read vertices are vertices allowing only GET or HEAD methods in the tuple and processing vertices are vertices with all other HTTP methods. This distinction can be used to apply optimised request routing mechanisms, where read vertices and processing vertices can be handled by separate, individually scalable subsystems.

Table 6.1: Conceptual distinction of dependency related graph types.

Resource Graph	All resources of a web service. Nesting of resources determines edges.
Dependency Graph	Dependencies between resources only. No resources that have no dependencies.
Structure Graph	Logical resource structure with unexpanded entities. Can be expanded to a resource graph.
Processing Tree	Tree that considers processing precedence constraints for a single resource as root.

6.3.2 Dependency Edges

To be able to keep the resource database up to date automatically, each resource of a web service needs to declare which other resources it influences. As an example, the creation of a blog post has a dependency on the sitemap where the blog post is listed. Additionally, the blog post is listed in the resource for all posts and it might also be listed on the index site of the blog as well. As shown on the right side of Figure 6.1, each dependency forms a directed connection between two resources. Hence resource dependencies are modelled as a directed graph. Table 6.1 presents the conceptual distinction of dependency related graph types, where the required dependency declaration extracts a dependency graph from a resource graph.

6.3.3 Graph Measures

In addition to well-known graph measures such as vertex count, edge count, vertex degree and clustering coefficient, dependency graphs exhibit special measures helping to classify and evaluate the dependency structure of web services.

6.3.3.1 Dependency Depth

The Dependency Depth $ddep(v)$ is the length of the longest-path of all dependencies connected to a vertex. In Figure 6.1 (c), the longest-path for B is (B, A, C, D, F, G), so the length is $ddep(B) = 5$. For L, $ddep(L) = 1$ and for J, $ddep(J) = 0$. The depth is important as it denotes the maximum number of steps required for dependency processing and is detailed in Section 6.4. The mean dependency depth of all vertices is used in the approximation model in Section 6.7 to calculate the average processing steps needed. In general, web services should try to minimise the dependency depth as it has a direct influence on the performance.

6.3.3.2 Dependency Degree

The Dependency Degree $ddeg(v)$ is the number of outgoing dependency edges for each vertex. In Figure 6.1 (d), vertex D has the highest dependency degree with $ddeg(D) = 2$. The mean dependency degree of all vertices can be used to estimate the portion of parallel processing, as all children of a vertex can be computed in parallel without collisions.

6.3.3.3 Read-Processing Vertex Ratio

The Read-Processing Vertex Ratio $RPVR$ denotes the proportion of read-only vertices to processing vertices in the resource graph. In Figure 6.1 (a), 3 of 12 vertices are read vertices so the $RPVR = 3/12$. The ratio is mainly interesting for the modelling of dependency graphs in Section 6.6 as read vertices do not have any processing dependencies that need to be processed. A high $RPVR$ generally indicates fewer dependency processing steps.

6.3.3.4 Cluster Count

Figure 6.1 (e) illustrates $CC = 2$ clusters in the dependency graph where one cluster is formed by the connection of (B, A, C, D, F, G) and the other is formed by the connection of (J, L, H). Clusters generally give an indication on the structure of a web service where a low number of clusters indicates a deeply joint application.

6.3.3.5 Cluster Size

The cluster size measures the mean number of connected vertices in a graph which helps finding the longest-paths in each cluster. In Figure 6.1 (e) the cluster size is calculated by counting the vertices in each cluster and dividing them by the number of clusters: $CS = (6 + 3)/2 = 4.5$.

6.3.3.6 Sparsity

A dense graph is a graph in which the number of edges is close to the maximal number of edges. From the data presented in Section 6.6, however, it follows that dependency graphs are sparse graphs. In order to express the sparsity of a graph, the sparsity S is defined to equal the number of edges divided by the number of vertices. In Figure 6.1 (a) the sparsity $S = 10/9$, where a sparsity of one denotes an equal presence of edges and vertices.

6.4 Processing Algorithms

For an efficient processing of dependencies the problem is formulated as follows: Given a vertex v from a dependency graph DG, how long does it take to process all dependencies of v while ensuring correct processing order. The key metric to optimise is the dependency processing duration $d_P(v)$ which opens up research question **R6.2**: *What algorithm can be used to optimise the performance of processing?* As Figure 6.2 (a) shows, a dependency graph defines multiple contingent processing paths. The resource vertex E can be reached via the path (A, B, E) and (A, C, E). The dependency graph states both B and C need to be finished processing before E can be processed as it contains changes from both B and C. Hence, an algorithm is needed to calculate the fastest and sequentially correct processing path. The proposed approach to calculate the processing path and thereby the duration is to

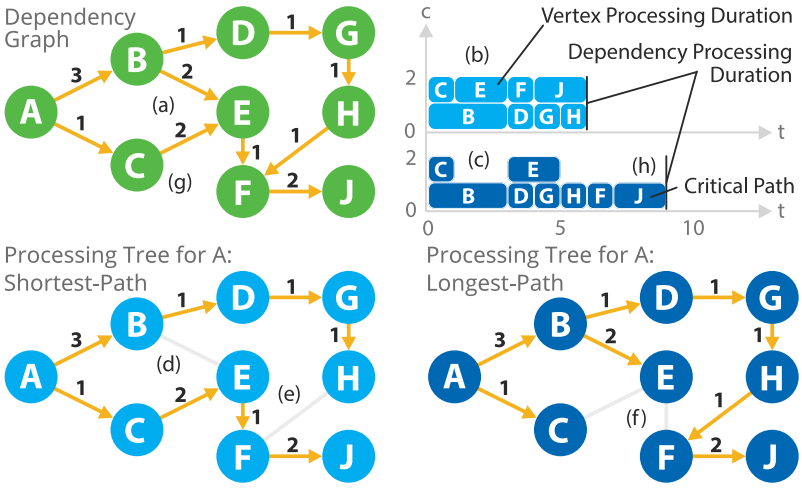


Figure 6.2: Dependency processing scheduling problem using a processing tree with a shortest-path and a longest-path approach.

weight the edges of the dependency graph with the mean processing delay introduced by the vertex the edge points to. This puts the algorithm into the domain of parallel precedence-constrained job scheduling algorithms (Sedgewick, 2014; Ray, 2013; Cormen, 2009) which generally determine a possible order of constrained events.

6.4.1 Evaluation

To find and evaluate suitable algorithms, 1000 random dependency graphs are generated with a custom created Incremental Edge Add (IEA) graph generation algorithm. The IEA generation algorithm starts with 1000 completely disconnected resource vertices in the first generation graph. In each next generation, it adds a random, directed edge to the previous generation graph while ensuring no cycles are created. The 1000 generations of the graph are build with a maximum number of 1000 edges. Each resource vertex introduces a processing delay of 100ms. In order to validate the algorithms presented in the subsequent sections with the 1000 dependency graphs, the following hypothesis is formulated:

- **H6.1**: With an increasing number of edges, the total processing duration increases monotonically.

The hypothesis is evaluated with a model of the applied algorithm and an empirical data collection on the *Pi-One* evaluation cluster presented in Section 2.8. The dependency processing algorithms are implemented using the Go programming language and evaluated with one million HTTP requests to measure the dependency processing duration.

6.4.2 Shortest-Path Approach

Firstly, algorithms to find the shortest processing paths in a graph are examined. Figure 6.2 (d) illustrates how a shortest-path algorithm extracts a shortest-paths tree (SPT) (Sedgewick, 2014) for the resource vertex A from the dependency graph

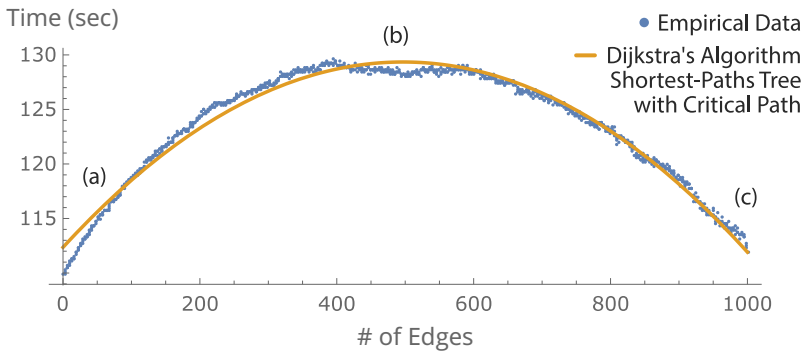


Figure 6.3: Shortest-Paths tree evaluation of 1000 dependency graphs with an increasing number of edges.

in Figure 6.2 (a). The edges (C, E) in Figure 6.2 (d) and (H, F) in Figure 6.2 (e) are removed as it takes longer to compute E after B and F after H. Figure 6.2 (b) shows how the total processing time is reduced by an optimal ordering of the processing blocks where the total dependency processing duration for A is 6. To find the shortest-paths processing tree (SPT) for all resource vertices Dijkstra's Algorithm is implemented to calculate the maximum processing duration from the critical path, which is the longest-path in the tree.

6.4.2.1 Results

Figure 6.3 illustrates the results from the shortest-paths evaluation. The processing duration starting at Figure 6.3 (a) increases with the number of edges. However, with the presence of more edges a shortest-path algorithm is able to reduce the processing duration as new shortcuts are added to dependencies previously connected through a longer path. This leads to a maximum at Figure 6.3 (b) which is further reduced as the number of edges increases at Figure 6.3 (c). The hypothesis **H6.1** needs to be rejected for a shortest-path approach. Consequently, the shortest-path approach is not feasible for dependency processing. It violates the constraint to strictly respect the order of processing by always selecting the fastest path available. Figure 6.2 shows this in (d), where E is processed after 1 time unit but B takes 3 time units to finish. Changes from B are not reflected in E, which violates the dependency edge (B, E).

6.4.3 Longest-Path Approach

From the collected data and work in Sedgewick (2014), Ray (2013) and Cormen (2009) it is concluded that a longest-paths processing tree (LPT) guarantees the precedence constraints for dependency processing. Generally, longest-path finding for a directed graph is an NP-hard problem which can not be solved efficiently for large datasets. However, if the problem is constrained to directed acyclic graphs, efficient algorithms exist. For dependency graphs this means, that in the design phase of a web service an automatic analysis tool continuously checks for cycles

in the dependency declaration and reports an error when a cycle is encountered. Figure 6.2 (f) illustrates how a longest-path processing tree for resource vertex A is extracted by deleting the edges (C, E) and (E, F). The scheduling solution in Figure 6.2 (c) shows that with 9 time units, the longest-path solution takes longer than the shortest-path solution with 6 time units. In contrary to the shortest-path solution however, the longest-path solution guarantees that all dependency constraints are satisfied. The total processing duration is determined by the critical path A, B, D, G, H, F, J in Figure 6.2 (h) which equals to the longest-path in the longest-path processing tree.

6.4.4 A Forest of Processing Trees

Figure 6.2 (g) additionally shows that processing trees are distinct for each resource vertex. The longest-path tree in Figure 6.2 provides the solution for resource vertex A. Resource vertex C can not use the solution for A as it has no dependencies in the processing tree for A. The correct processing path for C is C, E, F, J. In conclusion, each resource vertex stores its individual processing tree thus leading to a forest of processing trees.

6.4.4.1 Time Complexity

In order to calculate the time complexity of the algorithms, the maximum number of edges $E_{\max}$ in a directed acyclic graph DAG with V vertices is examined. Using an adjacency matrix, a dependency graph with maximum edges can be constructed by filling either the upper or the lower triangular part of the matrix excluding the diagonal with edges where a 1 marks the presence of an edge between two vertices:

$$E_{\max}(V) = \frac{1}{2} \cdot (-1 + V) \cdot V \tag{6.1}$$

Using each vertex as a root for a subgraph of a DAG, the maximum number of vertices $V_{sub,\max}$ in each subgraph must decrease by at least one to ensure no cycles are present:

$$V_{sub,\max}(V) = \sum_{r=0}^{V} V - r = \sum_{r=1}^{V} r \tag{6.2}$$

6.4.5 Forest of Processing Trees Extraction Algorithms

Two algorithms to determine the longest-path trees efficiently are analysed: A version of Bellman-Ford that uses negated edge weights and a novel algorithm that is based on a topological sort and uses dynamic programming.

6.4.5.1 Negated Bellman-Ford

As Dijkstra's algorithm, the Bellman-Ford algorithm calculates the shortest-path in a graph. In contrast to Dijkstra's algorithm however, it can deal with negative edge weights. Sedgewick (2014) proves the longest-path problem in edge-weighted directed acyclic graphs to be solvable by finding the shortest-paths in a graph where all edge weights are negated. For dependency graphs, this allows to calculate a processing tree with the following steps:

1. Calculate the shortest-paths processing tree for a vertex v with Bellman-Ford $SPT = BF(-DG, v)$, where $-DG$ is the dependency graph with negated edge weights.
2. Convert the solution to a longest-path tree by negating it where $LPT = -SPT$.
3. Find the critical path in LPT. The sum of the edges equals the dependency processing duration for v.

The Bellman-Ford algorithm takes time proportional to $E \cdot V$. The algorithm needs to be executed for every resource subgraph of the dependency graph, so from (6.1) and (6.2) follows that:

$$\sum_{v=1}^{V} E_{\max}(v) \cdot v \tag{6.3}$$

Albeit suitable, the Bellman-Ford algorithm is not used in this thesis as a faster and more efficient algorithm introduced in the following section exists.

6.4.5.2 Topological Sort with Dynamic Programming

By topologically sorting a DAG, a linear ordering of vertices is generated guaranteeing a vertex v_1 to come before a vertex v_2 if an edge $v_1 \rightarrow v_2$ exists. If dependencies are processed by the calculated order, it is guaranteed all changes are reflected in all dependent resource vertices. The approach can be optimised further by finding branches of jobs eligible for parallel processing as their outputs do not depend on other resources. Sedgewick (2014) proposes an algorithm to find a longest-path tree from a root vertex in linear time. The proposed algorithm however only calculates a single longest-path tree, where for dependency processing the longest-path trees for all resource vertices as root nodes are needed. In addition, the length of the critical path for each vertex is needed to model the processing duration. Therefore, the algorithm is extended with a dynamic programming approach so the computed result returns all processing trees as forest, as well as the durations of all critical paths:

1: $delays \leftarrow$ processing delays of vertices of DG
2: $order \leftarrow$ topologically sorted vertices of DG

3: $forest \leftarrow$ subgraphs for all v out components
4: $durations \leftarrow 0$ for all vertices of DG
5: **for all** vertices v in DG **do**
6: $suborder \leftarrow$ intersection of v in $forest$ and $order$
7: $distances \leftarrow -\infty$ to each vertex t in $forest$
8: $distances[v] = delays[v]$
9: **for all** vertices s in $suborder$ **do**
10: **for all** children c of s **do**
11: $d = distances[s] + delays[c]$
12: **if** $distances[c] < d$ **then**
13: $distances[c] = d$
14: **if** $durations[v] < d$ **then**
15: $durations[v] = d$
16: **end if**
17: **end if**
18: **end for**
19: $parents \leftarrow$ parents of vertex $forest[v][s]$
20: **if** length of $parents > 1$ **then**
21: $max \leftarrow$ maximum $distances$ of $parents$
22: **for all** vertices p in $parents$ **do**
23: **if** $distances[p] < max$ **then**
24: delete edge $p \rightarrow s$ from $forest[v][s]$
25: **end if**
26: **end for**
27: **end if**
28: **end for**
29: **end for**
30: **return** $[forest, durations]$

The proposed Forest of Processing Tree Extraction (FPTE) algorithm applies dynamic programming by calculating the topological order only once for the entire set of vertices. A subproblem is defined as finding a single-source longest-path tree based on the total order. The original algorithm from Sedgewick (2014) calculates only a single longest-path tree for a selected vertex. In detail, the FPTE algorithm initialises its data structures (lines 1-4), calculates the topological order of all vertices and creates arrays for the distances and durations for each root vertex v. For each vertex, subgraphs are generated (line 3) from where the algorithm step-by-step removes the shortest-path edges. Using each vertex once as root vertex (line 6), the vertices of the subgraph are extracted in suborder. The distance to each vertex in the subgraph is set to $-\infty$ on line 7, where the distance of a node to itself is the processing delay as shown on line 8. Next, the vertices are traversed in this subor-

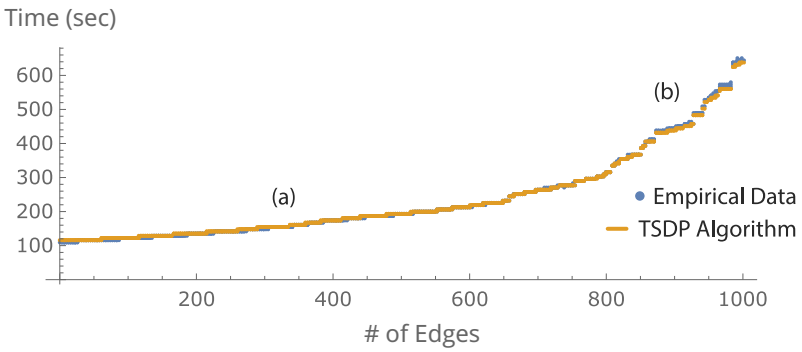


Figure 6.4: Longest-path tree evaluation of 1000 dependency graphs with an increasing number of edges.

der and each child is expanded (lines 9-10). As a next step, the edges are relaxed by checking if the currently stored distance to the child is smaller then the current path (lines 11-12). If so, a new longest-path is found to the child and stored as new longest distance and duration (lines 13-16). Additionally, the calculation of the critical path durations is injected into the original single-source algorithm's relaxation step to reduce the runtime (lines 14-16). By line 18, the maximum distance to the vertex s is known. If multiple edges point to s, then the edges from the parents with the lowest distance can be removed to keep only longest-paths. At the end of the loop for v (line 28), the subgraph is fully converted to a longest-path tree. The single-source longest-paths algorithm from Sedgewick (2014) takes time proportional to $E + V$ as it visits each vertex and each node exactly once. Following (6.1) and (6.2), the proposed FPTE algorithm determining all longest-paths and critical paths durations takes time proportional to:

$$\sum_{v=1}^{V} E_{\max}(v) + v \tag{6.4}$$

Hence, it is faster than the Bellman-Ford algorithm.

6.4.5.3 Results

Figure 6.4 shows the results from the longest-path algorithm evaluation. The processing duration starts with a low slope at Figure 6.4 (a) and then increases in a non-linear fashion towards Figure 6.4 (b) with the number of edges. The durations calculated with the FPTE algorithm matches to 99.4% with the empirical data collected on the computing cluster. Consequently, hypothesis **H6.1** can be supported for the longest-path approach as the durations increase monotonically with the number of edges. To further understand the most influential factors of dependency processing, the processing duration correlations are analysed in detail in the following section.

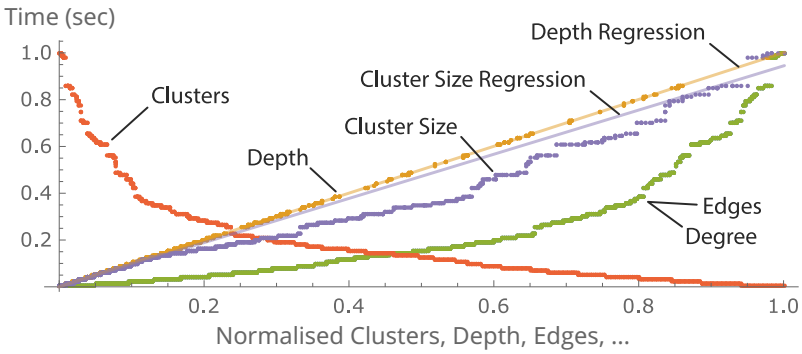


Figure 6.5: Normalised correlations of the processing duration with the number of edges, mean dependency degree, dependency depth, number of clusters and mean cluster size.

Table 6.2: Correlations of dependency measures with the processing duration.

Measure	R	R^2	**Function**	$RMSE$	Fit
Edges	0.89	0.8	$102 + 0.000405x^2$	32.35	0.93
Degree	0.89	0.8	$102 + 405x^2$	32.35	0.93
Depth	1	1	$113 + 0.1x$	0	1
Clusters	-0.84	0.72	$25.12 + 83034/x$	20.25	0.96
Cluster Size	0.98	0.97	$25.12 + 83.034x$	20.25	0.96

6.5 Dependency Analysis

In the previous section, the dependency processing algorithms were evaluated with 1000 generations of a dependency graph. The results in Figure 6.4 show the correlation between the processing duration and the number of edges without detailed analysis. This opens up research question **R6.3**: *What effects have dependency graph measures on the performance?* In this section, the correlations of the processing duration with the edge count, dependency depth, dependency degree, cluster count and cluster size are analysed in detail.

6.5.1 Correlations with Processing Duration

Figure 6.5 shows the normalised correlations of all dependency measures with the processing duration. For the analysis, the Pearson product-moment correlation coefficient R and the coefficient of determination R^2 are calculated to evaluate if a linear correlation between the measure and the processing duration exists. Additionally, a linear and nonlinear model, the Root-mean-square error $RMSE$ and Fit are calculated for each measure. Fit is determined using the normalised version of the $RMSE$ with $1 - NRMSE$ to denote the model fit. The best-fit functions along with the determined correlation metrics are shown in Table 6.2.

6.5.1.1 Edge Count

From the model fit in Table 6.2, the number of edges are found to have a quadratic effect on the processing duration shown in Figure 6.5. This stems from the fact that the probability to connect multiple vertices by adding a new edge is lower for initial

generations of the graph, where many vertices are connected by a single edge only. With an increasing number of edges, the probability to connect multiple other edges with a new edge increases in a quadratic fashion. Thereby longer paths are created, thus increasing the processing duration.

6.5.1.2 Dependency Degree

In the same way as the number of edges, the dependency degree is a measure directly related to the probability of a vertex connecting to other vertices. The mean vertex degree increases along with the number of edges as the probability that a new edge connects a vertex to a longer path also increases in a quadratic fashion (Table 6.2). Hence, the normalised correlation of the dependency degree shown in Figure 6.5 matches the normalised correlation of the number of edges.

6.5.1.3 Dependency Depth

Concluding from the correlation of the edge count and the vertex degree, the mean dependency depth has a linear influence on the processing duration as it directly reflects the average length of the longest-paths. Figure 6.5 shows this linear correlation along with Table 6.2, where the slope of the function matches the mean processing delay for each vertex.

6.5.1.4 Cluster Count

The cluster count expresses how many unconnected clusters of vertices exist. Figure 6.5 illustrates how more clusters lead to shorter processing paths and thereby influence the processing duration inversely as shown in Table 6.2.

6.5.1.5 Cluster Size

The cluster size signifies the mean length of connected components and thereby dictates the maximum length for the longest-paths. As shown in Table 6.2, the processing duration increases linearly with the cluster size. In Figure 6.5 however, the duration stays below a perfect linearity for the majority of the time and then jumps above the line close to the end. For a graph without edges, all vertices are disconnected from each other. Thus, the number of clusters equals the number of vertices with a cluster size of exactly 1. By adding some edges, small groups of vertices become connected forming clusters of small sizes, e.g. 2-3. With the further addition of more edges, small clusters become connected to other small clusters forming clusters of larger sizes, e.g. 30-40. This continues until eventually all clusters are connected to one single cluster. As a result, the cluster size jumps whenever multiple clusters join to a single cluster until finally the size equals the number of vertices.

6.5.2 Regressions for Processing Duration

Based on the results from the correlation analysis, two regression models are build in order to approximate the processing duration for a given dependency graph. Two measures build a linear correlation with the processing duration: the cluster size and the dependency depth.

6.5.2.1 Cluster Size Based

For sparse graphs with a sparsity $S \lessapprox 1$ the cluster size CS is steady and can be used to estimate the processing duration. Using the mean processing delay d_p and the network delay d_n from Chapter 5 the regression can be modelled as follows:

$$d_{reg,CS} = CS \cdot d_p + d_n \quad \text{if } S \lessapprox 1 \tag{6.5}$$

Figure 6.5 shows the cluster size regression with a model fit of 96%. The cluster size can be calculated very efficiently for the whole dependency graph through its weakly connected components, however with an increasing sparsity S the approximation results deteriorate.

6.5.2.2 Depth Based

A more exact approximation of the processing duration can be performed using the processing depth if there are more edges than vertices in the graph. However, it is more expensive to calculate the processing depths using a depth-first search algorithm, as the depth has to be computed for every starting vertex. Equation (6.1) and (6.2) denote the maximum number of edges and vertices for a DAG, where the runtime for a depth-first search generally is limited to $V + E$. Using the depth $ddep$, the regression can be modelled as follows:

$$d_{reg,ddep} = d_p + d_p \cdot ddep + d_n \tag{6.6}$$

Figure 6.5 shows the depth regression with a model fit of 1 as the processing delays for the evaluation are normally distributed around 0.1 (Table 6.2). The error of the regression is distributed exactly as the mean processing delay serving as regression slope.

6.6 Service Generation

To compare the performance of resource dependency processing with a traditional cache-eviction approach, web services consisting of dependency graphs and traffic traces are generated. Existing web services do not declare resource dependencies explicitly, hence no data is available and the graphs must be generated. This opens up research question **R6.4**: *How can resource dependencies be generated?* For the generation, two algorithms are developed in this section. The first algorithm bases

Table 6.3: Graph and traffic parameters with distributions used to generate evaluation data for the performance comparison.

Graph Based	**Parameter**	**Distribution**
Vertices	V	Uniform$(100, 1000)$
Edges	E	Based on clusters C and CS
Read/Processing	RPR	Uniform$(0, 1)$
Read Request	RR	Bernoulli(RPR)
Cache Hit/Miss	HMR	Uniform$(0, 0.7)$
Processing Delay	d_p	HyperErlang(Uniform(1,10))
		Weibull(Uniform$(0.1, 10)$, 1)
		Pareto(0.001, Uniform$(1, 10)$)
		Lomax(0.001, Uniform$(1, 10,\ 0)$)
Clusters	CC	Uniform$(10, 100)$
Cluster Size	CS	Uniform$(3, 10)$
Traffic Based	**Parameter**	**Distribution**
Duration	D	Const(20)
Requests	R	Uniform$(1000, 4000)$
Path	P	Zipf(V, Uniform$(10^{-6}, 0.1)$)
Offset	O	FARIMA(R, D)
		CMMPP(R, D)
		FractionalBrownianMotion(R, D)
		PoissonParetoBurstProcess(R, D)

its parameters on extracted values of six social network application APIs and the second algorithm selects its parameters at random.

6.6.1 Parameters

For the generation, parameters are identified as listed in Table 6.3. Parameters either relate to the dependency graph or the traffic traces.

6.6.1.1 Dependency Graph Based

The number of vertices V for each graph is distributed uniformly between 100 and 1000. The read/processing ratio RPR identifies the fraction of all resources that are read only. A $RPR = 0.3$ means that 30% of all vertices are read only. For each vertex in the graph, it is determined whether the vertex is a read or processing vertex using a Bernoulli distribution distributed by the RPR. The cache hit/miss ratio HMR determines how many vertices of the whole graph are marked as cached. This parameter is only used by the traditional cache-eviction approach. Based on work of Rajabi and Wong (2014) and Poggi, Carrera, Gavalda et al. (2014) the processing delay d_p for each vertex is calculated by uniformly choosing one of the distributions listed in Table 6.3 for each graph. The HyperErlang(n) distribution uses a probability vector of length n and the other distributions follow the standard signatures Weibull$(\alpha,\ \beta)$, Pareto(k, α) and Lomax(k, α, μ) where α is shape, β is scale, μ is location and k is a minimum value parameter. The number of clusters

and the cluster size is uniformly selected for each graph.

6.6.1.2 Traffic Based

For each graph, traffic for the duration D is generated with a uniformly selected number of requests R. Based on the work of Katsaros, Xylomenos and Polyzos (2012) and Visala, Keating and Khan (2014) the resource popularity P can be modelled using a Zipf(n, ρ) distribution where n is the range and ρ is the Zipf parameter. The offset O determines the arrival time of each request. The self-similar offset is modelled following the work of Dick, Yazdanbaksh, Tang et al. (2014), Chen, Ghorbani, Wang et al. (2014), Zukerman, Neame and Addie (2003), Chen, Addie, Zukerman et al. (2015) and Donthi, Renikunta, Dasari et al. (2014) by using a Fractionally Autoregressive Integrated Moving-Average process FARIMA(R, D), a Circulant Markov-Modulated Poisson process CMMPP(R, D), FractionalBrownianMotion(R, D) and a PoissonParetoBurstProcess(R, D) where R is the number of arrivals and D is the arrival interval. For FARIMA the AR coefficients are set to 0.99, the MA coefficient is random uniformly distributed between 0 and 0.49 and the white noise has a variance of 1. For the CMMPP a superposition of four two-state arrival rate vectors is used with a maximal arrival rate of 500. This rate is based on the maximum throughput of a single node in the evaluation cluster. The Fractional Brownian Motion uses a uniformly distributed hurst index between 0.5 and 0.99 in order to ensure self-similarity and the λ parameter of the Exponential distribution for the Poisson Pareto Burst process limits the maximum arrival rate to 500. For each graph, one of the models is selected at random for generation.

6.6.2 Service Based Graph Generation

In order to generate random dependency graphs exhibiting real-world properties, an algorithm is developed by extracting parameters from six social network services: The Facebook Graph API v2.2, the Twitter API v1.1, the Tumblr API v1, the Instagram API v1, the Google Plus API v1 and the SoundCloud API v1.

6.6.2.1 Service Structure Graphs

For each service, all API resources are extracted as vertices while the dependencies of the resources are extracted as edges into an API structure graph. Dependencies are not declared in the API specifications, hence the effects of a request to a resource are analysed by comparing changes in all resources before and after a request. The changed vertices have a dependency on the initially requested vertex and need to be added as dependency edges. Figure 6.6 illustrates all extracted service structure graphs. At Figure 6.6 (a) the Facebook structure graph is strongly connected with the central vertex representing a user's feed. Figure 6.6 (b) shows multiple sub-resources of a tumbler post that are updated with the post and Figure 6.6 (c) highlights weakly connected clusters with sparse dependencies of the Twitter API.

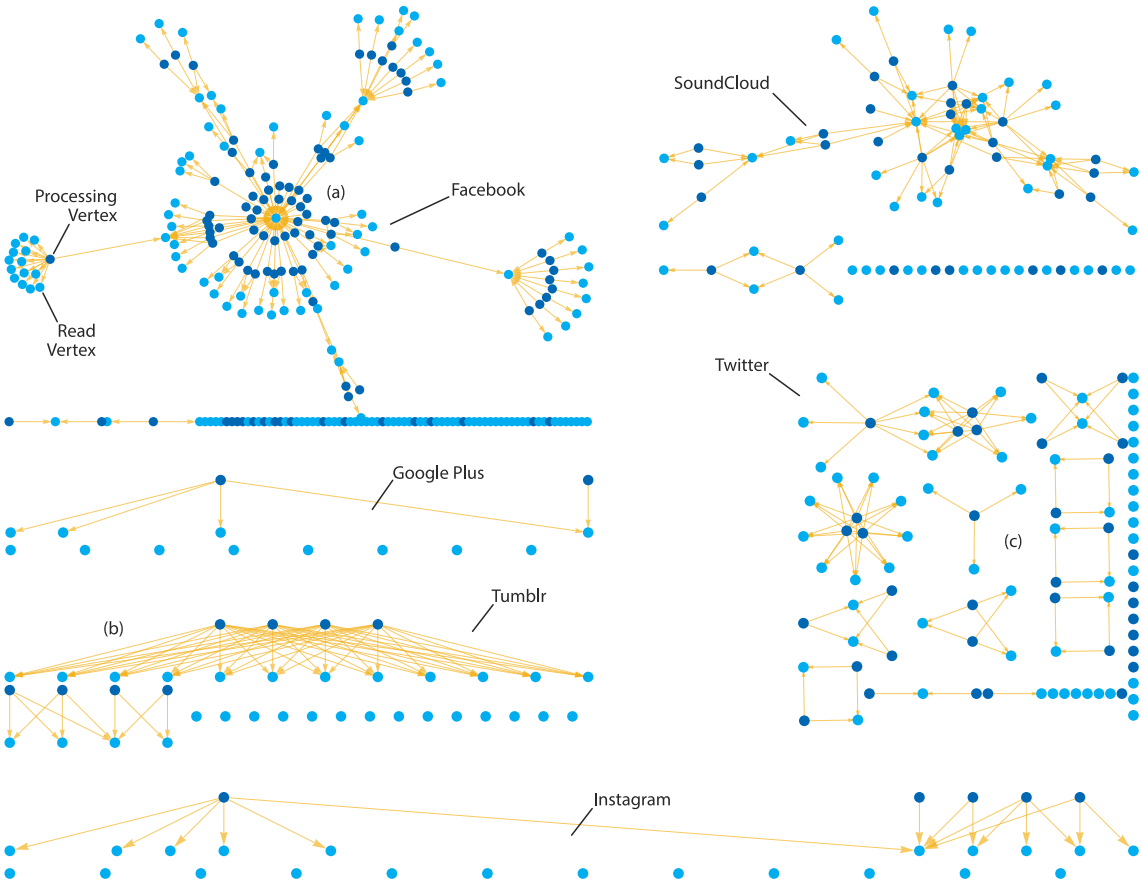


Figure 6.6: API structure graphs of six inspected real-world services with read and processing vertices.

Table 6.4: Key figures of the extracted service parameters.

Measure	Parameter	Min	Max	Mean	Var
Read/Processing Ratio	RPR	0.58	0.85	0.71	0.014
Processing Delay	d_p	0.004	0.18	0.09	2.52
Cluster Size	CS	14	235	81	6642
Dependency Depth	$ddep$	0	4	0.38	0.71
Dependency Degree	$ddeg$	0	14	0.57	2.79

For detailed inspection of all graphs the full evaluation dataset is provided as download available at webscalingframeworks.org/graphs (2016).

6.6.2.2 Parameter Extraction

From the service structure graphs the parameter ranges are extracted to be used for the generation of random dependency graphs. The results are presented in Table 6.4. Using a goodness-of-fit hypothesis test, the measured parameters do not follow a distribution. Hence, the algorithm selects random elements uniformly from all captured parameter data.

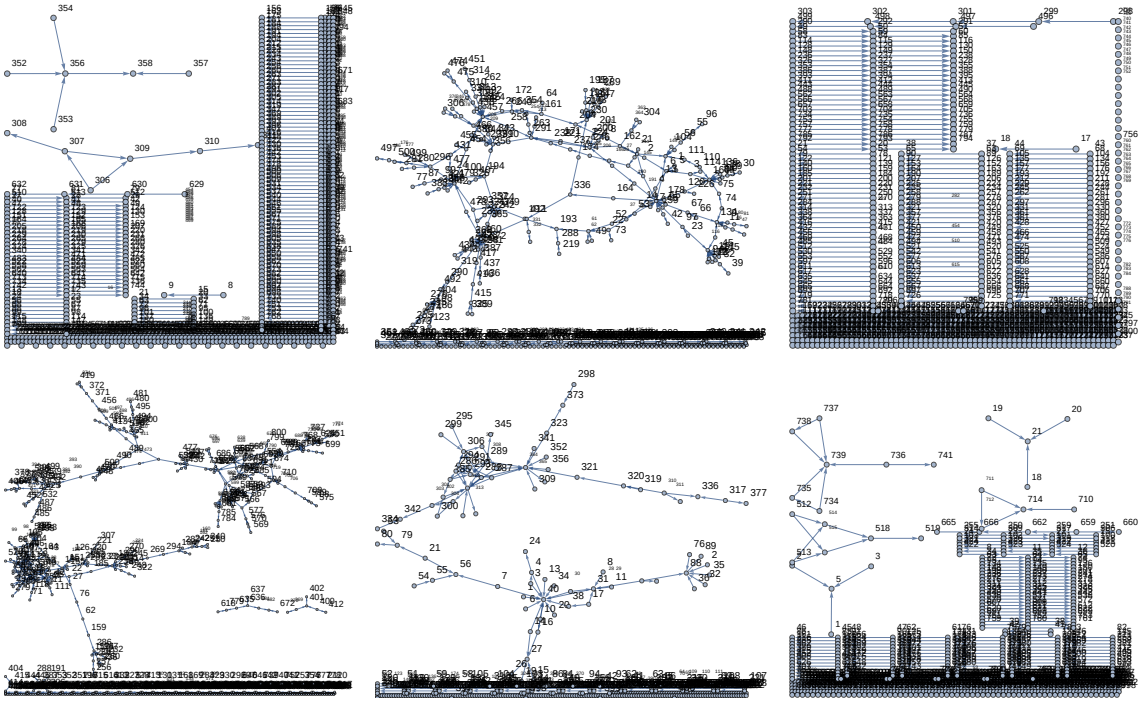


Figure 6.7: Resource graphs generated with the service based graph algorithm.

6.6.2.3 Algorithm

The Service Based Dependency Graph (SDG) algorithm is developed by superposing and manipulating multiple adjacency matrices as follows:

1. Create a sequencer function for the dependency depth that returns continuous sequences of 1s followed by a terminating 0, where the length of the sequence is drawn randomly from all service structure graph depths, e.g. $111011011110\ldots$ for $3, 2, 4$.
2. Create a sequencer function for the dependency degree in the same fashion as the depth sequencer function.
3. Create a $N \times N$ matrix, where N is drawn randomly from all service structure cluster sizes and fill it with 0s.
4. Fill the matrix diagonal above or below the main diagonal with a sequence from the depth sequencer function.
5. Fill the matrix columns until the diagonal above or below the main diagonal with sequences from the degree sequencer function.
6. Create another matrix of size N and fill it with random samples of a Bernoulli distribution where the probability equals a random read/processing ratio.
7. Multiply the read/processing matrix with the depth/degree matrix.
8. Repeat 3-7 and concatenate the resulting cluster matrices until the desired number of vertices is reached.

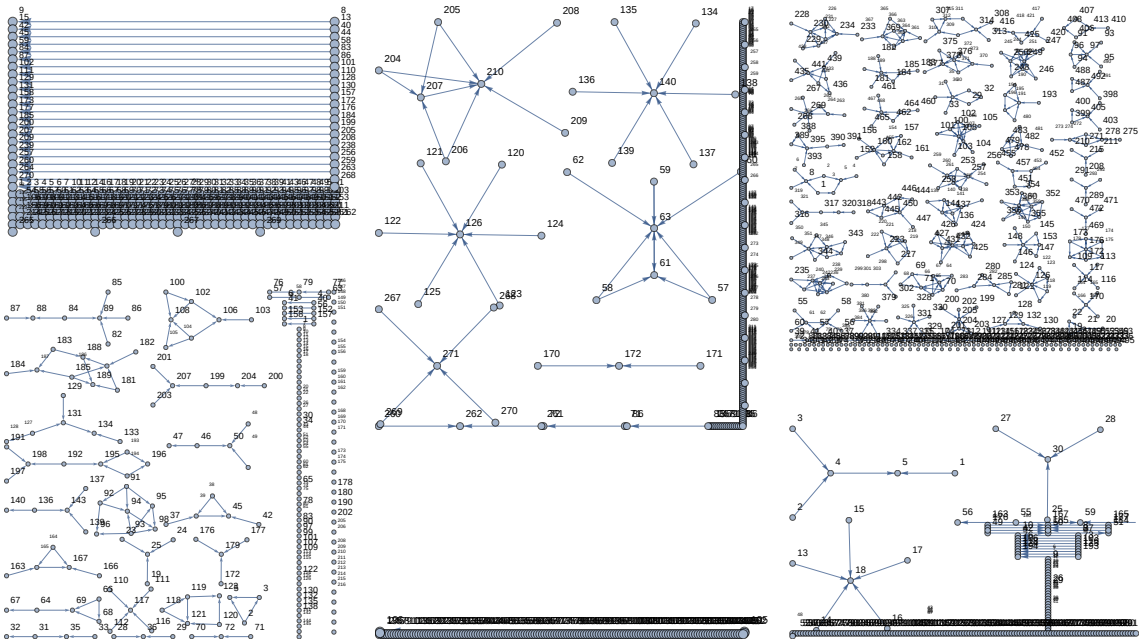


Figure 6.8: Resource graphs generated with the fuzzy graph algorithm.

By strictly manipulating either the upper or lower triangular portion of an adjacency matrix, the directed acyclic graph property of the resulting matrix is ensured. In addition, for each vertex a processing delay is drawn randomly from the service data. Furthermore, the vertex is marked as cache-hit or miss using the distribution shown in Table 6.3. The only input parameter to the SDG algorithm is the number of maximum vertices to be used as termination criteria. Figure 6.7 shows sample resource graphs generated with the SDG algorithm.

6.6.3 Fuzzy Graph Generation

As the service based graph generator strictly uses parameter values drawn from the analysis of the service structure graphs, a supplementary Fuzzy Dependency Graph (FDG) algorithm is developed creating graphs with a wider range of parameters. This ensures the evaluation is not overfitted to the analysed service structure graphs. The fuzzy graph generation steps are as follows:

1. Create C adjacency matrices with size V/C and randomly distribute a total of E edges in the upper or lower triangular portion.
2. Multiply the columns with a sequence of 1 and 0 distributed by the read processing ratio.
3. Concatenate all resulting cluster matrices.

The distributions used for the parameters are listed in Table 6.3. As for the SDG, the FDG additionally determines a processing delay and cache-hit for each vertex. The input parameters to the FDG algorithm are the number of vertices V, the number of edges E and the number of clusters C. Figure 6.8 shows sample resource graphs generated with the FDG algorithm.

6.7 Performance Modelling

In order to analyse the performance analytically a model which calculates the processing duration is developed. The model is created for both a resource dependency approach and a traditional approach. The parameters serving as input to the model can be calculated from the structure of an application as shown in Section 6.3. Furthermore, the model allows to compare the performance of both approaches in order to find the approach best suited for a specific application. In Section 6.8 the model fits are determined to answer research question **R6.5**: *How well can the dependency processing duration be modelled?*

6.7.1 Processing Duration

The processing duration from Chapter 5 is extended by replacing the d_p constant with an equation explicitly calculating the processing duration based on the dependency graph.

6.7.1.1 Traditional Processing

The processing duration d_p for the TP approach where a web service directly receives every request but can only serve a fraction of the resources defined by the cache hit/miss ratio HMR from a cache is modelled as:

$$d_{p,TP} = HMR \cdot d_l + d_n + (1 - HMR) \cdot d_p \tag{6.7}$$

The lookup delay d_l describes the time it takes to lookup an item in the cache and the network delay d_n is modelled as linear or quadratic variable as shown in Section 5.5.

6.7.1.2 Resource Dependency Processing

The resource dependency processing (RDP) approach is based on the linear correlation between the cluster size CS and the processing depth $ddep$ as analysed in Section 6.5. If the sparsity is $S \lessapprox 1$ the processing duration is modelled as follows:

$$d_{p,RDP} = (d_n + CS \cdot d_p) \cdot (1 - RPR) \quad \text{if } S \lessapprox 1 \tag{6.8}$$

If the sparsity $S \gg 1$, the more expensive to determine dependency depth $ddep$ is used:

$$d_{p,RDP} = (d_n + d_p + ddep \cdot d_p) \cdot (1 - RPR) \tag{6.9}$$

In contrast to the traditional processing approach, the resource dependency processing approach receives only a fraction of all requests defined by the read/processing ratio RPR. All other resources are served directly from the resource database and

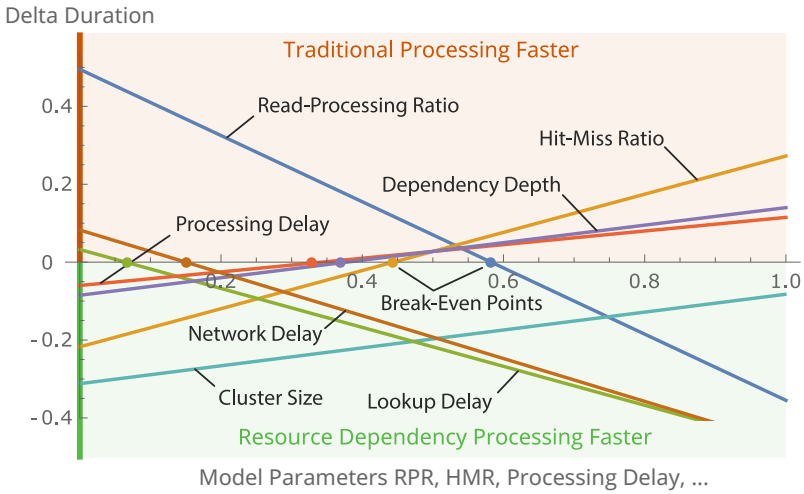


Figure 6.9: Analysis of the influence and break-points of all model parameters to the duration deltas.

do not influence the processing duration.

6.7.2 Processing Duration Delta

In order to compare both processing approaches, the processing duration delta can be modelled as:

$$\Delta d_P = d_{p,RDP} - d_{p,TP} \tag{6.10}$$

Figure 6.9 illustrates the influence of all model parameters on the duration deltas where each parameter is plotted in the range of 0 to 1 while all other parameters remain constant with default parameters $RPR = 0.45$, $HMR = 0.5$, $d_l = 0.01$, $d_p = 0.5$, $ddep = 0.5$, $CS = 0.5$ and $d_n = 0.1$. For negative duration deltas the resource dependency processing approach is faster than a traditional processing approach. A greater absolute slope in Figure 6.9 means that the analysed parameter has a greater influence on the processing duration. Consequently, the read/processing ratio RPR has the greatest influence on the processing duration as it directly affects the number of requests that need to be processed. The RPR is followed by the hit/miss ratio HMR which determines the amount of cached resources in the traditional processing approach. The processing delay d_p, the dependency depth $ddep$ and the cluster size CS have equal influences on both approaches. In the traditional approach, a greater lookup delay d_l negatively influences the caching performance and the network delay d_n is applied to every request. The resource dependency processing approach uses the processing tree, thus the network delay only applies to the initial request and its response.

6.7.3 Relative Performance Improvement

To calculate the factor of improvement the resource dependency processing exhibits over the traditional processing, the relative performance improvement is defined as

follows:

$$RPI = \frac{d_{p,TP}}{d_{p,RDP}} - 1 \tag{6.11}$$

For example, a positive RPI of 2.3 shows that the performance using resource dependency processing is 2.3 times better than the traditional processing performance. Similarly, a negative RPI means that a traditional processing is faster.

6.7.4 Break-Even Points for Processing Duration

A break-even point calculation allows to determine the exact value of a parameter where the processing duration of both the traditional processing TP and the resource dependency processing RDP are equal. The break-even point based on the dependency depth *ddep* can be calculated as follows:

$$ddep = \frac{-d_l \cdot HMR + d_p \cdot HMR - d_n \cdot RPR - d_p \cdot RPR}{d_p \cdot (-1 + RPR)} \tag{6.12}$$

Based on the cluster size CS, the break-even point is calculated as:

$$CS = \frac{-d_p - d_l \cdot HMR + d_p \cdot HMR - d_n \cdot RPR}{d_p \cdot (-1 + RPR)} \tag{6.13}$$

Figure 6.9 illustrates the break-even points for all parameters where Δd_P is zero.

6.8 Performance Evaluation

The analytical modelling of the performance opens up research question **R6.6**: *How is the performance compared with a typical traditional processing approach?* Thus, in this section the service based SDG and fuzzy graph FDG generation algorithms are used to compare the performance of the proposed resource dependency processing approach with a traditional processing approach. The evaluation is conducted in an aggregated combined case, best case, worst case and average case scenario, where the fit of the performance models developed in the previous section is evaluated. Further, four series of graphs with increasing graph measures are created to evaluate the influence of different structures on the performance. Finally, the results are mapped to structures observed in real-world APIs.

6.8.1 Aggregated Performance

For the aggregated evaluation 1000 web services are generated with the service based graph algorithm and further 1000 web services with the fuzzy graph algorithm. For each of the 2000 web services, a distinct traffic trace is generated following the distributions from Table 6.3. The key figures of all generated web services are listed in Table 6.5. The standard deviation (SD) of the path popularity P shows how much the requests spread to different resources and the SD of the offset O gives an

Table 6.5: Key figures of the generated evaluation data.

Service Based	**Parameter**	**Min**	**Max**	**Mean**	**SD**
Vertices	V	100.	1000.	537.6	265.2
Edges	E	18.	2127.	408.82	332.45
Sparsity	S	0.15	2.89	0.77	0.48
Clusters	C	41.	807.	338.62	188.7
Cluster Size	CS	1.16	3.06	1.68	0.38
Dependency Depth	$ddep$	0.15	1.69	0.58	0.29
Dependency Degree	$ddeg$	0.14	0.93	0.43	0.17
Processing Delay	d_p	0.	0.13	0.04	0.03
Cache Hit/Miss	HMR	0.	0.7	0.35	0.2
Traffic	**Parameter**	**Min**	**Max**	**Mean**	**SD**
Requests	R	1002.	4000.	2496.89	867.48
Read/Processing	RPR	0.	1.	0.49	0.29
SD Path Popularity	P	22.56	229.64	121.19	57.66
SD Offset	O	4.2	7.34	5.83	0.34
Fuzzy	**Parameter**	**Min**	**Max**	**Mean**	**SD**
Vertices	V	30.	1000.	347.11	215.46
Edges	E	0.	4095.	320.45	488.07
Sparsity	S	0.	4.18	0.82	0.82
Clusters	C	10.	872.	188.84	153.95
Cluster Size	CS	1.	10.	2.49	1.78
Dependency Depth	$ddep$	0.	4.05	0.72	0.72
Dependency Degree	$ddeg$	0.	4.04	0.71	0.72
Processing Delay	d_p	0.	0.22	0.07	0.05
Cache Hit/Miss	HMR	0.	0.7	0.36	0.2
Traffic	**Parameter**	**Min**	**Max**	**Mean**	**SD**
Requests	R	1000.	3994.	2488.43	870.15
Read/Processing	RPR	0.	1.	0.5	0.3
SD Path Popularity	P	7.68	227.	79.86	47.61
SD Offset	O	4.35	7.32	5.82	0.34

overview of the request arrival times. For detailed inspection of all graphs and traffic traces and to ensure the reproducibility of the results, the full evaluation dataset is provided as download available at webscalingframeworks.org/graphs (2016).

6.8.1.1 Implementations

All graphs are evaluated on the *Pi-One* evaluation cluster (Section 2.8) where a combination of two machines for a single test is used. One machine implements the web service with the generated resources and processing delays and the other creates requests according to the generated traffic trace. Each graph is tested with three different web service implementations: The traditional processing implementation responds with either the cached value or processes the value depending on the generated resources. The resource dependency processing implementation processes the request by traversing the processing tree while stalling the response until the full

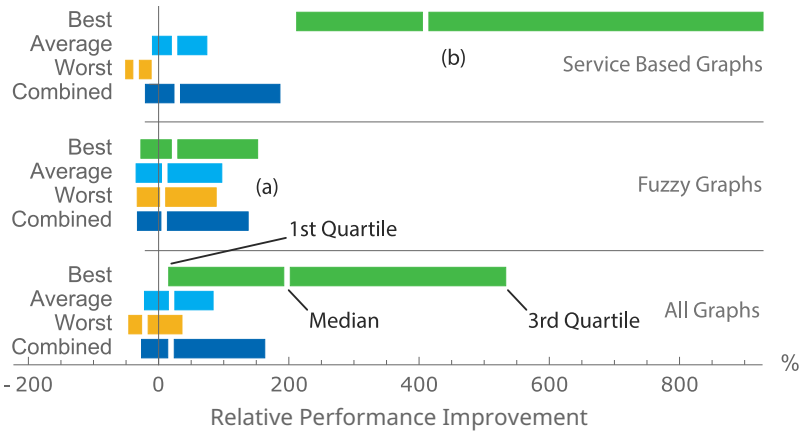


Figure 6.10: Relative performance improvements when using resource dependency processing over traditional processing.

tree is processed. The third implementation, the asynchronous resource dependency processing (ARDP) stalls the response only until the first level of the processing tree is finished processing and continues to process the rest asynchronously in the background.

6.8.1.2 Request Modes

Further, graphs are evaluated in two modes. In requester mode (RM) the requesting machine uses the generated traffic. In sequencer mode (SM) the generated traffic is bypassed and each resource of the graph is requested exactly once. This is done in order to measure the influence of the traffic on the performance.

6.8.1.3 Empirical Data and Modelled Data

The empirical data is collected by measuring the accumulated processing time for each request. The processing times for all three implementations are compared in two modes, where a lower accumulated processing time is better. Additionally, the models developed in Section 6.7 are used to calculate the accumulated processing times and the model fits are determined. In total over 850 thousand resources are evaluated with over 6 million requests in 2000 web services. To compare the performance, the relative performance improvements are calculated for service based graphs, fuzzy graphs and all graphs combined in the resource dependency processing and asynchronous resource dependency processing implementations both in requester and sequencer mode. Consequently, there exist $4 \cdot 2000$ $RPIs$, where a positive RPI denotes the factor of improvement the resource dependency processing approach exhibits over the traditional processing approach. Further, the results are evaluated in four cases, where each case individually limits the range of the read/processing ratio RPR and hit/miss ratio HMR.

6.8.1.4 Combined Case Results

In the combined case, the results of all aggregated graphs are presented. Figure 6.10 shows the quantiles of the relative performance improvement where the size of the boxes around the median divide the results in equal parts regarding the first quartile and the third quartile. As the minimum and maximum relative performance improvement factors have a high variance, the whiskers are intentionally left out in Figure 6.10 to be able to visualise quantiles in a meaningful way. In total, 59% of all combined case services are faster using resource dependency processing rather than traditional processing.

6.8.1.5 Best Case Results

From Figure 6.9 follows, that for all read/processing and hit/miss ratios between 0.0 and 0.3 the resource dependency processing is faster than the traditional processing. Consequently, for the best case aggregation both ratios are restricted to be within the 0.0-0.3 range. Figure 6.10 illustrates median performance improvements of a factor higher than four for the service based graphs, 25% for the fuzzy based graphs and a factor of almost two for the combination of both. In total, 79% of all best case services are faster using resource dependency processing rather than traditional processing.

6.8.1.6 Worst Case Results

The worst case aggregates results in the ranges where in Figure 6.9 the traditional processing is faster. Thus, the read/processing ratio is limited to the range between 0.6 and 1.0. The hit/miss ratio is limited to the range between 0.5 and 0.7 as this is the maximum evaluated hit/miss ratio (Table 6.5). The median performance improvements illustrated in Figure 6.10 are -34% for the service based graphs, 5.4% for the fuzzy based graphs and -21% for the combination of both. In total, 37% of all worst case services are faster using resource dependency processing rather than traditional processing.

6.8.1.7 Average Case Results

For the average case aggregation, results in the ranges where in Figure 6.9 both approaches exhibit similar performance are selected. Consequently, a read/processing ratio in the range between 0.3 and 0.6 and a hit/miss ratio in the range between 0.3 and 0.5 are used. Further, the selected ranges are confirmed to be within typical ranges as presented by Du and Wang (2015) and Songwattana, Theeramunkong and Vinh (2014). As shown in Figure 6.10, the median performance improvements are 25% for the service based graphs, 9.4% for the fuzzy based graphs and 20% for the combination of both. In total, 62% of all average case services are faster using resource dependency processing rather than traditional processing.

6.8.1.8 Model Fits

The processing duration delta Δd_p is calculated for all 8000 evaluations and compared to the empirical performance results. The residuals of all evaluations have a root-mean square error of $RMSE = 30.4$. Using the normalised $RMSE$ to put the errors in relation to the observed values (Section 6.5) the Fit is calculated as $1 - NRMSE$. The mean fit for the cluster size based model is $Fit_{CS} = 0.96$ and the mean fit for the dependency depth based model is $Fit_{ddep} = 0.98$. This implies that both duration delta models have very good fits. Further, it is observable that the cluster size based model is cheaper to compute while the dependency depth model is more accurate.

6.8.2 Structure Based Performance

The structure based evaluation is performed to analyse the effects of different graph structures on the performance. Therefore, four series of graphs with increasing graph measures are created and performance tested with the resource dependency processing and a traditional processing approach. As presented in Figure 6.9, the major performance influencing parameters are the read/processing ratio and the cache hit/miss ratio. In order to analyse the effects of the graph structures only, for all series both the read/processing ratio and the hit/miss ratio are set to their calculated performance break-even points $RPR_{BEP} = 0.58$ and $HMR_{BEP} = 0.44$. A series of five graphs is created for an increasing dependency depth in Figure 6.11 (a-e), dependency degree in Figure 6.11 (f-j), cluster size in Figure 6.11 (k-o) and number of clusters in Figure 6.11 (p-t). The processing delay for each resource is set to 0.2 seconds and for presentation issues, all graphs have a total of 50 vertices. Each series starts with a low value of the measure that increases in five steps to a maximum measure as detailed in Table 6.6. The graphs are created using a special version of the fuzzy dependency graph generator, where the desired graph measure (depth, degree, cluster size and count) is set instead of letting the algorithm choose a random value.

6.8.2.1 Performance Results

The performance is calculated using the processing duration delta model, where negative values indicate a better performance when using our proposed resource dependency processing approach. The results are presented in Figure 6.11 and Table 6.6. From all 20 graphs, the performance using the resource dependency approach presented in this thesis is better for 13 structures (a-b,f-h,k-n,q-t). The increasing depth in the depth series (a-e) has a major influence on the performance as the length of the chains as seen in (d-e) massively increases the processing duration. An increasing degree (f-j) has a minor influence on the performance as most dependencies can be processed in parallel. The growing mean cluster size series in

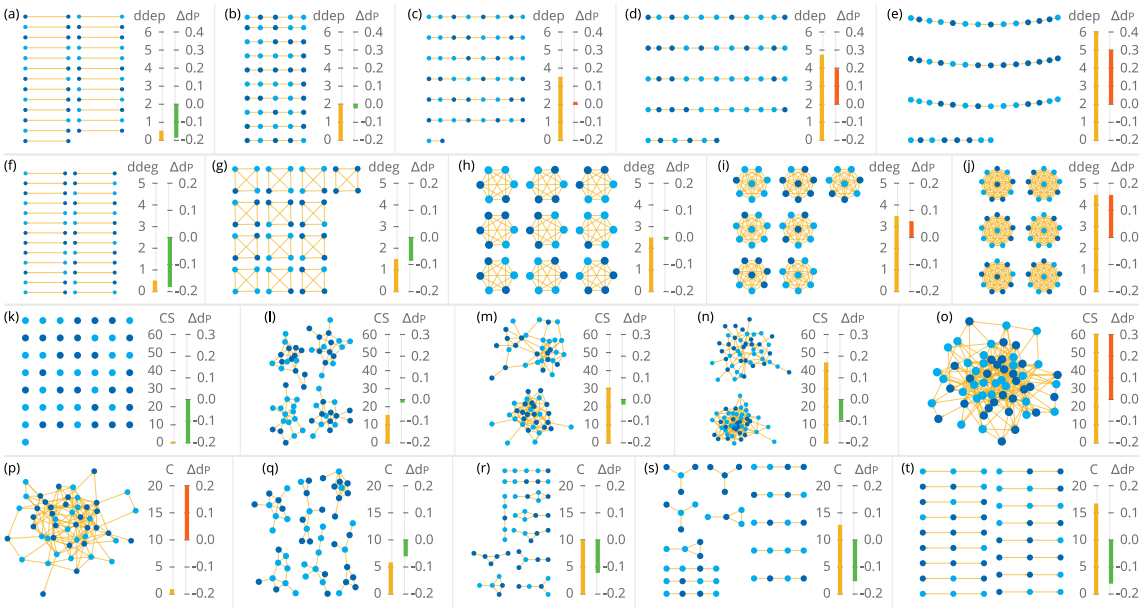


Figure 6.11: Structure based results of four series of increasing graph measures.

(k-o) has an effect on both the depth and the degree, where a lower cluster size results in lower maximal depths. Finally, the increasing total number of clusters series (p-t) is tightly inversely related to the mean cluster size. Thus, with many clusters the performance is better as the maximal depth is reduced.

6.8.2.2 Mapping to Real-World Structures

When searching for resemblance between graphs in Figure 6.6 and Figure 6.11, it is noteworthy that Figure 6.6 presents structure graphs, where Figure 6.11 presents full resource graphs (for distinction see Table 6.1). To extract resource graphs from structure graphs as presented in Figure 6.6, variables for the number of users, posts, comments, photos etc. majorly influence the measures of the resulting resource graph. Thus, it is suggested that similarities must be compared with care. Facebook's rather centralised structure can be extracted to graphs similar to Figure 6.11 (l-m) as the centralised structure is based around clusters of users. A single user in Instagram, Google Plus or Tumblr has a smaller cluster of resources leading to structures more similar to Figure 6.11 (g,h,s). This leads to an increased processing duration with Facebook's structure compared to more decentralised structures such as Twitter's, Instagram's or Tumblr's. However, it is suggested that this stems from the massively higher range of functions Facebook offers compared to the other platforms.

6.9 Discussion

In this chapter, an efficient resource database update mechanism that allows to build scalable web service architectures with optimised request routing has been presented. With respect to **R6.1** *How can resource dependencies be measured and stored?*,

Table 6.6: Structure based performance results for Figure 6.11.

Measure	**Param**	**Dependency Depth Series** (a-e)				
Dependency Depth	*ddep*	0.5	2.	3.4	4.7	6.
Dependency Degree	*ddeg*	0.5	0.8	0.86	0.9	0.92
Cluster Size	CS	2.	5.	7.1	10.	13.
Number of Clusters	C	25	10	7	5	4
Processing Delta	Δd_p	-0.16	-0.034	0.081	0.19	0.3
Measure	**Param**	**Dependency Degree Series** (f-j)				
Dependency Depth	*ddep*	0.5	1.5	2.5	3.5	4.5
Dependency Degree	*ddeg*	0.5	1.5	2.5	3.5	4.5
Cluster Size	CS	2.	4.	6.	8.	10.
Number of Clusters	C	26	13	9	7	6
Processing Delta	Δd_p	-0.17	-0.083	-0.0064	0.071	0.14
Measure	**Param**	**Cluster Size Series** (k-o)				
Dependency Depth	*ddep*	0.	2.6	2.9	3.	6.4
Dependency Degree	*ddeg*	0.	1.9	2.6	2.7	4.2
Cluster Size	CS	1.	16.	31.	46.	61.
Number of Clusters	C	50	4	2	2	1
Processing Delta	Δd_p	-0.2	-0.035	-0.0044	-0.099	0.3
Measure	**Param**	**Number of Clusters Series** (p-t)				
Dependency Depth	*ddep*	4.8	1.6	1.	0.81	0.55
Dependency Degree	*ddeg*	3.2	1.4	0.96	0.79	0.67
Cluster Size	CS	50.	9.	5.	4.	3.
Number of Clusters	C	1	6	10	13	17
Processing Delta	Δd_p	0.2	-0.078	-0.12	-0.14	-0.16

it was shown that resource dependencies can be stored as directed acyclic graphs, where vertices represent resources and edges dependencies. Further, the dependency depth, dependency degree, cluster count, cluster size and sparsity were identified as the most influential graph measures. With respect to **R6.2** *What algorithm can be used to optimise the performance of processing?*, it was found that applying shortest-path algorithms for dependency processing is not suitable as the processing order needs to be maintained. Hence a longest-path algorithm combining a topological sort with dynamic programming which determines the processing order in linear time was developed. With respect to **R6.3** *What effects have dependency graph measures on the performance?*, it was further found that the dependency depth and cluster size have a linear correlation with the processing duration, where the accuracy of regressions based on both measures depends on the sparsity of the graph. With respect to **R6.4** *How can resource dependencies be generated?*, the generation of random dependency graphs was based on service structures where the parameters were extracted from six real-world social applications and a fuzzy algorithm with random parameters. With respect to **R6.5** *How well can the dependency processing duration be modelled?*, the processing duration was found to be adequately modelled based on the cluster size and the dependency depth. This allows to replace

the constant resource dependency processing delay from Chapter 5 with an exact model. The cluster size based model has an overall model fit of 96% and is cheap to compute, where the dependency depth based model has a model fit of 98%, thus being more accurate but also more expensive to determine. The duration delta and relative performance improvement enables to compare the performance of a traditional processing approach versus a resource dependency approach. Finally, with respect to **R6.6** *How is the performance compared with a typical traditional processing approach?*, the evaluation of 2000 web services with 850 thousand resources and over 6 million requests showed the combined synchronous and asynchronous resource dependency processing approach to be up to a factor of two faster than a traditional processing approach.

6.10 Summary

In conclusion, it was shown that the PRSM pattern that depends on the fast processing of dependencies can be implemented with algorithms that are able to process dependencies in linear time. The processing performance directly depends on the structure of the application. Long dependency paths negatively influence the performance as they must not be processed in parallel but in a correct sequential order. The analysis of the existing applications however showed, that even complex resource graph structures such as those from Facebook or SoundCloud exhibit a maximal path depth of 4. When introducing a detailed resource graph structure analysis into the development process, it can be expected that the resource dependency depth can further be reduced to provide an optimal basis for resource dependency processing. Consequently in the following Chapter 7, a portable and interoperable prototype of a WSF and an application created with a WAF is designed and evaluated to deliver optimal dependency processing performance.

7. Cloud Portable and Interoperable Prototype Implementation and Evaluation

7.1 Overview

In this chapter, a cloud portable and interoperable prototype implementation of a WSF is proposed. It is shown how existing web applications can be integrated into a WSF to enhance the scalability and portability. Further, a model to calculate and compare the processing cost and storage space of resources is developed. The model is evaluated with a traditional processing approach and a dependency processing implementation that is integrated into a WSF. Finally, the results of the traditional and the dependency processing approach are compared to analyse the processing cost and storage requirements. The literature review in Chapter 3 identified four open research questions related to the cloud portable and interoperable implementation and evaluation of a prototype:

- **R7.1**: How can modules and components be designed in a portable and interoperable fashion?
- **R7.2**: What needs to be done in order to integrate a traditional app into a WSF?
- **R7.3**: How well can the processing cost and storage space required for dependency processing be modelled?
- **R7.4**: How is the performance trade-off between processing cost and processing duration when using a dependency processing approach?

The remainder of the chapter is organised as follows: Section 7.3 examines the prototypical implementation of a cloud portable and interoperable WSF. Section 7.4 integrates and extends an existing web application to support operation by a WSF. Section 7.5 develops models to analyse the processing cost and storage size required for dependency processing, while Section 7.6 evaluates both the performance and model fits with the evaluation cluster. Section 7.7 presents the results and provides answers to the aforementioned research questions. Finally, Section 7.8 concludes the chapter with a summary of the presented work.

7.2 Motivations and Objectives

The conceptual architecture presented in Chapter 4 proposes an abstract framework specification of a WSF. The modules, interfaces, parameters and components are

designed to be portable across different cloud providers and interoperable to operate with multiple cloud providers at the same time. In order to illustrate and validate the conceptual architecture, in this chapter a prototypical implementation of a WSF is explored. The prototype is implemented and deployed to the Google Cloud Platform (2008), Amazon Web Services (2006) and the *Pi-One* evaluation cluster presented in Section 2.8. In order to integrate a web application into a WSF, the web application needs to be adapted. Consequently, in this chapter a social web application is designed, implemented and evaluated for the necessary adaptations required to operate it with a WSF. Cloud provider service charges are typically based on multiple metrics. One metric is the number of machines that are utilised. In this thesis, this metric is modelled and evaluated in Chapter 5, where the request flow optimisation scheme reduces the number of provisioned machines. Another metric is the duration of time a service is used. In this thesis, this metric is modelled and evaluated in Chapter 6, where the resource dependency processing algorithms minimise the time spent for processing. Further metrics used to charge cloud customers are the total amount of processing and the storage space occupied by a web service. Both the amount of processing and the storage space are determined by the resource dependency graph and corresponding optimisation algorithms. Consequently, in this chapter a model for both the processing cost and required storage space is developed and evaluated with the prototypical implementation.

7.3 Prototypical Implementation

In Chapter 4, the conceptual design specifies required modules, interfaces and components of a WSF. In order to operate modules and components on multiple cloud providers, they have to be designed to run on any cloud platform and allow simple migration between multiple platforms. This opens up research question **R7.1**: *How can modules and components be designed in a portable and interoperable fashion?* Consequently, this section presents a prototypical implementation of WSF modules and components operated by multiple cloud providers.

7.3.1 Cloud Providers with Linux Container Support

As presented in Section 3.5, Linux containers enable a portable and efficient deployment of modules and components. A Linux container consists of an operating system and fully customisable software that runs inside the container. Multiple containers can be linked with distinct networking interfaces and data can be shared using volume containers that can serve as data storage.

7.3.1.1 Docker Container Engine

The Linux container implementation used in this thesis is Docker (2013) as it is supported by all major cloud providers. Generally, a Docker container is described

by a *Dockerfile*. The *Dockerfile* contains information on how to build the container, such as:

- The operating system the container should use
- The command to clone the repository with the source code of the application
- Commands that should be run to install application dependencies
- Environment variables in the container
- Ports the container should expose
- Volumes that should be mounted
- The entry point that is run when the container is executed

An exemplary *Dockerfile* that creates a portable app can look like this:

```
FROM ubuntu
RUN apt-get update && apt-get install -y go

# Install App
RUN mkdir /app
RUN git clone http://github.com/wsf/app /app
WORKDIR /app
RUN go install

# Expose Port
EXPOSE 3000

# Run App
CMD ["app", "--port=3000"]
```

With such a *Dockerfile*, a container engine can build and run independent components on various hosting environments. All major cloud providers offer container services supporting Docker. In the following sections, a short overview of the Amazon EC2 Container Service (2014), Google Container Engine (2014) and IBM Containers for Bluemix (2014) APIs is presented.

7.3.1.2 Amazon Elastic Container Service (ECS)

The Amazon ECS API uses clusters, services and tasks. A cluster is a logical grouping of machines that can host a service. A service consists of at least one instance of a task. A task is defined by a task definition. A task definition defines the container setup such as the used images that are built by the *Dockerfile*, environment variables and ports. An exemplary task definition file with two containers is provided as follows:

```
{
  "containerDefinitions": [
    {
      "name": "worker",
      "links": ["queue"],
      "image": "worker",
      "essential": true,
      "portMappings": [{
        "containerPort": 80,
        "hostPort": 80
      }],
      "memory": 500,
      "cpu": 10
   },
   {
      "environment": [{
        "name": "Q_PASSWORD",
        "value": "secret"
      }],
      "name": "queue",
      "image": "queue",
      "cpu": 10,
      "memory": 500,
      "essential": true
   }
  ],
  "family": "hello_wsf"
}
```

7.3.1.3 Google Container Engine

The Google Container Engine API uses container clusters, services, jobs and pods. A container cluster provides machines to host a service or run a job. Services and jobs combine one up to many pods. A pod defines the container setup such as the used images, environment variables and ports. An exemplary pod with two containers is provided as follows:

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "worker",
```

```
    "labels": {
      "name": "worker"
    },
  },
  "spec": {
    "containers": [
    {
      "name": "worker",
      "image": "worker",
      "ports": [{
        "containerPort": 80,
      }],
      "resources": {
        "cpu": "10"
        "memory": "500"
      }
    },
    {
      "name": "queue",
      "image": "queue",
      "env": [{
        "name": "Q_PASSWORD",
        "value": "secret"
      }],
      "resources": {
        "cpu": "10"
        "memory": "500"
      }
    }
    ],
  }
}
```

7.3.1.4 IBM Bluemix Containers

The IBM Bluemix Containers API provides compute nodes, services, container groups and containers. Compute nodes run services that consist of container groups. A container group includes one up to many containers that are built from *Dockerfiles*. Notably all presented cloud providers share a common hosting structure where containers are composed into groups that can be used by services that run on clusters of machines. Unfortunately, the cloud providers however do not share a common

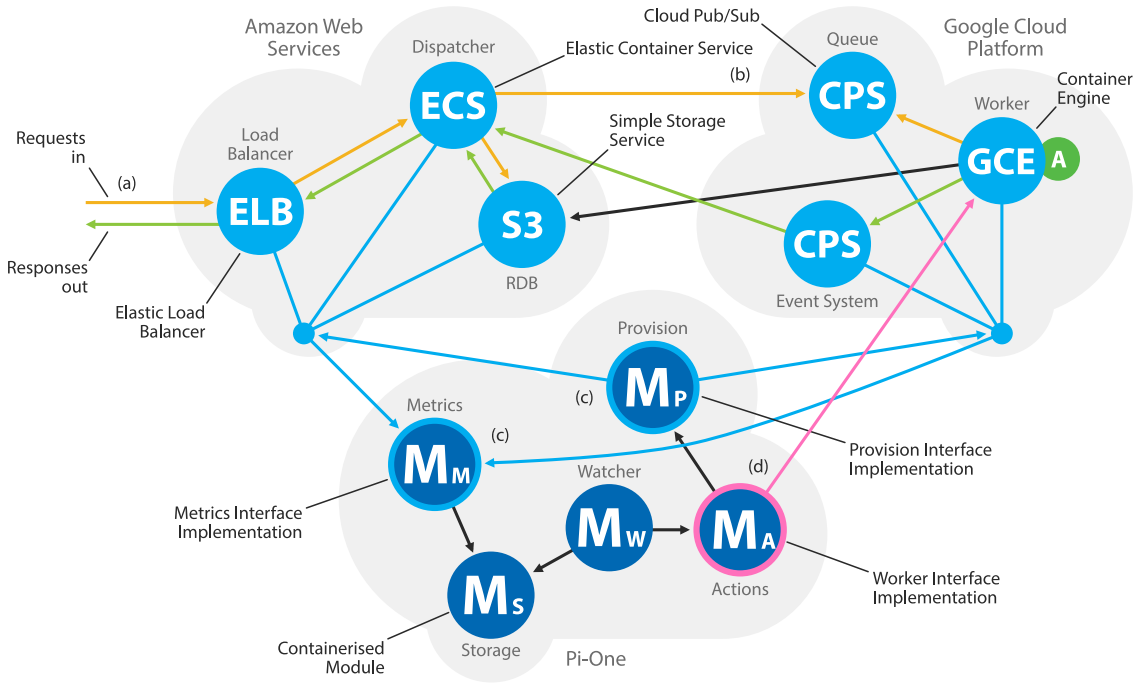


Figure 7.1: Portable and interoperable prototypical implementation of components and modules.

standard to express this hosting structure. Hence, the provisioning module of a WSF needs to translate actions for each cloud provider.

7.3.2 Prototype Components

As illustrated in Figure 7.1, the prototypical implementation uses components deployed to the *Pi-One* cluster, Amazon Web Services (2006) and Google Cloud Platform (2008). At Figure 7.1 (a), the load balancer uses the Elastic Load Balancer (ELB) SaaS from Amazon Web Services (2006). The ELB API allows a user to register and remove target components. Additionally, it allows setting different balancing policies, where for the prototype, a round robin selection of dispatchers is chosen. The dispatcher is deployed as a container that runs on the Amazon Web Services (2006) Elastic Container Service (ECS). It is implemented using Go and divides the read from the processing requests. The read requests are looked up in the resource database, and the processing requests are put into the queue. The resource database uses the Amazon Web Services (2006) Simple Storage Service (S3) to store and retrieve data. At Figure 7.1 (b), both the queue and the event system are deployed using Google Cloud Platform (2008) Cloud Pub/Sub (CPS) message middleware. The Cloud Pub/Sub service allows both push and pull delivery. The dispatcher pushes requests into a queue where the worker pulls them for processing. The worker is deployed as a container in the Google Cloud Platform (2008) Container Engine. It is implemented using Go and incorporates an application that is presented in the subsequent section of this chapter. When the worker has finished processing, it pushes the response to the Cloud Pub/Sub system where the

dispatcher pulls it for delivery to the client.

7.3.3 Prototype Modules

All modules of the prototypical implementation are deployed as containers to the *Pi-One* cluster. As shown in Figure 7.1 (c), both the provision and metrics module implement connectors to the Amazon Web Services (2006) Elastic Load Balancer, Elastic Container Service and Simple Storage Service. Additionally, they implement connectors to the Google Cloud Platform (2008) Cloud Pub/Sub and Container Engine. Both modules implement the corresponding provision and metrics interfaces by translating abstract actions to cloud provider specific actions. The storage module is implemented as a simple key-value database using the Redis (2009) data structure server. It stores component metrics and framework parameters and provides them to other modules. The watcher module is implemented in Go to frequently request metrics from the storage module. It implements the component and composition models to calculate the optimal machine configuration and triggers the actions module for provisioning when required. As shown in Figure 7.1 (d), the actions module is created in Go and implements the action interface by providing available actions to provision components and deploy applications. It is noted that the resilience and interface modules are not implemented for the prototype to reduce complexity.

7.4 Web Application Integration

A WSF operates web applications generated with traditional WAFs. In order to cooperate with a WSF, however, web applications have to be adapted. This opens up research question **R7.2**: *What needs to be done in order to integrate a traditional app into a WSF?* Consequently, in this section a traditional web application is developed and integrated into the prototype.

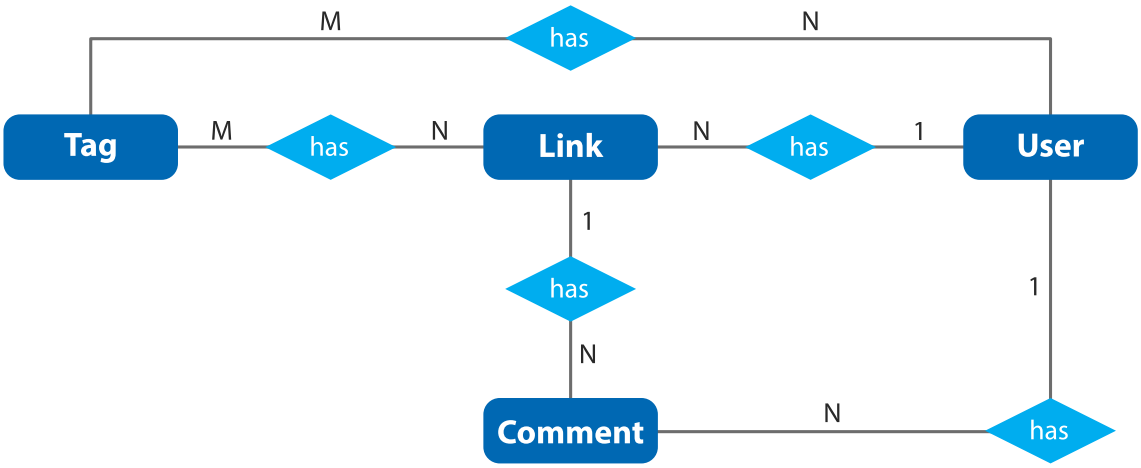


Figure 7.2: Entity relationship model of the LinkR web application.

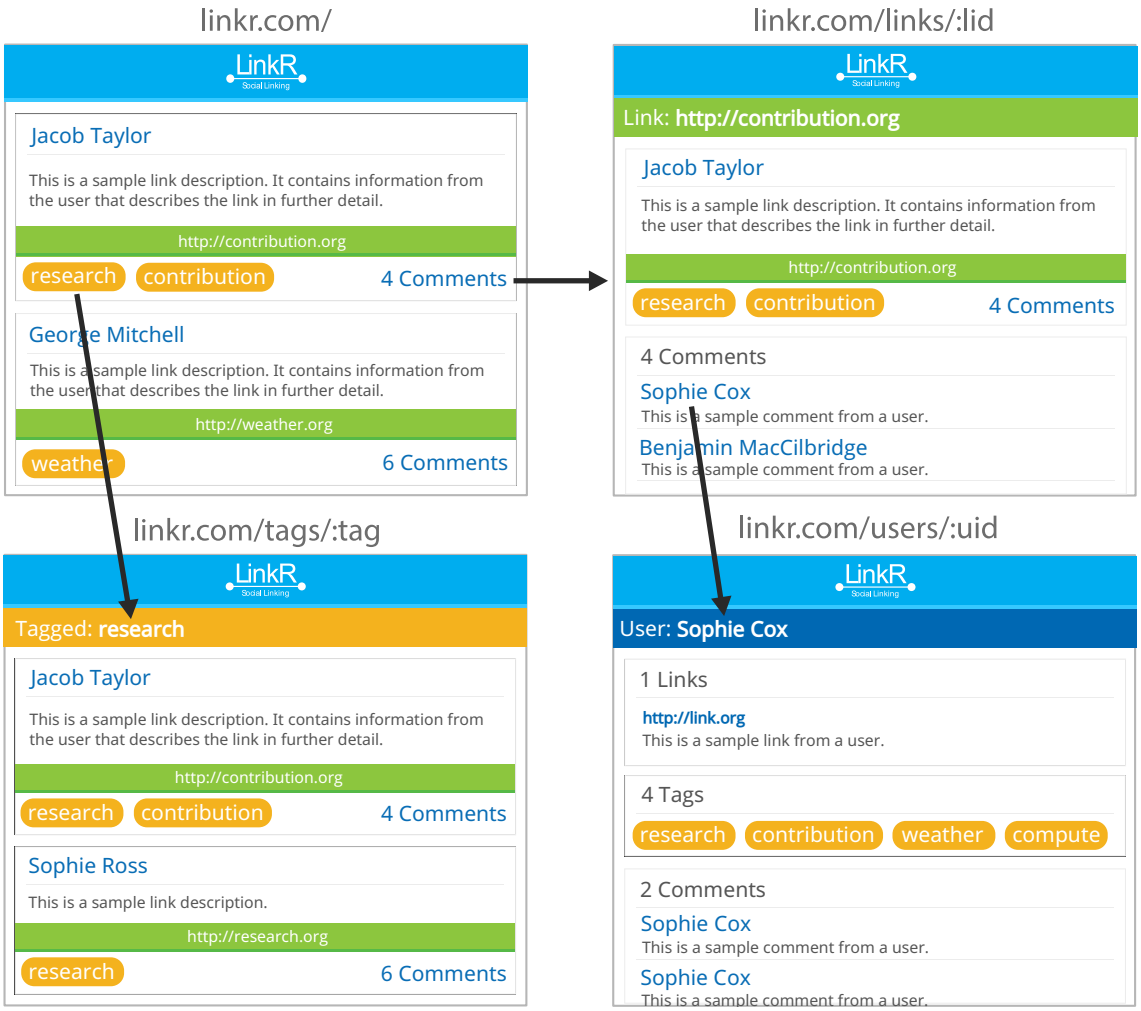


Figure 7.3: Screenshots from four LinkR web application service views with annotated dependencies.

7.4.1 LinkR Web Application

The showcase web application developed for the prototype is named *LinkR*. It is designed to demonstrate an application with two major functionalities: social connections between resources and organisation and search of content. The social connections property leads to a high content spread across multiple resources. The organisational properties on the other hand categorises and bundles resources into other resources. *LinkR* resembles a social web application that allows sharing links with other users. For the social property, users can comment on links and have a profile page that lists all of their contributions. For the organisational property, links can be tagged and therefore are searchable based on their tags. A link can have many comments from multiple users and a user can have tags through her links. Figure 7.2 shows the entity relationship model of the web application with the entities link, comment, user and tag. The web application is implemented using Go. It serves CSS, image and HTML files dynamically rendered for each requests to follow the traditional scheme of web applications. Figure 7.3 shows screenshots from four of five application views available, where the view that displays a single

comment only is omitted due to space limitations. The dynamic parts in the URIs are denoted by the link id *:lid*, the user id *:uid* and the tag *:tag*. The omitted comment view can be accessed through the */links/:lid/comments/:cid* URI from each link view.

7.4.2 Adaptations for Integration into a WSF

As presented, the *LinkR* web application can be deployed in the traditional scheme (Chapter 5). Often, the traditional scheme incorporates a partial caching solution. For the partial caching solution to work, a caching policy has to be implemented that pushes resources to the cache and invalidates them when the data behind the resources change. In order to implement the invalidations, typically manual triggers that observe data are added. With complex data structures, it can become hard to manually implement the invalidations correctly. In order to integrate a web application into a WSF, three automatable adaptation steps are required:

1. Analysis of the service structure with automatic dependency extraction
2. Injection of dynamic dependency declarations and resource push
3. Resource Index generation for initial resource database fill up

Each step is presented in the subsequent sections.

Table 7.1: LinkR web application routes and dependencies.

Read Routes	**Dependencies**
GET / GET /links/:lid GET /links/:lid/comments/:cid GET /users/:uid GET /tags/:tag	
Processing Routes	**Dependencies**
POST /links	GET / GET /links/:lid GET /users/:uid GET /tags/:tag
POST /links/:lid/comments	GET / GET /links/:lid/comments/:cid GET /links/:lid, GET /users/:uid
DELETE /links/:lid	GET / GET /users/:uid
DELETE /links/:lid/comments/:cid	GET / GET /links/:lid GET /users/:uid

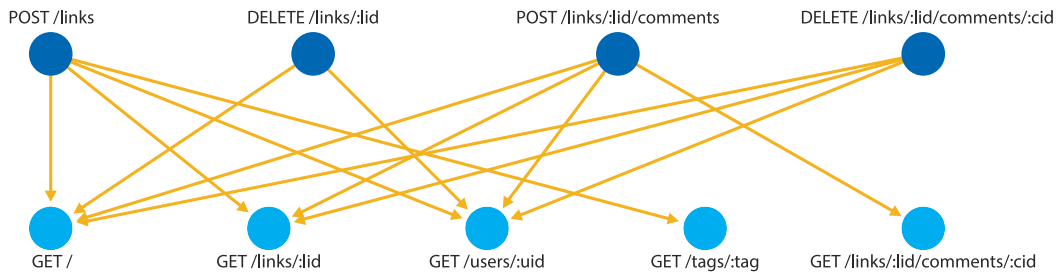


Figure 7.4: Service structure graph for the LinkR web application with resource nodes and dependency edges.

7.4.3 Service Structure Graph Analysis

As a first step, the service structure of the *LinkR* application is analysed for its resources and dependencies. In Section 6.6, service structure graphs are extracted from six social network services based on their APIs. The service structure graph is created by inspecting all available application resources and their dependencies on other resources. A dependency between two resources exists, if the processing of one resource changes the output of the other resource. A route uniquely identifies a resource and is defined by the HTTP method and a path. Each route can have a list of dependencies that needs to be processed when the route is processed. Table 7.1 shows the routes and dependencies for the *LinkR* web application. The dependencies can be deduced from the entity relationship model in Figure 7.2 and the screens in Figure 7.3. A link is listed in the index page and a custom link page. Additionally, it is represented in the user page and all tag pages. A comment is listed in the index page, the custom link page and a custom comment page. Additionally, a user page lists all comments the user created. By following all dependencies, a service structure graph such as that shown in Figure 7.4 can be generated that visualises all routes and dependencies. For bigger applications, the service structure graph can be extracted automatically from the data and render logic of the application. The view layer of the application knows exactly which data it accesses. From that information, other views that use the same data can be found as dependencies.

7.4.4 Dependency Graph Injection and Resource Push

The dependency graph can be extracted from the service structure graph. Static dependencies are dependencies where there is no variable part in the request path. For *LinkR* the only static dependency is the index page *GET /*. In contrast, dynamic dependencies are dependencies with variable parts in the request path. For *LinkR* all dependencies except *GET /* are dynamic. During runtime, the worker needs to resolve all dependencies for a request. To resolve the dynamic dependencies, the worker needs to know with which variables the dependencies need to be expanded. For example the *POST /links* route has a dynamic dependency on *GET /tags/:tag*. In order to update the correct tag pages, the worker needs to know which tags the posted link contains. This information needs to be injected into the *LinkR*

web application. The following pseudocode illustrates the *POST /links* action with dynamic dependency expansion:

```
1: params ← request with link parameters in body
2: link ← createLink(params)
3: worker.addDependency("GET/links/" + link.id)
4: worker.addDependency("GET/user/" + link.user.id)
5: for all tag t in link.tags do
6:     worker.addDependency("GET/tag/" + t)
7: end for
8: return [201, renderLink(link)]
```

On the first two lines the link is created from the request params and stored to the *LinkR* database. Lines 3-4 expand dependencies to the link itself and the user that created the link. On lines 5-6, the tag dependency is expanded for every tag. The dependency to *GET /* is stored as static dependency and therefore must not be expanded by the application. Finally, on line 8 the response is rendered and send with the HTTP status code 201 (created). In order to update the resources in the database, the updates need to be pushed when they are finished rendering. Typically, there is a one-to-one mapping of read routes and resource database keys, and thus the resource *GET /links/abc* is stored in the resource database under key */links/abc*. This allows an automatic mapping of all read routes to database keys and therefore must not be integrated into the *LinkR* web application.

7.4.5 Resource Index Generation

Finally, in order to fill the resource database on startup, all requestable resources need to be pushed to the database once. This can be easily done by having the worker to request all available read routes that can be derived from the entities. The following pseudocode illustrates this initial indexing process:

```
1: index ← [] start with an empty array
2: index.add("GET/")
3: for all link l in database.links do
4:     index.add("GET/links/" + l.id)
5:     for all comment c in l.comments do
6:         index.add("GET/links/" + l.id + "/comments/" + c.id)
7:     end for
8: end for
9: for all user u in database.users do
10:    index.add("GET/users/" + u.id)
11: end for
12: for all tag t in database.tags do
13:    index.add("GET/tags/" + t)
```

14: **end for**

15: **return** index

On the first two lines an empty index is generated and the static routes are added. Each read route is then expanded with the entities from the database and added to the index on lines 3-14. The index returned on line 15 is then put into the queue component where multiple workers can fill up the resource database. This does not require any extra implementation as the processing subsystem processes requests from the queue by default.

7.5 Processing Cost and Storage Space Modelling

As mentioned in the introduction of this chapter, cloud providers charge customers based on the number of machines, usage duration, processing cost and storage space. The models in Chapter 5 and Chapter 6 consider both the number of machines and usage duration of a dependency processing approach. This opens up research question **R7.3**: *How well can the processing cost and storage space required for dependency processing be modelled?* Consequently, in the subsequent section models for the processing cost and required storage space for the traditional and dependency processing approach are developed.

7.5.1 Processing Cost

The processing of a single request involves work by the CPU, system memory and input/output system. For cloud services, however, these metrics are often abstracted away from the user. Thus, the processing cost is abstractly expressed as the number of dependencies that need to be processed for each incoming request. For the traditional approach TP, each incoming request needs to be processed that is not in the cache. Thus it can be modelled as:

$$PC_{TP} = 1 \cdot (1 - HMR) \tag{7.1}$$

For the dependency processing approach, only processing requests need to be processed. To determine the number of updates, the mean number of dependencies needs to be calculated. This can be done with the help of the service structure graph where the degree for each processing node is counted. Dynamic dependencies are counted with their mean quantity of the expanded form. For example in the *LinkR* web application, a link has an average of 3 tags, and thus 3 needs to be added to the degree instead of 1. With the mean dependency degree *ddeg* the processing cost can be modelled as:

$$PC_{DP} = (1 - RPR) \cdot (1 + ddeg) \tag{7.2}$$

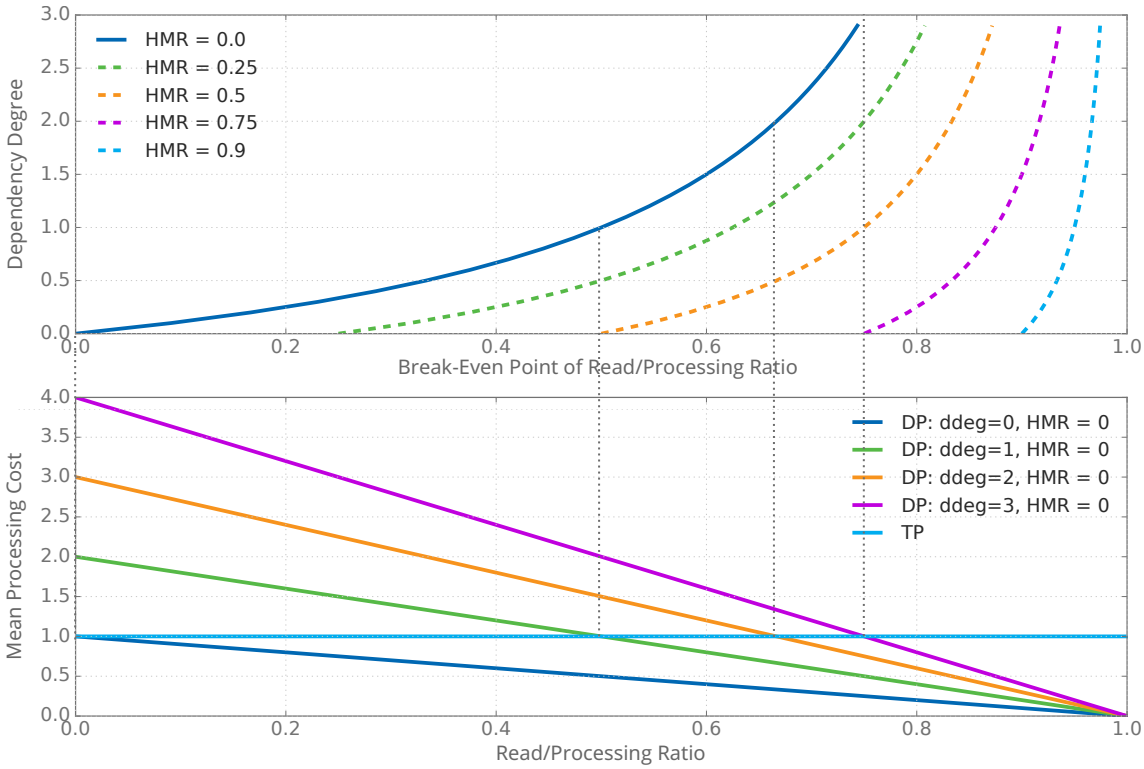


Figure 7.5: Mean processing cost and break-even points for multiple hit/miss ratios.

7.5.2 Break-Even Point for Processing Cost

From Equation (7.1) and Equation (7.2) the break-even point where both approaches have equal processing cost can be calculated. This can be done by equalising PC_{TP} with PC_{DP} and solving it for RPR as follows:

$$\begin{aligned} RPR_{BEP} &= (PC_{TP} = PC_{DP}), \quad \text{solve for } RPR \\ &= \frac{HMR + ddeg}{ddeg + 1} \end{aligned} \tag{7.3}$$

Figure 7.5 shows both the mean processing cost for the TP and the DP approach in the lower subgraph. The corresponding break-even points are where the TP and DP lines are crossing. At these crossings, both approaches have the same processing cost. The upper subgraph in Figure 7.5 shows the RPR_{BEP}s for multiple dependency degrees $ddeg$.

7.5.3 Storage Space

The calculation of the required storage space is based on the total number and resource size of each entity. From the entity model in Figure 7.2, the following set of entities E is extracted and assigned to the following variables:

$$E : \{Link \rightarrow l, Tag \rightarrow t, User \rightarrow u, Comment \rightarrow c\} \tag{7.4}$$

For each entity in the set, the quantity of existing instances is expressed as q_x:

$$Q : \{q_x \mid x \in E\} \tag{7.5}$$

A $q_l = 1000$ and $q_u = 300$ means that an application has 1000 links and 300 users. Each entity is typically expressed as a fragment of similar contents. As presented in Figure 7.3, a user is described using markup language for his name, where a comment contains a username, date and the comment text. A link contains the URI, the username and some description text. For modelling, each of the fragment sizes is measured and extracted as f_x:

$$F : \{f_x \mid x \in E\} \tag{7.6}$$

A $f_l = 1000$ and $f_u = 400$ means that the fragment size for a link in a resource is 1000 bytes and the fragment size of a user is 400 bytes. In addition to the fragments, a base size b is defined that measures the size of a resource containing only the skeleton of the resource, such as the HTML headers. The total size needed to store all resources can then be calculated by defining a system of equations for all read resources. The read resources can be extracted from the service structure graph as R:

$$\begin{aligned} R : \{ & \\ & / \rightarrow ls, \\ & /links/ : id \rightarrow l, \\ & /tags/ : id \rightarrow t, \\ & /users/ : id \rightarrow u, \\ & /links/ : id/comments/ : id \rightarrow c \\ \} & \end{aligned} \tag{7.7}$$

For each resource in R, the mean resource size s_x can now be modelled with respect to the contents of each resource:

$$s_{ls} = b + q_l \cdot f_l \tag{7.8}$$

$$s_l = b + f_l + (\frac{q_c}{q_l} \cdot f_c) \tag{7.9}$$

$$s_t = b + (\frac{q_l}{q_t} \cdot f_l) \tag{7.10}$$

$$s_u = b + (\frac{q_l}{q_u} \cdot f_l) + (\frac{q_t}{q_u} \cdot f_t) + (\frac{q_c}{q_u} \cdot f_c) \tag{7.11}$$

$$s_c = b + q_c \cdot f_c \tag{7.12}$$

The modelled size equations count the number of fragments that appear in each resource. For example in Equation (7.11), the size of the user resource is calculated. The variable b represents the basic size of the resource. To that, the link fragment size f_l multiplied by the mean number of links q_l per user q_u is added. This step is repeated for the tags per user and comments per user. Finally, the storage size S can be calculated by summing up all mean resource sizes s_x multiplied by the quantity of the resource q_x:

$$S = \sum_{x \in R} q_x \cdot s_x \tag{7.13}$$

The size S can change over time as it is based on the quantities and fragment size of resources. During development of a web application, it can be helpful to track the mean resource sizes when migrating an application. The size model enables to continuously calculate the total storage space impact of fragment size changes in a single resource.

7.6 Processing Cost and Storage Space Evaluation

The analytical modelling of the processing cost opens up research question **R7.4**: *How is the performance trade-off between processing cost and processing duration when using a dependency processing approach?* Consequently, in this section the prototypical implementation is empirically evaluated to measure the performance differences and find the fit of the models developed in the previous section.

7.6.1 Evaluation Data

For the evaluation, application resource and traffic data are generated. The key figures for the generated evaluation data are shown in Table 7.2. For the application, a total of 600,000 requestable resources are generated and distributed according to the listed quantities. The fragment sizes are extracted from the delivered HTML markup shown in Figure 7.3. A total of 9 traffic traces with 1000 requests per trace are generated where each trace has a mean read/processing ratio of 0.1 up to 0.9. Each read request in the trace randomly selects one of the 5 read routes shown in Table 7.1. Each processing request randomly selects one of the 4 processing routes shown in Table 7.1. The processing actions *POST /links* and *POST /links/:lid/comments* create new links and comments that are processed by and added to the system. The processing actions *DELETE /links/:lid* and *DELETE /links/:lid/comments/:cid* select a random link and delete it or one comment of it.

7.6.2 Results

All traffic traces are run on the prototypical implementation where between all runs the application's resources data are reset by deleting the application and resource

Table 7.2: Key figures of the generated application resources and traffic traces to evaluate the prototypical implementation.

Resource Based	**Parameter**	**Value**
Link Quantity	q_l	100,000
Comment Quantity	q_c	500,000
User Quantity	q_u	50,000
Tag Quantity	q_t	10,000
Total Resources	$\|R\|$	660,000
Mean Base Fragment Size	b	1.2 kB
Mean Link Fragment Size	f_l	0.9 kB
Mean Comment Fragment Size	f_c	0.3 kB
Mean Tag Fragment Size	f_t	0.05 kB
Mean User Fragment Size	f_u	0.25 kB
Traffic Based	**Parameter**	**Value**
Number of Traces	$\|T\|$	9
Number of Requests per Trace	N	1,000
Read/Processing Ratios	RPRs	0.1-0.9
Mean Dependency Degree	$ddeg$	1.486

databases. For each request, the amount of required processing, the processing duration and the current storage space are recorded. Figure 7.6 presents a series of graphs for each traffic trace with increasing RPRs. The processing cost subgraphs present the results for the traditional approach and the dependency processing approach along with the modelled results. It is noticed that starting from a $RPR = 0.1$ up to a $RPR \approx 0.6$, the traditional approach has lower processing cost than the dependency processing approach. This stems from the fact that with a low portion of read requests, a high number of dependencies needs to be processed. Based on the processing cost break-even model, the exact break-even point where both approaches exhibit equal processing cost is $RPR_{BEP} = 0.597$ with a $ddeg = 1.486$. The second subgraph shows that all traces exhibit a faster processing performance for the full spectrum of RPRs. The third subgraph shows the storage space required throughout the evaluation with the corresponding storage space model results. The storage space randomly increases and decreases based on the processing actions that add or remove content. In order to determine the model fit, the results are calculated with the processing cost and storage space models developed in the previous section. For all traces, the combined root-mean-square error for the processing cost is $RMSE = 18.24$ and the storage size is $RMSE = 0.1$. Thus, the processing cost model fit is $Fit = 0.97$ and the storage size fit is $Fit = 0.99$. This supports the validity of both models. Overall, Figure 7.6 shows the trade-offs between reduced processing durations and increased processing cost and storage space requirements. Figure 7.7 presents an overview of all trade-offs in one figure. Starting at an $RPR = 0.1$, the dependency processing approach needs 60% more processing than the traditional approach, however, is 10% faster. At Figure 7.7 (a), both approaches

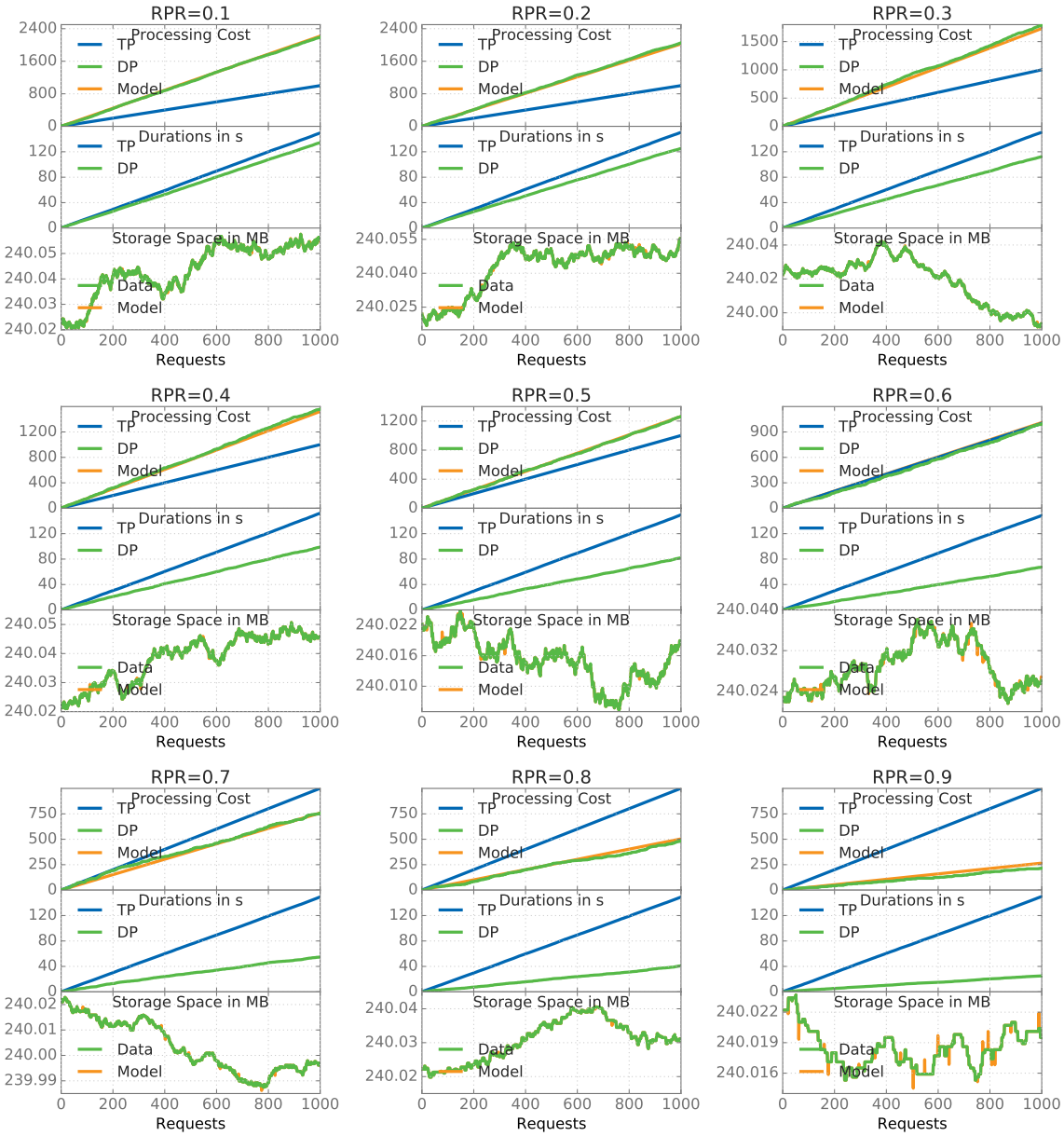


Figure 7.6: Trade-off graphs for a series of read/processing ratios (lower is better).

exhibit the same processing cost when the $RPR = 0.6$, although the dependency processing approach is 60% faster. For a $RPR = 0.9$, the dependency processing approach both has 35% lower processing cost and is 80% faster than the traditional processing approach.

7.7 Discussion

In this chapter, a cloud portable and interoperable prototype implementation of a WSF, an analytical model to determine the processing cost and storage space and an evaluation of the model have been presented. With respect to **R7.1** *How can modules and components be designed in a portable and interoperable fashion?*, it has been shown how components can be operated and moved between multiple cloud providers by utilising Linux containers and provider specific SaaSs. With respect to **R7.2**

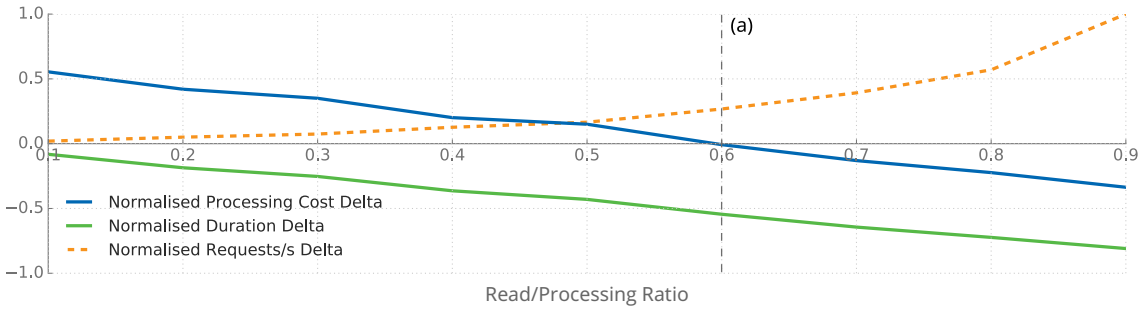


Figure 7.7: Normalised processing cost, duration and requests/s for the evaluated prototype.

What needs to be done in order to integrate a traditional app into a WSF?, three steps have been presented where in the first step the service structure was analysed to extract the dependencies, in the second step the dynamic dependencies were injected into the web application, and in the third step the resource index was created for the initial resource database fill up. With respect to **R7.3** *How well can the processing cost and storage space required for dependency processing be modelled?*, a processing cost and storage space model based on mean dependency degrees, entity quantities and resource fragment sizes has been developed. Both models fit the evaluation data by 97% and 99% for the processing cost and storage space models. With respect to **R7.4** *How is the performance trade-off between processing cost and processing duration when using a dependency processing approach?*, it has been presented that in optimal cases the dependency processing approach both has 35% lower processing cost while being 80% faster than the traditional processing approach. When both approaches exhibit equal processing cost, the proposed dependency approach was 60% faster than the traditional processing approach. A high RPR is better suited for the proposed scheme and typical for consume-oriented applications where only a small fraction of users add or change content. Typical applications for this scenario are video on-demand platforms, Wikis and e-commerce platforms. For low RPRs the proposed scheme can require a higher number of processing than a traditional scheme. Low RPRs are typical for produce-oriented applications such as online messaging platforms where not all messages are read. In general, it must be noted that information typically is processed to be consumed, as otherwise the information remains unused.

7.8 Summary

In conclusion, it has been shown that the overall performance of a web application integrated into a WSF is a trade-off among optimised processing cost, processing duration and storage space. Figure 7.8 shows a SPD (read *speedy*) performance optimisation triangle in the style of the CAP theorem (Gilbert and Lynch, 2002). The triangle illustrates how a system can only be optimised for two out of three goals simultaneously. A traditional web application with vertical scaling typically

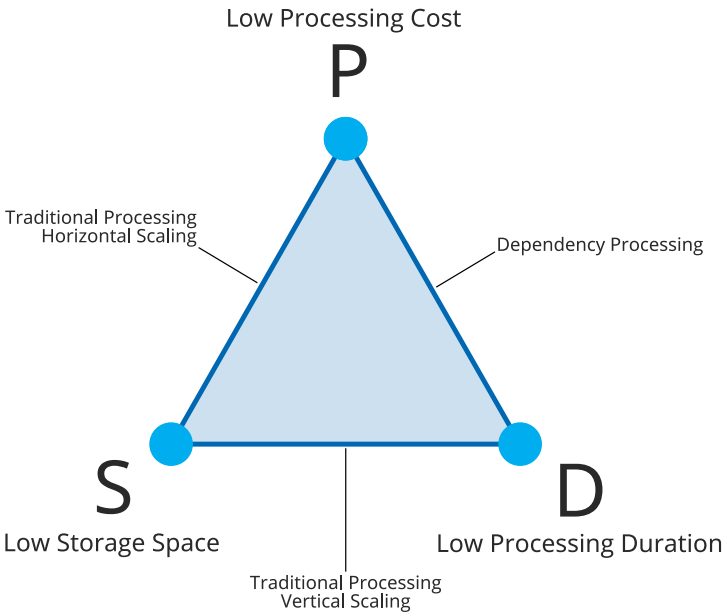


Figure 7.8: Performance optimsation triangle with low processing cost, low storage space and low processing duration.

requires low storage space S as it caches only parts of all resources. It further can achieve low processing durations D by scaling the system vertically, e.g. by adding a faster CPU or better network. However, it can not exhibit low processing cost P as vertical scaling is more expensive than horizontal scaling and cache misses need to be processed. A traditional web application with horizontal scaling also requires low storage space S due to partial caching. It further can achieve low processing cost P by employing multiple, inexpensive machines with low hardware specifications. However, with these low hardware specifications it can not achieve low processing durations D, with the same number of machines than SD uses. A web application using resource dependency processing can provide low processing durations D as all requestable resources are preprocessed and immediately available for delivery. It further can achieve low processing cost P as it processes only requests that require processing and does not evict and reprocess resources to save storage space. Consequently, it can not achieve the goal for low storage space S. To fully evaluate all approaches, the optimisation goals have to be weighted by the cost that occurs when violating a goal. Both the storage space S and processing cost P are based on cloud provider pricing. The low processing duration goal D is based on user experience as it directly influences the response times for customer facing requests. For illustration purposes it is assumed that 1000 requests should be processed where the RPR of the requests is 0.8. Further, a large CPU is given that is able to process 1000 requests/s for 20$, a small CPU is given that is able to process 100 requests/s for 2$ and the storage of 100 resources is assumed to cost 1$. For the traditional approach with vertical scaling this means that 200 resources are stored for 2$, 800 resources are processed with 1 large CPU for 20$ in 0.8 seconds of time which totals to 22$. Further, for the traditional approach with horizontal scaling this means that 200

resources are stored for 2$, 800 requests are processed with 8 small CPUs for 16$ in 1 second of time which totals to 18$. Finally, for the resource dependency approach this means that 1000 resources are stored for 10$, 200 resources are processed with 2 small CPUs for 4$ in 1 second of time which totals to 14$. Consequently, the triangle shows the trade-offs between S, P and D. Further, it has been shown that the use of a common Linux container format enables efficient portability and interoperability of components between cloud providers. When integrating applications into a WSF, a detailed knowledge of the entity relations and resource structure is needed. However, with the help of automated application analysis tools, future applications can be integrated automatically during the development process.

8. Conclusions and Future Work

8.1 Overview

This thesis has examined a novel class of frameworks that can take over the complex task of automatic scaling in an optimised fashion. The work has been split into four major research objectives that all have been completed successfully and are further described in the section of proposed solutions. The conclusions of the work are based on the completed research objectives and are presented in the major findings and contributions to technical and methodological knowledge sections in a generalised form. Finally, the limitations of this study are presented, where the future work section gives an outlook on how to overcome those limitations and concludes the thesis.

8.2 Proposed Solutions

The research aim and objectives defined in Chapter 1 have been met by proposing and investigating novel solutions within the problem space. The solutions to the four objectives can be summarised as follows:

- **O1**: *(To) separate the concerns of application logic and scaling logic:* A conceptual architecture design including required modules, interfaces, parameters and components valid for all implementations of Web Scaling Frameworks (WSFs) has been presented. The major identified design goal was to create an architecture that enables building maintainable, automatable, scalable, resilient, portable and interoperable implementations of WSFs. Further, the provider adapter cloud pattern, the managed configuration cloud pattern, the elastic manager cloud pattern, the command query responsibility segregation and Flux pattern, the watchdog pattern and the microservice architecture pattern were applied to the conceptual architecture. The architecture was designed to use a storage module, a metrics module, a watcher module, a resilience module, an actions module, a provision module, an interface module and a worker module in order to separate the concerns of implementation. Parameters were divided into component parameters, system parameters and traffic parameters to configure and manage scaling. Additionally, a minimal viable set of interfaces has been presented that includes component interfaces, framework interfaces and application interfaces.
- **O2**: *(To) distribute work to multiple components and fully benefit from a novel caching approach:* A novel request flow scheme has been designed and imple-

mented. The proposed Permanent Resource Storage and Management pattern divides resource models into individually scalable, manageable and decoupled read and processing subsystem that guarantees constant response times for all read requests, releases applications from avoidable load and ensures that changes are processed only exactly once for the whole system. An implementation of the Permanent Resource Storage and Management (PRSM) pattern was presented that provides an efficient and scalable composition of components. Additionally, a mechanism to synchronously and asynchronously process dependencies in order to enforce eventual or strong consistency of resources was proposed. To the minimum viable interfaces, a resource interface was presented as extension to the worker interface. The resource interface provides actions to manage the storage and meta information such as dependencies of resources. Component and composition parameters and models were developed that allow the analytical evaluation of request flow and total machines performance. Additionally, models to compare the performance between a traditional scheme and the proposed scheme were presented. Metrics and models were developed that are able to measure and optimise the performance of components by operating them in the optimal concurrency range. Results showed that all evaluated real-world applications need significantly fewer machines (63%, 32%, 92%) with the proposed scheme than the traditional composition and flow scheme. Additionally, the results have shown that the average time available to process dependencies was positive for all applications with 2.69, 1.19 and 26.04 seconds. The component, composition and mean prediction fit for all applications further support the proposed models.

- **O3**: *(To) optimise the processing performance of the novel caching approach:* It has been shown that resource dependencies can be stored as directed acyclic graphs, where vertices represent resources and edges dependencies. Further, the dependency depth, dependency degree, cluster count, cluster size and sparsity were identified as the most influential graph measures. With respect to optimisation algorithms, it has been found that applying shortest-path algorithms for dependency processing is not suitable as the processing order needs to be maintained. Hence a longest-path algorithm combining a topological sort with dynamic programming which determines the processing order in linear time was developed. The dependency depth and cluster size were found to have a linear correlation with the processing duration, where the accuracy of regressions based on both measures depends on the sparsity of the graph. The generation of random dependency graphs was based on service structures where the parameters were extracted from six real-world social applications and a fuzzy algorithm with random parameters. The processing duration was found to be adequately modelled based on the cluster size and

the dependency depth. This allowed replacing the constant resource dependency processing delay from Chapter 5 with an exact model. The cluster size based model has an overall model fit of 96% and is inexpensive to compute, where the dependency depth based model has a model fit of 98%, thus being more accurate but also more expensive to determine. The duration delta and relative performance improvement models enable comparing the performance of a traditional processing approach versus a resource dependency approach. Finally, the evaluation of 2000 web services with 850 thousand resources and over 6 million requests has shown the resource dependency processing approach to be up to a factor of two faster than a traditional processing approach.

- **O4**: *(To) enable multiple cloud provider systems and predict resource cost:* It has been shown how components can be operated and moved across multiple cloud providers by utilising Linux containers and provider specific Software as a Services (SaaSs). Three steps to integrate a traditional app into a WSF were presented where in the first step the service structure was analysed to extract the dependencies, in the second step the dynamic dependencies were injected into the web application, and in the third step the resource index was created for the initial resource database fill up. A processing cost and storage space model based on mean dependency degrees, entity quantities and resource fragment sizes was developed. Both models fit the evaluation data by 97% and 99% for the processing cost and storage space models. It was found that in optimal cases the dependency processing approach both has 35% lower processing cost while being 80% faster than the traditional processing approach. When both approaches exhibit equal processing cost, the proposed dependency approach was 60% faster than the traditional processing approach.

8.3 Major Findings

In recent relevant work, no platform, framework or class of frameworks exists that can take over the complex task of automatic scaling in an optimised fashion. Current solutions propose work that improve single components with optimised algorithms and data structures or propose the set-up and use of individual hosting solutions. For the scaling of web services, this means that for each application a customised hosting system needs to be set up from scratch, or with the help of services from a cloud provider. The proposed composition of multiple cloud architecture patterns into a reusable framework reduces hosting complexity and improves the overall service reliability. The major design goal for WSFs is to create an architecture that enables building maintainable, automatable, scalable, resilient, portable and interoperable implementations of WSFs. While single cloud application patterns exist to implement these goals, there is no class of frameworks that compose this patterns into concrete and usable solutions to scale web services. For the scaling of web

services, this means that for each application individual cloud application patterns have to be evaluated and implemented. This increases the required time to develop scalable web services enormously while reducing the time available to implement the application logic. A permanent storage and update mechanism with an optimised routing scheme in WSFs allows a fine-grained scalability of read and processing subsystems and thereby helps to reduce the total amount of machines required to satisfy a targeted load. In current solutions, all requests flow through the web application that queries a cache to find out if a request needs to be processed or the response can be taken from cache. For the scaling of web services, this means that the finest granularity available for scaling is the entire application, where for cached responses only the cache is required to respond to a request. Consequently, to adapt to traffic that mainly reads resources, the entire application has to be scaled, where scaling the cache subsystem would be sufficient. This results in an unnecessary waste of resources which can be reduced by creating a system with finer-grained scalability, such as the proposed WSF. An explicit definition and management of resource dependencies can remove a heuristic cache invalidation approach, leading to fewer total processings and a lower processing duration. In current solutions, cache eviction is based on heuristics, such as access frequency, patterns or timeouts. This leads to both unpredictable response times when a cache miss occurs and multiple processings of the same resource. For the scaling of web services, the explicit definition and processing of resource dependencies enables decoupling the resource access from the resource update and management and thereby allows an individual scaling of both subsystems. Further, knowledge of the dependency structure allows predicting update times that can be used to scale the system more accurately than a heuristic cache eviction. The utilisation and performance prediction of services from multiple cloud providers enables building resilient, flexible and cost-effective systems. For the scaling of web services, this means that a prediction of required storage space and number of processings can be used to base scaling decisions on calculated costs of different service providers. Identifying accurate measurements of web service properties, such as content distributions and sizes, are important to effectively manage and predict resource requirements prior to implementation.

8.4 Contributions to Knowledge

The work in this thesis largely contributes to the generation of a new level of abstraction for cloud deployment and hosting. With the adoption of the proposed class of frameworks, a variety of WSF implementations can contribute to enabling future applications to focus on enhanced application logic as opposed to deploying and hosting logic. This enhanced focus in turn can be used to create a series of new generation smart services helping to unravel the true power of cloud computing. The major abstractions of the proposed WSF are the generalisation of the manage-

ment and composition of service components and the optimisation of request and processing performance. The generalisation of the management and composition of service components enables creating interoperable, mathematically predictable and optimised compositions that can be reused for various web services. The presented component normalisation with delays over different hardware and implementations contributes to methodological knowledge, where future modelling can be based on the versatile definition of delays as presented by this work. Further, the division of the data flow system into multiple, decoupled subsystems as presented in this work, enables a versatile and fine-grained scalability of individual subsystems and thereby contributes to technical knowledge. The generalisation of the performance optimisation through detailed analysis of service data structures enables creating reusable performance optimisation models that can be developed and applied to web services in an optimised fashion. The analysis, modelling and generation of evaluation data based on real world Application Programming Interfaces (APIs) have showed that if a direct evaluation approach is not possible, the indirect can lead to usable data, which contributes to methodological knowledge. Further, by understanding the inner relationships and data structures of an application in detail, heuristic optimisations can be replaced by more accurate optimisations, which contributes to technical knowledge. To sum up, the benefits that are introduced by the new level of abstraction as presented in this work, are an increased reusability of hosting and deployment logic, which reflects in an enhanced reliability through community tested software, and the reduction of the work needed to implement scalable web services.

8.5 Limitations

This study is subject to the following limitations that can be overcome by future work presented in the next section. Due to budget limitations, all evaluations were run with a maximum of 42 machines, where large scale web applications are known to use hundreds of machines. This limits the scope of the evaluations, where future work can further validate the models with more machines. Further, due to limitations of project time, the evaluations were not carried out on full production web services over a long period of time. This limits the scope of the evaluations to the utilised traffic traces and generated application resources. These, however, were chosen to reflect a large spectrum of applications with versatile properties. The interaction with a service that uses resource dependency processing changes the dependency graph during runtime. In this work, after each change the complete forest of processing trees is recalculated as currently there are no online algorithms to calculate incremental changes only. Consequently, the evaluation results are limited in processing speed due to the complete recalculation of the optimal processing trees. Further, in this work a resource can be updated by the resource dependency processing algorithm at any point in time. This can lead to multiple subsequent up-

dates of a key that are overwritten in near future. Consequently, the modelling and evaluation results include unnecessary updates not visible to a user, which could be eliminated with future work. In Chapter 2 it has been found that web applications have an optimal concurrency range depending on software and hardware, where the throughput is at its maximum. This throughput maximum is not utilised by the algorithms presented in this work. Consequently, the results are limited to the average throughput which could be improved by future work. Finally, the results in this work expect an application to be either completely operated with a traditional request flow scheme or the novel proposed request flow scheme. Consequently, the modelling and evaluations do not include a dynamic approach that can switch between both approaches.

8.6 Future Work

The proposed optimisation schemes are open to further research. The following is a non-comprehensive list of potential future work with a focus on further improving the novel request routing scheme, resource dependency processing and component orchestration of the proposed solutions:

- With algorithms exhibiting incremental graph updates online, the management work involved when a resource changes its dependencies could be reduced significantly. This would lead to speed-ups in the worker ultimately reflecting in a higher component throughput.
- Restrict the number of resource dependency updates to discrete time slots. With an update rate limit of one per second, this could reduce the number of updates during a five second period to a maximum of five. It is expected that this optimisation reduces the volume of total processing massively, while having a minor influence on content propagation speed.
- With the knowledge of the number of updates including dependencies, algorithms can be developed that operate application components only within their optimal concurrency range. Due to workers pulling requests out of the queue, the load is expected to be distributed in a further optimised fashion compared with a request push approach that is used for traditional applications. This is expected to significantly increase the throughput of the novel routing scheme.
- The results from all models and evaluations in this work have shown that for low read/processing ratios a traditional processing approach can achieve superior performance compared with dependency processing. The design of a hybrid system able to switch between both processing modes could increase the field of application to applications with highly dynamic read/processing ratios while benefiting from the best of both approaches.

- In the current conceptual architecture, WSFs have to be configured to use components from a selection of cloud providers. The development of automated component/cloud provider selection algorithms could reduce the cost by implementing time-based price bidding functionalities and selecting the optimal workload/component composition and orchestration.
- Developer assistive systems can be designed to help developing application structures optimally suited for dependency processing. Assistive systems pointing out critical paths with high dependency depths are expected to reduce application complexity ultimately leading to fewer, thus faster updates. By analysing existing application structures and extracting dependency graphs, assistive systems could additionally reduce the time needed to adopt the optimisation schemes proposed in this thesis.

List of References

Abrishami, S., M. Naghibzadeh and D.H.J. Epema (2012). 'Cost-Driven Scheduling of Grid Workflows Using Partial Critical Paths'. In: *IEEE Transactions on Parallel and Distributed Systems* 23.8, pp. 1400–1414.

Addo, I.D., Duc Do, Rong Ge and S.I. Ahamed (2015). 'A Reference Architecture for Social Media Intelligence Applications in the Cloud'. In: *2015 IEEE 39th Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2, pp. 906–913.

Ahn, Jaesuk, Eui-Jik Kim, Bokuk Seo, Ki-Young Lee and Eunju Kim (2015). 'Open Cloud Architecture for Public Sector: Requirements and Architecture'. In: *2015 International Conference on Platform Technology and Service (PlatCon)*, pp. 43–44.

Ajwani, Deepak and Tobias Friedrich (2010). 'Average-case analysis of incremental topological ordering'. In: *Discrete Applied Mathematics* 158.4, pp. 240–250.

Amazon CloudFormation (2011). [Online] Available: `http://aws.amazon.com/cloudformation` [Accessed 1st Apr. 2016].

Amazon EC2 Container Service (2014). [Online] Available: `https://aws.amazon.com/ecs` [Accessed 1st Apr. 2016].

Amazon Kinesis Streams (2014). [Online] Available: `https://aws.amazon.com/kinesis` [Accessed 1st Apr. 2016].

Amazon Lambda (2014). [Online] Available: `https://aws.amazon.com/lambda` [Accessed 1st Apr. 2016].

Amazon Web Services (2006). [Online] Available: `https://aws.amazon.com` [Accessed 1st Apr. 2016].

Andrikopoulos, Vasilios, Santiago Gómez Sáez, Frank Leymann and Johannes Wettinger (2014). 'Optimal Distribution of Applications in the Cloud'. In: *Advanced Information Systems Engineering*. Vol. 8484. Springer International Publishing, pp. 75–90.

Apache Hadoop YARN (2011). [Online] Available: `http://hadoop.apache.org` [Accessed 1st Apr. 2016].

Apache Kafka (2012). [Online] Available: `http://kafka.apache.org` [Accessed 1st Apr. 2016].

Apache Mesos (2012). [Online] Available: `http://mesos.apache.org` [Accessed 1st Apr. 2016].

Apache Samza (2012). [Online] Available: `http://samza.apache.org` [Accessed 1st Apr. 2016].

Apache Spark (2014). [Online] Available: `http://spark.apache.org` [Accessed 1st Apr. 2016].

Apache Storm (2015). [Online] Available: `http://storm.apache.org` [Accessed 1st Apr. 2016].

Bangar, P. and K.N. Singh (2015). 'Investigation and performance improvement of web cache recommender system'. In: *Proceedings IEEE International Conference on Future Trends on Computing Analytics (ABLAZE15)*, pp. 585–589.

Batarfi, Omar, Radwa El Shawi, Ayman G. Fayoumi, Reza Nouri, Seyed-Mehdi-Reza Beheshti, Ahmed Barnawi and Sherif Sakr (2015). 'Large scale graph processing systems: survey and an experimental evaluation'. In: *Springer Cluster Computing* 18.3, pp. 1189–1213.

Bellman, Richard (1954). 'The Theory of Dynamic Programming'. In: *RAND Corp, Santa Monica*.

Binz, Tobias, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak and Sebastian Wagner (2013). 'OpenTOSCA – A Runtime for TOSCA-Based Cloud Applications'. In: *Service-Oriented Computing*. Vol. 8274. Springer Berlin Heidelberg, pp. 692–695.

Binz, Tobias, Uwe Breitenbücher, Oliver Kopp and Frank Leymann (2014). 'TOSCA: Portable Automated Deployment and Management of Cloud Applications'. In: *Advanced Web Services*. Springer New York, pp. 527–549.

Bocchi, E., M. Mellia and S. Sarni (2014). 'Cloud storage service benchmarking: Methodologies and experimentations'. In: *Proceedings IEEE International Conference on Cloud Networking (CloudNet14)*, pp. 395–400.

CAMP v1.1 (2014). [Online] Available: `http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.pdf` [Accessed 1st Apr. 2016].

Chanas, Stefan and Paweł Zieliński (2001). 'Critical path analysis in the network with fuzzy activity times'. In: *Fuzzy Sets and Systems* 122.2, pp. 195–204.

Chen, Jiongze, Ronald G Addie, Moshe Zukerman and Timothy D Neame (2015). 'Performance evaluation of a queue fed by a Poisson Lomax Burst process'. In: *IEEE Communication Letter* 19.3, pp. 367–370.

Chen, Shuang, Mohammadmersad Ghorbani, Yanzhi Wang, Paul Bogdan and Massoud Pedram (2014). 'Trace-Based Analysis and Prediction of Cloud Computing User Behavior Using the Fractal Modeling Technique'. In: *IEEE International Congress on Big Data*, pp. 733–739.

Ching, Avery, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis and Sambavi Muthukrishnan (2015). 'One Trillion Edges: Graph Processing at Facebook-scale'. In: *Proceedings VLDB Endow.* 8.12, pp. 1804–1815.

Cormen, Thomas H (2009). *Introduction to Algorithms*. MIT press.

Di Martino, Beniamino, Giuseppina Cretella and Antonio Esposito (2015). 'Cloud Portability and Interoperability'. In: *Cloud Portability and Interoperability*. Springer Briefs in Computer Science. Springer International Publishing, pp. 1–14.

Dick, Scott, Omolbanin Yazdanbaksh, Xiuli Tang, Toan Huynh and James Miller (2014). 'An empirical investigation of Web session workloads: Can self-similarity be explained by deterministic chaos?' In: *Information Processing & Management* 50.1, pp. 41–53.

DigitalOcean (2011). [Online] Available: `https://digitalocean.com` [Accessed 1st Apr. 2016].

Docker (2013). [Online] Available: `https://www.docker.com` [Accessed 1st Apr. 2016].

Docker Swarm (2015). [Online] Available: `https://www.docker.com/docker-swarm` [Accessed 1st Apr. 2016].

Donthi, Ranadheer, Ramesh Renikunta, Rajaiah Dasari and Malla Reddy Perati (2014). 'Self-Similar Network Traffic Modeling Using Circulant Markov Modulated Poisson Process'. In: *Fractals, Wavelets, and their Applications*. Springer, pp. 437–444.

Du, Cong and Suozhu Wang (2015). 'Research on Mobile Web Cache Prefetching Technology Based on User Interest Degree'. In: *Proceedings of 3rd International Conference on Logistics, Informatics and Service Science*. Springer Berlin Heidelberg, pp. 1253–1258.

Espadas, Javier, Arturo Molina, Guillermo Jiménez, Martín Molina, Raúl Ramírez and David Concha (2013). 'A tenant-based resource allocation model for scaling Software-as-a-Service applications over cloud computing infrastructures'. In: *Future Generation Computing Systems* 29.1, pp. 273–286.

Facebook Flux (2014). [Online] Available: `https://facebook.github.io/flux` [Accessed 1st Apr. 2016].

Fankhauser, T., Q. Wang, A. Gerlicher and C. Grecos (2016). 'Resource Dependency Proceedingsssing in Web Scaling Frameworks'. In: *IEEE Transactions on Services Computing*, pp. 1–1.

Fankhauser, T., Q. Wang, A. Gerlicher, C. Grecos and Wang, X. (2015). 'Web Scaling Frameworks for Web Services in the Cloud'. In: *IEEE Transactions on Services Computing*, pp. 1–1.

Fankhauser, T., Q. Wang, A. Gerlicher, C. Grecos and X. Wang (2014). 'Web Scaling Frameworks: A novel class of frameworks for scalable web services in cloud environments'. In: *Proceedings IEEE International Conference on Communications (ICC14*, pp. 1414–1418.

Fehling, Christoph, Frank Leymann, Ralph Retter, Walter Schupeck and Peter Arbitter (2014). 'Cloud Computing Fundamentals and Composite Cloud Application Patterns'. English. In: *Cloud Computing Patterns*. Springer Vienna. ISBN: 9783709115671.

Fielding, Roy Thomas (2000). 'Architectural styles and the design of network-based software architectures'. PhD thesis. University of California, Irvine.

Fowler, Martin (2009). *Eager Read Derivation*. [Online] Available: `http://martinfowler.com/bliki/EagerReadDerivation.html` [Accessed 1st Apr. 2016].

Fowler, Martin (2011). *CQRS*. [Online] Available: `http://martinfowler.com/bliki/CQRS.html` [Accessed 1st Apr. 2016].

Fowler, Martin (2014). *Microservices*. [Online] Available: `http://martinfowler.com/articles/microservices.html` [Accessed 1st Apr. 2016].

Gilbert, Seth and Nancy Lynch (2002). 'Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services'. In: *SIGACT News* 33.2, pp. 51–59. ISSN: 0163-5700.

Go (2009). [Online] Available: `https://golang.org` [Accessed 1st Apr. 2016].

Google Cloud Dataflow (2015). [Online] Available: `https://cloud.google.com/dataflow` [Accessed 1st Apr. 2016].

Google Cloud Deployment Manager (2015). [Online] Available: `https://cloud.google.com/deployment-manager` [Accessed 1st Apr. 2016].

Google Cloud Platform (2008). [Online] Available: `https://cloud.google.com/compute` [Accessed 1st Apr. 2016].

Google Container Engine (2014). [Online] Available: `https://cloud.google.com/container-engine` [Accessed 1st Apr. 2016].

Google Kubernetes (2014). [Online] Available: http://kubernetes.io [Accessed 1st Apr. 2016].

Guo, Yong, M. Biczak, A.L. Varbanescu, A. Iosup, C. Martella and T.L. Willke (2014). 'How Well Do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis'. In: *IEEE Parallel and Distributed Proceedings Symposium*, pp. 395–404.

Haeupler, Bernhard, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen and Robert E. Tarjan (2012). 'Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance'. In: *ACM Transactions Algorithms* 8.1, 3:1–3:33.

Han, Hyuck, Young Choon Lee, Woong Shin, Hyungsoo Jung, H.Y. Yeom and AY. Zomaya (2012). 'Cashing in on the Cache in the Cloud'. In: *IEEE Transactions on Parrallel and Distributed Systems* 23.8, pp. 1387–1399. ISSN: 1045-9219. DOI: 10.1109/TPDS.2011.297.

Han, Rui, Moustafa M Ghanem, Li Guo, Yike Guo and Michelle Osmond (2014). 'Enabling cost-aware and adaptive elasticity of multi-tier cloud applications'. In: *Future Generation Computing Systems* 32, pp. 82–98.

Haupt, F., F. Leymann, A. Nowak and S. Wagner (2014). 'Lego4TOSCA: Composable Building Blocks for Cloud Applications'. In: *Proceedings IEEE International Conference on Cloud Computing (CLOUD14)*, pp. 160–167.

Hewlett-Packard (1999). [Online] Available: http://ita.ee.lbl.gov/html/contrib/WorldCup.html [Accessed 1st Apr. 2016].

Hummer, Waldemar, Benjamin Satzger and Schahram Dustdar (2013). 'Elastic stream processing in the Cloud'. In: *Wiley Interdisciplinary Revision: Data Mining and Knowledge Discovery* 3.5, pp. 333–345.

IBM Bluemix (2014). [Online] Available: https://www.ibm.com/bluemix [Accessed 1st Apr. 2016].

IBM Containers for Bluemix (2014). [Online] Available: https://www.ng.bluemix.net/docs/containers [Accessed 1st Apr. 2016].

IETF RFC2616 (1999). [Online] Available: https://tools.ietf.org/html/rfc2616 [Accessed 1st Apr. 2016].

Internet Archive, a 501(c)(3) non-profit (1996). [Online] Available: http://httparchive.org [Accessed 1st Apr. 2016].

Inzinger, C., S. Nastic, S. Sehic, M. Vögler, Fei Li and S. Dustdar (2014). 'MADCAT: A Methodology for Architecture and Deployment of Cloud Application Topologies'.

In: *Proceedings IEEE International Symposium Service Oriented Systems Engineering (SOSE'14)*, pp. 13–22.

Jacko, Julie A, Andrew Sears and Michael S Borella (2000). 'The effect of network delay and media on user perceptions of web resources'. In: *Behaviour & Information Technology* 19.6, pp. 427–439.

Jiang, Jing, Jie Lu, Guangquan Zhang and Guodong Long (2013). 'Optimal Cloud Resource Auto-Scaling for Web Applications'. In: *Proceedings IEEE/ACM International Symposium Cluster, Cloud and Grid Computing (CCGrid13)*, pp. 58–65.

Kalashnikov, D., A. Bartashev, A. Mitropolskaya, E. Klimov and N. Gusarova (2015). 'Cerrera: In-stream data analytics cloud platform'. In: *2015 Third International Conference on Digital Information, Networking, and Wireless Communications (DINWC)*, pp. 170–175.

Katsaros, G., M. Menzel, A. Lenk, J. Rake-Revelant, R. Skipp and J. Eberhardt (2014). 'Cloud Application Portability with TOSCA, Chef and Openstack'. In: *Proceedings IEEE International Conference on Cloud Engineering (IC2E14)*, pp. 295–302.

Katsaros, Konstantinos V, George Xylomenos and George C Polyzos (2012). 'GlobeTraff: a traffic workload generator for the performance evaluation of future Internet architectures'. In: *IEEE International Conference on New Technology, Mobility and Security (NTMS12)*. IEEE, pp. 1–5.

Kelley Jr, James E. and Morgan R. Walker (1959). 'Critical-path Planning and Scheduling'. In: *ACM IRE-AIEE Computing Conference*. ACM, pp. 160–173.

Kostoska, Magdalena, Marjan Gusev and Sasko Ristov (2014). 'A New Cloud Services Portability Platform'. In: *Proceedings DAAAM International Symposium Intelligent Manufacturing and Automation* 69, pp. 1268–1275.

Krintz, C. (2013). 'The AppScale Cloud Platform: Enabling Portable, Scalable Web Application Deployment'. In: *Journal IEEE International Computing* 17.2, pp. 72–75.

Kroß, Johannes, Andreas Brunnert, Christian Prehofer, ThomasA. Runkler and Helmut Krcmar (2015). 'Stream Processing on Demand for Lambda Architectures'. In: *Computer Performance Engineering*. Vol. 9272. Springer International Publishing, pp. 243–257. ISBN: 9783319232669.

Le Scouarnec, N., C. Neumann and G. Straub (2014). 'Cache Policies for Cloud-Based Systems: To Keep or Not to Keep'. In: *Proceedings IEEE International Conference on Cloud Computing (CLOUD14)*, pp. 1–8.

libuv (2011). [Online] Available: https://github.com/joyent/libuv [Accessed 1st Apr. 2016].

Little (1961). 'A Proof for the Queuing Formula'. In: *Operations Research* 9.3, pp. 383–387.

Loulloudes, Nicholas, Chrystalla Sofokleous, Demetris Trihinas, Marios D Dikaiakos and George Pallis (2015). 'Enabling Interoperable Cloud Application Management through an Open Source Ecosystem'. In: *IEEE Internet Computing* 19.3, pp. 54–59.

LXC (2008). [Online] Available: https://linuxcontainers.org [Accessed 1st Apr. 2016].

Maheshwari, Ketan, Eun-Sung Jung, Jiayuan Meng, Vitali Morozov, Venkatram Vishwanath and Rajkumar Kettimuthu (2016). 'Workflow performance improvement using model-based scheduling over multiple clusters and clouds'. In: *Future Generation Computing Systems* 54, pp. 206–218.

Malcolm, D. G., J. H. Roseboom, C. E. Clark and W. Fazar (1959). 'Application of a Technique for Research and Development Program Evaluation'. In: *INFORMS Operations Research* 7.5, pp. 646–669.

Mao, Ming and Marty Humphrey (2011). 'Auto-scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows'. In: *Proceedings ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*. Seattle, Washington: ACM, 49:1–49:12.

Margara, Alessandro and Guido Salvaneschi (2014). 'We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees'. In: *ACM Proceedings of International Conference on Distributed Event-Based Systems, (DEBS14)*, pp. 142–153.

Marshall, Paul, Kate Keahey and Tim Freeman (2010). 'Elastic Site: Using Clouds to Elastically Extend Site Resources'. In: *Proceedings IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID'10)*, pp. 43–52. ISBN: 9780769540399. DOI: 10.1109/CCGRID.2010.80.

Masdari, Mohammad, Sima ValiKardan, Zahra Shahi and Sonay Imani Azar (2016). 'Towards workflow scheduling in cloud computing: A comprehensive analysis'. In: *Journal of Networking and Computing Applications*.

Mathematica, Wolfram (1988). [Online] Available: http://reference.wolfram.com/language/ref/NonlinearModelFit.html [Accessed 1st Apr. 2016].

Merkel, Dirk (2014). 'Docker: Lightweight Linux Containers for Consistent Development and Deployment'. In: *ACM Linux Journal* 2014.239. ISSN: 1075-3583.

Meusel, Robert, Sebastiano Vigna, Oliver Lehmberg and Christian Bizer (2014). 'Graph structure in the web—revisited: a trick of the heavy tail'. In: *Proceedings of the International Conference on World wide web Computing*, pp. 427–432.

Microsoft Azure (2010). [Online] Available: `https://azure.microsoft.com` [Accessed 1st Apr. 2016].

Namiot, Dmitry and Manfred Sneps-Sneppe (2014). 'On Micro-services Architecture'. In: *International Journal of Open Information Technology* 2.9, pp. 24–27.

Negrão, AndrésPessoa, Carlos Roque, Paulo Ferreira and Luís Veiga (2015). 'An adaptive semantics-aware replacement algorithm for web caching'. In: *Journal of International Service and Applications*.

NIST SP800-145 (2011). [Online] Available: `http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf` [Accessed 1st Apr. 2016].

node.js (2009). [Online] Available: `https://nodejs.org` [Accessed 1st Apr. 2016].

OASIS (1993). [Online] Available: `https://www.oasis-open.org` [Accessed 1st Apr. 2016].

OpenGroup Cloud Computing Portability and Interoperability (2004). [Online] Available: `http://www.opengroup.org/cloud/cloud/cloud_iop/cloud_port.htm` [Accessed 1st Apr. 2016].

OpenStack Heat (2014). [Online] Available: `http://docs.openstack.org/developer/heat` [Accessed 1st Apr. 2016].

OpenVZ (2005). [Online] Available: `https://openvz.org` [Accessed 1st Apr. 2016].

Pang, Chaoyi, Junhu Wang, Yu Cheng, Haolan Zhang and Tongliang Li (2015). 'Topological sorts on {DAGs}'. In: *Information Processing Letters* 115.2, pp. 298–301.

Petcu, Dana, Georgiana Macariu, Silviu Panica and Ciprian Crciun (2013). 'Portable Cloud applications-From Theory to Practice'. In: *Future Generation Computing Systems* 29.6, pp. 1417–1430.

Petcu, Dana, BeniaminoDi Martino, Salvatore Venticinque, Massimiliano Rak, Tamás Máhr, GorkaEsnal Lopez, Fabrice Brito, Roberto Cossu, Miha Stopar, Svatopluk Šperka and Vlado Stankovski (2013). 'Experiences in building a mOSAIC of clouds'. In: *Journal Cloud Computing* 2.1, 12.

Pettersen, R., S.V. Valvag, A. Kvalnes and D. Johansen (2014). 'Jovaku: Globally Distributed Caching for Cloud Database Services Using DNS'. In: *Proceedings IEEE International Conference on Mobility Cloud Computing (MobileCloud14)*, pp. 127–135.

Poggi, Nicolas, David Carrera, Ricard Gavalda, Eduard Ayguadé and Jordi Torres (2014). 'A methodology for the evaluation of high response time on E-commerce users and sales'. In: *Information Systems Frontiers* 16.5, pp. 867–885.

Qanbari, S., Fei Li and S. Dustdar (2014). 'Toward Portable Cloud Manufacturing Services'. In: *Journal IEEE International Computing* 18.6, pp. 77–80.

Qin, Xiulei, Wenbo Zhang, Wei Wang, Jun Wei, Hua Zhong and Tao Huang (2011). 'On-line Cache Strategy Reconfiguration for Elastic Caching Platform: A Machine Learning Approach'. In: *Proceedings IEEE Annual Computing Software and Applications Conference (COMPSAC11)*, pp. 523–534.

RackSpace (1998). [Online] Available: `https://rackspace.com` [Accessed 1st Apr. 2016].

Rajabi, Ali and Johnny W Wong (2014). 'Provisioning of Computing Resources for Web Applications under Time-Varying Traffic'. In: *IEEE International Symposium on Modelling, Analytics & Simulation of Computing and Telecommunication Systems*. IEEE, pp. 152–157.

Ramachandran, Arthi, Yunsung Kim and Augustin Chaintreau (2014). 'I knew they clicked when i saw them with their friends: identifying your silent web visitors on social media'. In: *ACM Proceedings of the Conference on Online Social Networks*, pp. 239–246.

Ray, Santanu Saha (2013). *Graph Theory with Algorithms and Its Applications*. Springer, India.

Redis (2009). [Online] Available: `http://redis.io` [Accessed 1st Apr. 2016].

Salvaneschi, G., A. Margara and G. Tamburrelli (2015). 'Reactive Programming: A Walkthrough'. In: *IEEE International Conference on Software Engineering (ICSE15)*. Vol. 2, pp. 953–954.

Salvaneschi, Guido and Mira Mezini (2014). 'Towards Reactive Programming for Object-Oriented Applications'. In: *Transactions on Aspect Oriented Software Development XI*. Springer Berlin Heidelberg, pp. 227–261.

Sarhan, A., A.M. Elmogy and S.M. Ali (2014). 'New Web cache replacement approaches based on internal requests factor'. In: *Proceedings IEEE International Conference on Computing Engineering Systems (ICCES14)*, pp. 383–389.

Sedgewick, Robert (2014). *Algorithms II*. Addison-Wesley Professional.

Songwattana, Areerat, Thanaruk Theeramunkong and Phan Cong Vinh (2014). 'A learning-based approach for web cache management'. In: *Mobile Networks and Applications* 19.2, pp. 258–271.

TOSCA v1.0 (2013). [Online] Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html [Accessed 1st Apr. 2016].

Tung, T., Shaw-Yi Chaw, Qing Xie and Qian Zhu (2012). 'Highly Resilient Systems for Cloud'. In: *Proceedings IEEE International Conference on Web Services (ICWS12)*, pp. 678–680. DOI: 10.1109/ICWS.2012.66.

Visala, Kari, Ana Keating and Reduan H Khan (2014). 'Models and tools for the high-level simulation of a name-based interdomain routing architecture'. In: *IEEE Conference on Computing Communication Works. (INFOCOM WKSHPS14)*, pp. 55–60.

W3C Web Service Architecture (2004). [Online] Available: http://www.w3.org/TR/ws-arch [Accessed 1st Apr. 2016].

webscalingframeworks.org/graphs (2016). [Online] Available: http://webscalingframeworks.org/graphs [Accessed 1st Apr. 2016].

webscalingframeworks.org/traces (2016). [Online] Available: http://webscalingframework.org/traces [Accessed 1st Apr. 2016].

Wolke, Andreas and Gerhard Meixner (2010). 'TwoSpot: A Cloud Platform for Scaling Out Web Applications Dynamically'. In: *Towards a Service-Based Internet*. Ed. by Elisabetta Di Nitto and Ramin Yahyapour. Vol. 6481. Springer Berlin Heidelberg, pp. 13–24.

Wu, Yingjun and Kian-Lee Tan (2015). 'ChronoStream: Elastic Stateful Stream Computation in the Cloud'. In: *2015 IEEE 31st International Conference on Data Engineering (forthcoming)*.

Young, Greg (2010). *CQRS Documents*. [Online] Available: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf [Accessed 1st Apr. 2016].

Zareian, S., R. Veleda, M. Litoiu, M. Shtern, H. Ghanbari and M. Garg (2015). 'K-Feed - A Data-Oriented Approach to Application Performance Management in Cloud'. In: *2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*, pp. 1045–1048.

Zhang, Zhizhong, Chuan Wu and David W.L. Cheung (2013). 'A Survey on Cloud Interoperability: Taxonomies, Standards, and Practice'. In: *Journal ACM Performance Evaluation (SIGMETRICS13)* 40.4, pp. 13–22.

Zukerman, Moshe, Timothy D Neame and Ronald G Addie (2003). 'Internet traffic modeling and future technology implications'. In: *IEEE Joint Conference of Computing and Communication Society (INFOCOM2003)*. Vol. 1, pp. 587–596.

Appendix A - List of Acronyms/Abbreviations

R^2 Coefficient of Determination.

API Application Programming Interface.

ARDP asynchronous resource dependency processing.

CMMP Circulant Markov-Modulated Poisson.

CPU Central Processing Unit.

CQRS Command Query Responsibility Segregation.

CRUD Create Read Update and Delete.

DAG Directed Acyclic Graph.

DNS Domain Name System.

FARIMA Fractionally Autoregressive Integrated Moving-Average.

FBM Fractional Brownian Motion.

FDG Fuzzy Dependency Graph.

FPTE Forest of Processing Tree Extraction.

HMR Hit-Miss Ratio.

HTTP Hypertext Transfer Protocol.

IaaS Infrastructure as a Service.

IEA Incremental Edge Add.

IETF Internet Engineering Task Force.

IOPS Input/Output Operations Per Second.

IoT Internet of Things.

JSON JavaScript Object Notation.

LFU Least Frequently Used.

LPT longest-paths processing tree.

MIME Multipurpose Internet Mail Extensions.

MVI Minimum Viable Interfaces.

NIST National Institute of Standards and Technology.

NRMSE Normalised RMSE.

PaaS Platform as a Service.

PCP Partial Critical Paths.

PERT Program Evaluation and Research Task.

PLB Poisson Lomax Burst.

PPB Poisson Pareto Burst.

PRSM Permanent Resource Storage and Management.

RDB Resource Database.

RDP resource dependency processing.

REST Representational State Transfer.

RM requester mode.

RMSE Root-Mean-Square Error.

SaaS Software as a Service.

SD standard deviation.

SDG Service Based Dependency Graph.

SM sequencer mode.

SOA Service Oriented Architecture.

SPT shortest-paths processing tree.

SSH Secure Shell.

TP Traditional Processing.

URI Uniform Resource Identifier.

VCS Version Control System.

VM Virtual Machine.

W3C World Wide Web Consortium.

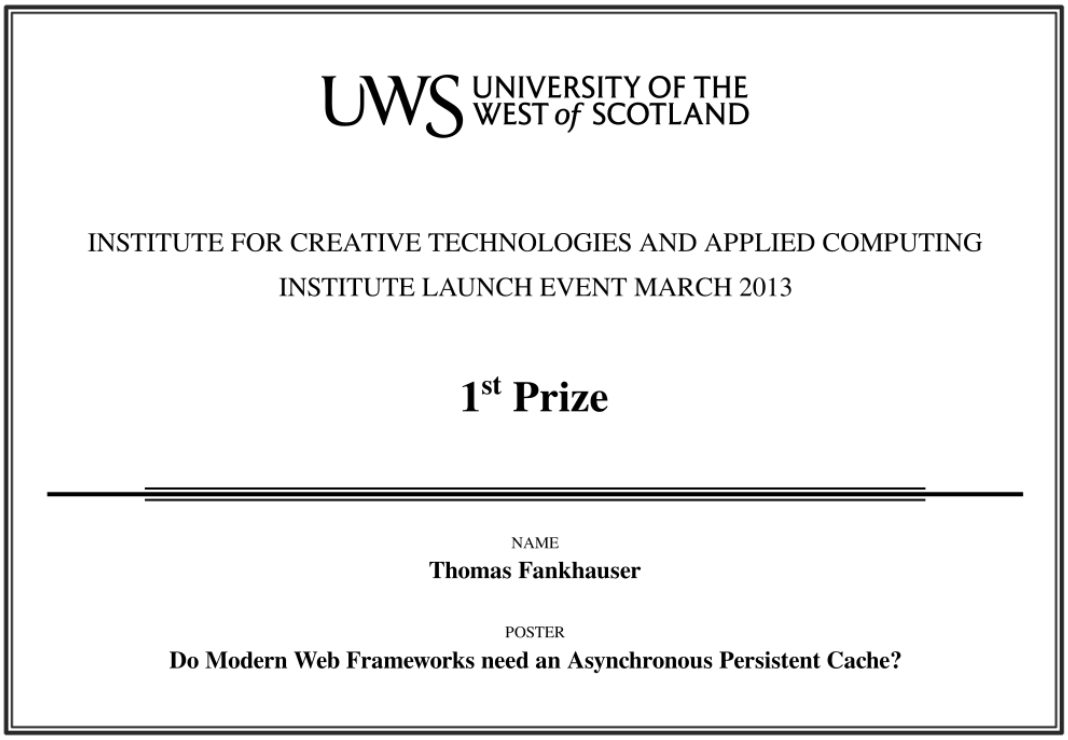
WAF Web Application Framework.

WRP Weighting Replacement Policy.

WSF Web Scaling Framework.

XML Extensible Markup Language.

Appendix B - Awards and Certificates



UWS UNIVERSITY OF THE WEST of SCOTLAND

INSTITUTE FOR CREATIVE TECHNOLOGIES AND APPLIED COMPUTING

INSTITUTE LAUNCH EVENT MARCH 2013

1st Prize

NAME

Thomas Fankhauser

POSTER

Do Modern Web Frameworks need an Asynchronous Persistent Cache?

Appendix C - Pi-One Evaluation Cluster

The cluster was custom build with 42 Raspberry Pi - Model B computers and a HP ProCurve 48 switch. The management and orchestration software shown on the monitor is a web interface monitoring the status of the cluster that runs on a separate additional node.