PLATFORM AS A SERVICE (PAAS) AGGREGATOR: A MULTI-CLOUD LIBRARY

Supervised by: Dr. H. M. N. Dilum Bandara

Prepared by: S.A.F.M. Pulle (158241A)

M.Sc. in Computer Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

May 2017

PLATFORM AS A SERVICE (PAAS) AGGREGATOR: A MULTI-CLOUD LIBRARY

Supervised by: Dr. H. M. N. Dilum Bandara

Prepared by: S.A.F.M. Pulle (158241A)

Thesis submitted in partial fulfillment of the requirements for the Degree of MSc in Computer Science specializing in Cloud Computing

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

May 2017

DECLARATION

I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief, it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: …… Date: …….

Name: S.A.F.M. Pulle

The above candidate has carried out research for the Masters dissertation under my supervision.

Signature of the supervisor: ……. Date: …….

Name: Dr. H. M. N. Dilum Bandara

ABSTRACT

Platform as a Service (PaaS) has become a key enabler of software-driven innovation facilitating rapid iteration and developer agility. While PaaS is capable of abstracting the infrastructure from the PaaS consumer, this abstraction itself, by design, makes it tightly coupled to a particular PaaS provider. Hence, a failure in any of the PaaS services could put the PaaS user in trouble. For example, in 2014 distributed cache service in Windows Azure was unavailable due to a network problem that caused service to be unreachable. Consuming PaaS services from multiple service providers has been identified as a solution for this tight coupling. These solutions rely on having another third-party layer between the PaaS services and SaaS applications. However, this does not fix the problem, as this just moves the problem from one layer to another still creating a single point of failure.

We address this problem by developing a relatively thin, abstraction layer in the form of a multi-cloud library named as the PaaS Aggregator. PaaS Aggregator, being a library, is necessarily a part of the SaaS application. It provides a unified API for the SaaS application developers to consume PaaS services. Thus, PaaS users do not need to worry about vendor-specific implementations, as the multi-cloud library provides seamless migration among different PaaS providers in case of a failure. PaaS Aggregator identifies the accessible service providers at platform level, and in turn invoke vendor-specific service calls. Proof of concept implementation of PaaS Aggregator supports database, cache, and storage services provided by Windows Azure and Amazon Web Services. Performance evaluation using a SaaS application configured to use the PaaS Aggregator showed that the throughput and response times are not affected when compared to the same application implemented on PaaS-specific APIs. However, evaluations further showed that an inefficient log storage provider might lower the overall performance of the application.

ACKNOWLEDGMENTS

My sincere appreciation goes to my family for the continuous support and motivation given to make this thesis a success. I also express my heartfelt gratitude to Dr. Dilum Bandara, my supervisor, for the supervision and advice given throughout to make this research a success. I am also thankful to the staff of the 99X Technology for providing all the required data. Last but not least, I also thank my friends who supported me in this whole effort.

I am also grateful for the support and advice given by Dr. Malaka Walpola and Dr. Indika Perera, by encouraging continuing this research till the end. Further, I would like to thank all my colleagues for their help on finding relevant research material, sharing knowledge and experience and for their encouragement.

I am deeply grateful to my parents for their love and support throughout my life. I also wish to thank my loving wife, who supported me throughout my work. Finally, I wish to express my gratitude to all my colleagues at 99X Technology, for the support given me to manage my MSc research work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Amazon EC2	Amazon Elastic Compute Cloud
API	Application Program Interface
AWS	Amazon Web Services
CaaS	Cache as a Service
CSB	Cloud Service Broker
DBaaS	Database as a Service
HA	High Availability
HS	High Security
HuaaS	Human as a Service
IaaS	Infrastructure as a Service
IaaSM	IaaS Manager
IT	Information Technology
LXC	Linux Containers
ORM	Object Relational Mapping
OVF	Open Virtualization Format
PaaS	Platform as a Service
PaaSM	PaaS Manager
PAC	PaaS Aggregator Cache
PAD	PaaS Aggregator Database
PAS	PaaS Aggregator Storage
REST	Representational State Transfer
SaaS	Software as a Service
SSaaS	Static Storage as a Service
SLA	Service Level Agreement
SDK	Software Development Kit

CHAPTER 1
INTRODUCTION

Cloud computing has become the pinnacle of utility computing over the last few years. Cloud computing provides a centralized control over the computing resources in distributed and interconnected data centers under the supervision and administration of a single service provider. Cloud computing offers many economic benefits over the traditional in-house IT systems and services due to economies of scale from the suppliers' point of view, more efficient resource utilization via demand aggregation, as well as a considerable reduction in IT management cost per Cloud consumer due to the concept of multi-tenancy architecture [1].

These benefits have led to increasing adaptation of Cloud services into many businesses, which are seen as more lucrative, affordable, and astonishingly reliable alternatives compared to conventional data center based in-house services. Nevertheless, drawbacks of Cloud computing paradigm are also surfacing due to various concerns such as lack of standardized service interfaces, protocols, and data formats are susceptible to vendor lock-in [2]. These problems, although seems to be outweighed by the advantages Cloud computing provides to most small businesses, can lead to underinvestment, an economically inefficient consequences in the long run, and hence requires immediate attention.

1.1 Developer Perspective of Cloud

Cloud computing as a paradigm has not only provided immense advantages to businesses from profitability point of view, but it has also made the life of developers considerably easier with respect to many concerns. Outsourcing the hectic and time-consuming work in different levels has enabled them concentrating more on building more feature rich and quality software. While there are three principal service models in the Cloud stack, namely IaaS, PaaS, and SaaS, platform abstraction or PaaS has always been qualified as the layer, which mostly eases the life of developers. PaaS abstracts almost all the services at platform level that application demands. As a developer, it is just a matter of executing APIs (mostly REST based) provided by the particular PaaS provider to get the required service. Consequently, PaaS-aware

solutions allow developers to take advantage of reduced complexity and achieve a better time to market figure, also being able to benefit from the auto-scalable features of Cloud infrastructure [3].

1.2 Problem Statement

While it is quite apparent that PaaS provides many advantages and enables both developers and operations teams work in tandem to build better software, there is a lot of speculation about the growth and adoption of PaaS [6]. In fact, according to a study published in Gigaom [8], "*in 2015, with a combined market of @ $73 billion, most of the growth will occur in the IaaS and SaaS space, with PaaS only progressing about $2 to $3 billion*". There are many contributing factors affecting this slow adoption to PaaS over other service models. Among many challenges discussed by David [7], the threat of *data and service lock-in* and *the degree to which the underlying IaaS abstracted by the PaaS provider* are the most contributing factors. While the threat of lock-in is being addressed through the container based virtualization approaches like Linux Containers (LXC) and Docker, latter challenge still remains as a partially answered question.

Although much research work is happening in the Cloud spectrum, still the IaaS transparency in PaaS offerings becomes more questionable when compared to the advantages it provides. In my current assignment where we developed a fully SaaS application on top of Windows Azure, we experienced some adverse consequences due to this very fact. Among which service unavailability and service unreachability are the most commonly experienced issues we faced over the last couple of years [36, 37]. We, as developers, train our mindset on PaaS guarantees, could not simply apply workarounds in some situations, which cause massively negative impact on our SaaS consumers. Most of the issues that we faced up to now can be boiled down to one simple observation. That is the tight coupling of PaaS users to a specific PaaS provider, which itself is tightly coupled to a set of libraries/middleware and IaaS(es).

It is imperative to address this problem to realize the full benefits of PaaS. There exist solutions like Jelastic [29], OpenCloudWare [27], and PaaS Manager [20] which provide some level of transparency in the form of a REST interface which abstracts the underlying platform-specific implementations, thus by getting rid of tight coupling to a particular PaaS provider. However, since the application now has to depend on these

hosted environments, they introduce another layer of coupling as well as the risk of single point of failure. What we really need is a thin abstract layer which can provide a simple unified API which hides the underlying provider so that if it is required to migrate to a different service provide, it will not be a total rewrite of the data access layer so that the transition would be smooth.

Therefore, the problem that this research attempts to address can be stated as:

How to aggregate multi-cloud PaaS services to enhance availability?

Given that it is relatively easier to connect to many PaaS providers, we focus on the sub-problem of how to seamlessly switch among different service providers when it is required to provide an uninterrupted service. From the developers' point of view, the proposed solution should not be tightly coupled to implementations of each PaaS provider. Hence, the proposed solution should enable developers to utilize a standard API to access platform services without worrying about vendor-specific implementations. Whereas from a business point of view, there should not be a profit loss due to prolonged downtime of the system due to the interrupted service of a particular PaaS. Hence, the proposed solution should be free from potential single point of failure, as well as should not hinder application performance such as throughput and latency.

1.3 Objectives and Output

The main idea of this research is to come up with a way to answer the aforementioned issue that is prevalent in individual PaaS environments. In essence, at the end of the research, the choice of selecting PaaS providers will be given to PaaS consumers. Ideally, there should not be any tightly coupled PaaS dependencies in a SaaS offering. The objective is to provide a *thin* layer between the PaaS and SaaS layers, which will enable the developers to utilize a standard API which abstracts the underlying vendor-specific implementations and at the same time providing necessary configurations to switch among different PaaS providers when and if necessary. Another key objective of this research is to make this layer as thin as possible, i.e., not to make it another tight dependency to SaaS. Making SaaS providers heavily depend on this layer may cause adverse outcomes.

Due to the vastly different types of services provided at PaaS level, it would be impractical to support each vendor-specific service available at platform level. Therefore, we focus on the following main PaaS services that are required by all applications:

- *Database* as a Service (DBaaS) – DBaaS is a Cloud-based approach to the storage and management of structured data. DBaaS provides a flexible, scalable, on-demand platform that is oriented toward self-service and easy management, particularly in terms of provisioning a business' own environment.
- *Static Storage* as a Service (SSaaS) – SSaaS is an on-demand storage service that can be used to reliably store application content such as media files, static assets, and user uploads. It allows users to offload your entire storage infrastructure and offers better scalability, reliability, and speed than just storing files on the file system.
- *Cache* as a Service (CaaS) – CaaS will allow multiple applications to access managed in-memory cache instead of slow disk-based databases.

Conceptually, providing support for other types of services will be the same. The reason for selecting the three above-mentioned services is based on how widely they are used among the other types of PaaS services. *Compute* as a service is another very widely used essential PaaS service. However, as we found out during our literature survey, high availability and uninterrupted service at compute level is being achieved predominantly through containerization. Technologies like *Docker* [26] together with container orchestration providers like *Kubernetes* [28], *Jelastic* [29], etc., are market leaders, which provides uninterrupted services at compute level.

1.4 Outline

The rest of the document is structured in the following manner. Chapter 2 contains the literature review, which covers the theoretical aspects of different types of Cloud technologies, container-based virtualization, Cloud services aggregation at different layers, as well as relined work on this area. Chapter 3 presents the high-level architecture of the proposed library, which enables SaaS application developers to develop applications without worrying about the underlying vendor-specific complex

implementation to achieve uninterrupted services by deploying their applications across different PaaS layers. Chapter 4 presents evaluation of our proposed approach using a real-world application. Concluding remarks, research limitations, and suggested future works are discussed in Chapter 5.

CHAPTER 2
LITERATURE REVIEW

Chapter 2 is organized as follows. Section 2.1 describes the basics of Cloud Computing, its evolution as a utility computing paradigm over the time, different service models in Cloud Computing, the concept of PaaS, and threats to PaaS. Section 2.2 focuses on container-based virtualization, several popular container-based implementations in the Cloud like Docker, and container orchestration platforms like Kubernetes and Jelastic. Section 2.3 describes the concept of service aggregation with respect to different service models in Cloud Computing.

2.1 Cloud Computing

Companies, individuals, startups, etc., who possess big ideas to streamline their businesses and drive sales through the roof. But to get there, those ideas need business and enterprise applications come from giant application providers like SAP, Oracle, Microsoft, Apple, etc. These applications were used to be really expensive. Other than the massive price tag, behind each of these applications, there is a world of other complexities as well. They need dedicated data centers with office space, power, cooling, bandwidth, networks, servers and storage, and a complicated software stack, which requires a team of experts who can configure, install and run them. Users require an internal IT department to customize this software, testing and staging environments, production failover environments and then when new versions of these software come out, users would upgrade and that might bring the whole system down due to incompatibilities. When users need to get these things done for dozens of applications for enterprise needs, it is easy to comprehend that why big companies and enterprises need large and dedicated IT departments in-house.

Cloud computing is a better way to run businesses. Instead of running applications on one's own datacenter, they run in a shared datacenter. What is needed is a plugin, like a utility, which enables especially smaller businesses to start business with the minimum capital expenditure. Furthermore, it reduces operational expenditure as users are paying only for what they use. Applications delivered *on demand* as services over the Internet and the infrastructure which comprised of hardware and system software, can be referred to as "Cloud computing" [2]. Businesses are running all kinds of apps these

days, including custom-built apps. The main reason behind this is that you can be up and running in just a few days, which was nearly unheard of with traditional business software couple of years back. They cost less because users do not need to pay for dedicated IT department, office space, and software product dependencies to run. Thus, it turns out they are more scalable, more secure and more reliable than the vast majority of traditional apps out there in the market.

The basic reason why Cloud providers are able to give massive scale advantages to businesses is because they are built based on an architecture called *multi-tenancy*. With a multi-tenant app, there is not a copy of the app for each business using it, it is one app that everyone shares but it is flexible enough for everyone to customize for his/her specific needs. A very good analogy is a giant office building where everyone shares the infrastructure and services like security. However, each office can customize their respective office space. This means apps are elastic, they can scale up to tens of thousands of users or down to only a few. Upgrades are taken care for you, so your apps get security and performance enhancements and new features automatically. The other important factor worth mentioning here is that the way you pay for Cloud apps are very much different. When a user's app is up there in the Cloud, user does not buy anything at all. It is all rolled up into a predictable monthly subscription. In business terms, Cloud apps do not eat up one's IT resources, instead business can focus on projects that will really impact the business, like deploying more and more apps.

Cloud computing has evolved considerably during the past years into an alternative for many companies as a means of hosting and maintaining their enterprise applications [9]. Most public Cloud providers follow a *utility computing* business model, which allows Cloud users to use Infrastructure as a Service (IaaS), Platform as a Service (PaaS) or Software as a Service (SaaS), services that are provided on a pay-as-you-go basis [2]. The technical and financial models adopted by cloud computing have awakened the interest of many companies that have seen in Cloud computing an alternative to maintaining private data centers and a means of reducing costs. Clouds allow companies to externalize their system maintenance processes as well as providing them with a scalable solution regarding the computing resources they may require at any given moment and the costs of using these resources. Figure 2.1 shows different classes of utility computing, where service providers fall into different areas in the spectrum

depending on the flexibility at the infrastructure level and their level of abstraction presented at developers who mainly work at the platform level.

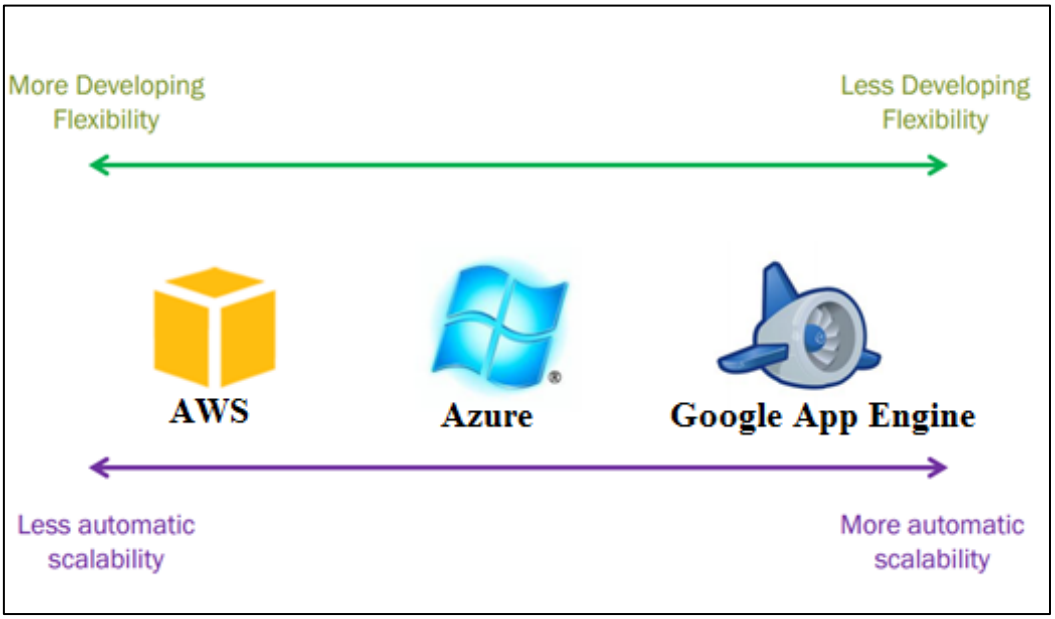

Figure 2.1: Different classes of utility computing.

2.1.1 Cloud Service Models

Higher the responsibilities you push towards Cloud service provider, less the control you have over application security, customization, etc. However, the good thing about this approach is that, because the service provider has a better control, they will provide better scalability, disaster recovery, less complexity on development cycle thus faster time to market. Figure 2.2 shows a subset of the different components within each service model along with who has responsibility for those components.

With *SaaS*, only flexibility that the consumer gets is just altering various configurations. Other than that, SaaS consumer has very little control over the Service Level Agreements (SLAs), maintenance cycles, infrastructure, etc. The main advantage is that the time to market is very less, hence the consumer can quickly be up and running with the application, thus does not have to worry about all the underlying infrastructure or platform level upgrades, etc. All these things will be taken care of by the service provider. SaaS providers will make sure that the technology updates, support for various kinds of devices, and all other platform level functionality will be available for you as and when required [10]. Famous examples are Google Docs and Human as a Service (HuaaS).

With *PaaS* however, the consumer will be given much more flexibility over the application they develop, although they do not have to manage and take care of the underlying infrastructure components like hardware, operating system, database systems, programming stacks, etc. Instead, PaaS consumers can focus on building better software by utilizing those robust platforms provided by the PaaS provider. The downside is that the developers must work within the constraints of the platform, which may not be optimal for high performing architectures.

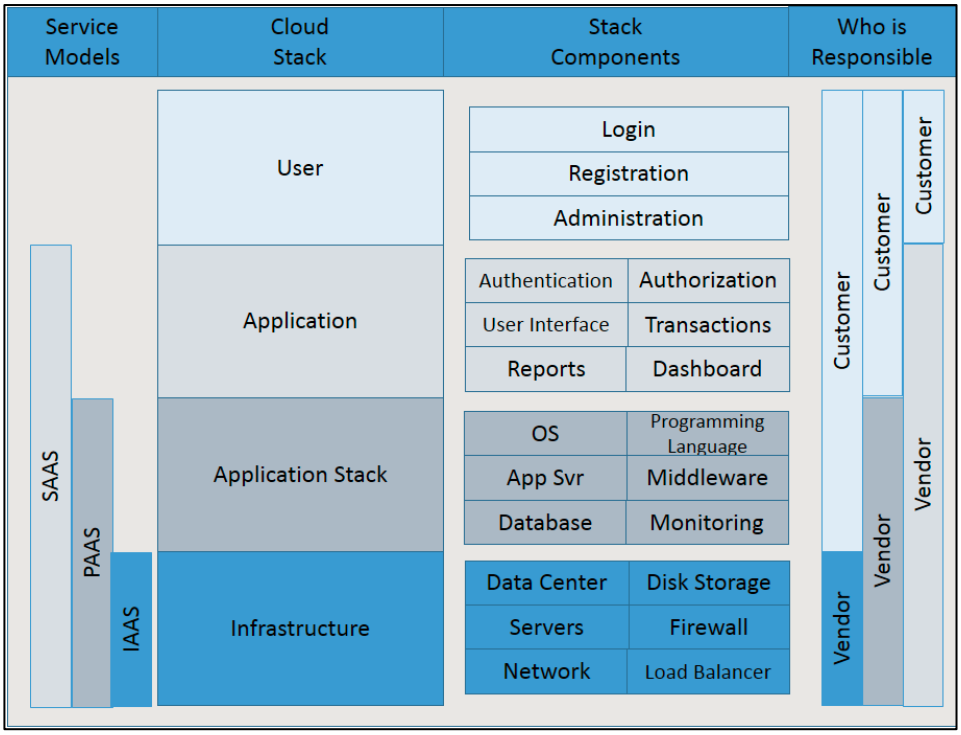

Figure 2.2: Cloud service models and their responsibilities [23].

Another disadvantage is that the consumer is highly rely on the SLAs of the PaaS providers. Some of these PaaS providers like Windows Azure run on top of Windows-based infrastructure, an IaaS provider. When Azure is facing service outages or any other issues, the developers are at the mercy of PaaS providers to stay highly available. When the PaaS service goes down the developers are mostly helpless and must wait until the PaaS provider restores services [10]. Famous examples are Microsoft's Azure, Google's App Engine, SalesForce' Force.com, and Amazon's Elastic Beanstalk.

With *IaaS*, consumers can control almost all the services provided from infrastructure level except the hardware (indeed, at least for some extend you can decide which

hardware configurations you need). IaaS service providers will provide the infrastructure services as a collection of APIs over the Internet, which will enable the IaaS consumers to spawn up IaaS level services within minutes. Application can be built to scale on demand as the workload fluctuates, thus optimizing the infrastructure usage. The downside is the consumer is constrained to a subset of virtual Cloud servers. Some applications require very specific hardware requirements, which may not be available from the Cloud service provider [10]. Typical representatives of infrastructure services are Amazon's EC2 and Amazon's S3.

2.1.2 Platform as a Service

While Cloud computing provides quite a lot of advantages, Platform as a Service (PaaS) stands out as a game changer for modern application development [4]. It enables businesses to foster rapid application development by outsourcing lots of configuration work to PaaS provider, which seems very vital in a world of complex and immensely scalable and distributed systems. As IaaS is a great way to introduce Cloud computing for making the IT operations of a business more scalable and efficient, private Cloud service delivery is in no way limited to IaaS. This is where the PaaS becomes handy. As PaaS abstracts the underlying infrastructure layer, it does not matter how complex and secured the infrastructure is. This important factor itself is providing a significant opportunity for driving maximum value even from private Clouds.

PaaS provides IT organizations with significant benefits such as [5]:

- Capex is minimized - Improved time to market

  Because of the availability of several automated tools and technologies, developers can quickly design and deploy Cloud-aware applications with very minimum capital expenditure. This encourages many startup initiatives to blossom, as the developers with innovative ideas for new Internet services no longer require the large capital outlays in hardware to deploy their service or the human expense to operate it.
- Ability to access services available in any Cloud

  As the services built on top of PaaS platforms are available from anywhere, it is very easy to customize, extend and integrate software as a service offerings from public Cloud providers.

- Legacy application re-engineering and re-architecting
  As developers do not need to worry about the infrastructure anymore, it enables the businesses to re-architect the legacy applications to integrate existing applications with the new entrants which can cut IT operational costs, increase agility in business, broaden the reach to customers and more importantly enable developers to focus more on core competencies rather than infrastructure issues.
- Ability to address integration issues through Cloud-aware applications
  In a situation where a particular SaaS depends on the hybrid Cloud environment, as the infrastructure is abstracted, PaaS can provide seamless integration of both the environments, which will immensely make the life of developers easier from implementation, configurations and deployment point of view.

2.1.3 High availability and resiliency in PaaS

According to the specifications of PaaS, consumers should not be worried about the high availability of the platform itself. Compared to traditional in-house application development where a considerable time is allocated to understand High Availability (HA) concerns of the underlying platform and how they may be aligned into an application's final design, running an application on PaaS does mean that the developers are freed from such concerns.

As the underlying infrastructure of the platform is hidden and abstracted from the PaaS consumers, it is important that the required and promised availability figures be maintained at service provider level. The difference between PaaS high availability and traditional solutions boils down to better contract management. As a Cloud service consumer, it is paramount that good SLA definition with the service provider is established.

In order to provide high availability of the services provided at PaaS level, different service providers follow different approaches. If we take a typical web application as an example, we can configure the website to automatically scale up or out based on different KPIs to provide high availability. We can also configure to increase the number of web server instances according to the shape of the traffic a particular web server is received. Under the hood, it will utilize the load balancer and maintain the

traffic flow smoothly across the spawned web server nodes to provide better user experience, thus a better availability.

Various Cloud service providers tackle the high availability concern in various ways. More restricted PaaS solution like Windows Azure will not let us go deeper into infrastructure level and configure the High Security (HS) zones, load balancer rules, etc. Instead, all of these concerns will be handled at the infrastructure level. However, a provider like AWS allows us to perform more granular configurations. Figure 2.3 shows how we can configure the auto scaling rules in computing resources depending on the resource utilization, while Figure 2.4 shows how we can maintain the nodes highly available in the AWS environment by enabling redundancy at different levels. It is important to notice that auto scaling at IaaS is more flexible compared to scaling at PaaS level.

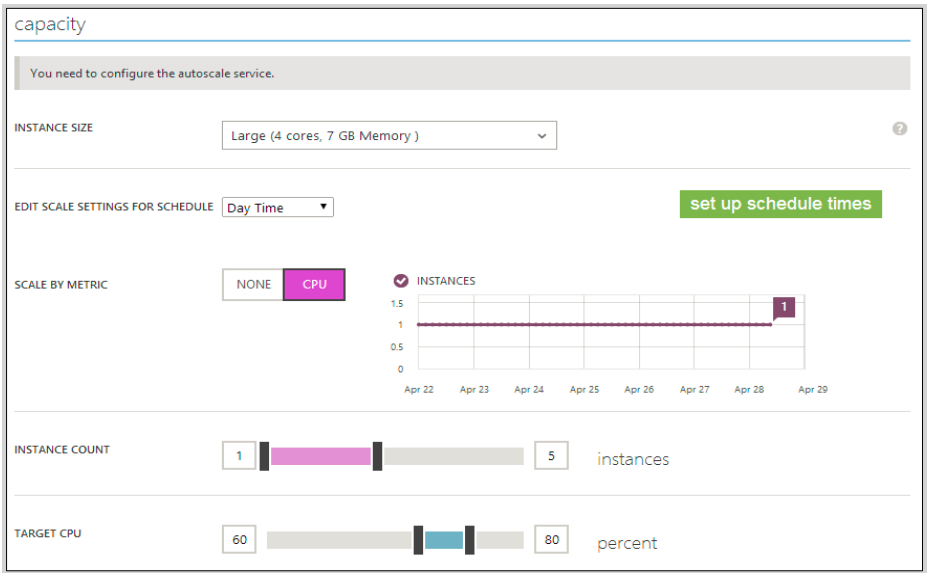

Figure 2.3: Scaling out web server nodes according to the configured parameters [24].

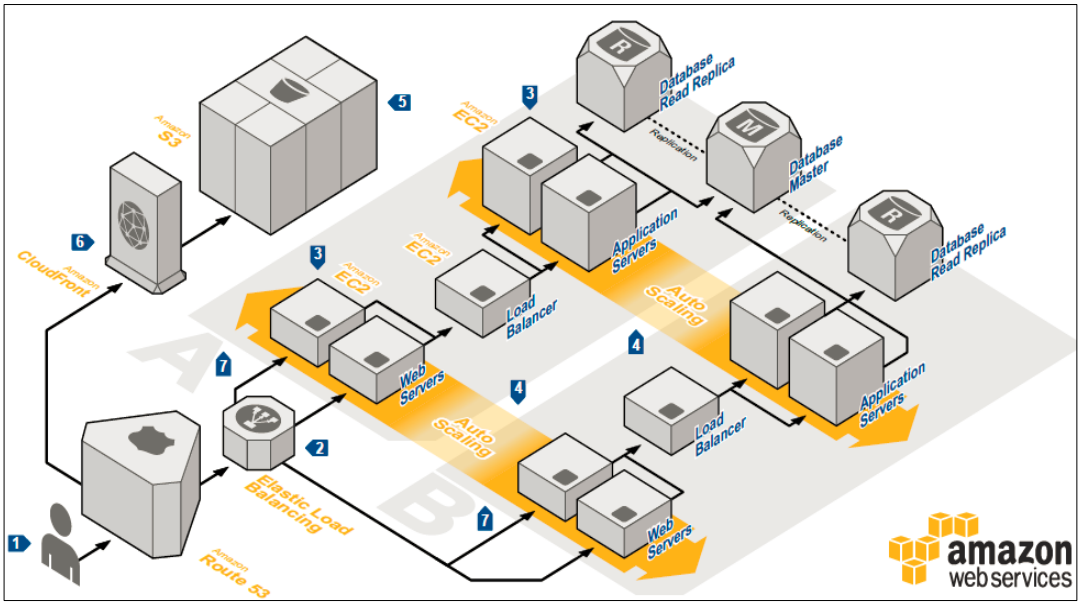

Figure 2.4: Maintaining a HA architecture in different PaaS services in AWS [25].

2.1.4 PaaS vendor landscape

Since PaaS has started to become the most lucrative Cloud service model from the Cloud service provider's point of view [11], [12], the landscape for PaaS is expanding very rapidly. Different vendors can include support for multiple languages, application services, and data technologies, as well as integration and business process management services. Some of the well-established PaaS players include [13]:

- Amazon Elastic Beanstalk
- Salesforce1 platform
- Google App Engine
- IBM Bluemix
- Oracle Cloud platform
- Red Hat OpenShift
- Windows Azure platform
- Heroku platform

2.1.5 PaaS development challenges

Although it is obvious that PaaS provides many advantages for businesses, downsides are also surfacing from businesses and developers' point of view. Given the vast landscape of PaaS providers, it is none trivial to choose one platform among them to go ahead with. Some are providing a very renown stable platform, but at a higher cost. Some are cost effective, but do not provide all the services that might be essential in the

future for an application. Apart from the selection of a PaaS provider, following are some of the challenges for PaaS when choosing a PaaS model:

- Vendor Lock-in

  Vendor lock-in has been the most alarming challenge to PaaS consumers [14]. Vendor lock-in can be due to (a) *Uncertainty of selecting an unknown technology* and (b) *Learning curve of a technology*. Vendor lock-in has occurred mainly due to the growing number of Cloud computing service providers and service offerings, especially in computing and storage services. Once a particular customer decided to go ahead with a PaaS provider, their offerings tie the user to a specific technology and protocols, which cannot be switched or replaced without significant switching cost.

- Interoperability

  In a highly scalable and distributed environment, it is very common that Cloud applications will have to interact with other applications that may be hosted in very different service provider. This will become even cumbersome if a particular application is aiming for a higher level of scalability and availability, where parts of the applications are deployed in several PaaS providers. Because the heterogeneous nature of different PaaS providers, this is a nightmare unless otherwise they expose a standard API based on REST or SOAP. Even so, that will make our application tightly coupled with the underlying platform it works with in a particular instance.

2.1.6 Summary

Cloud computing enables the businesses and developers to achieve considerable advantages over the traditional datacenter approach. From the main Cloud service models, Platform as a Service has become very popular with its stack of vital devops advantages over others. Since PaaS landscape is growing extensively, vendor lock-in has become an alarming issue, which prevents most potential businesses to move to the Cloud. This problem can lead to underinvestment, an economically inefficient situation, and therefore deserves our attention.

2.2 Container-based virtualization

As it goes with traditional hypervisors, which are used to achieve high level of virtualization, container based systems enables us to achieve system virtualization with high degrees of both isolation and efficiency [15]. Stephen et al. [15] showed how

container-based virtualization enable us to spawn very much secure containers, which are another abstraction of the concept of virtual machines, yet very efficient. They used Linux-VServer as the representative instance of a container-based system.

Unlike the traditional hypervisors, container-based virtualization the operating system leveling system level, instead of the hardware level. In other words, guest operating systems that runs at each container (a.k.a. virtual environment), shares a lot of the host operating system resources. This sharing itself gives the containers a great advantage over traditional virtual machines, as they are leaner and smaller than hypervisor guests. As the containers themselves are simply resources managed in the same address space by the host kernel, sharing information across the containers are also much easier and efficient.

2.2.1 Containers in Cloud environment

Containers have been considered as a cheap way of packing a hosted environment as conceptually, it is very simple way of bundling an application with everything that it depends on historically, when deploying an application. However, with the rise of Cloud computing, Cloud provider's consider containers as the best technology to achieve higher density and elasticity in a very cost efficient manner. Containers provide a unit of an execution environment for applications hosted in the PaaS. In other words, containers are closer to infrastructure (IaaS) compared to PaaS. Each application and service pushed into PaaS (to be launched and executed) requires runtime configurations, control resources and isolation between the other applications hosted in the same environment or the data center. Containers allow these features by their design. That is the reason why containers are more suitable to Cloud environment.

2.2.2 Docker

Docker is an open source container based technology (Like *LXC, FreeBSD jails, AIX Workload Partitions* and *Solaris Containers*). Essentially Docker allows developers to package up an application and all of its parts from the stack that it runs on, the dependencies they are associated with it, package it all up in this box which is called the *container*. The main idea behind Docker containers is that it enables the applications to run in an isolated environment, the application has all that it needs to run inside this container. It means that the underlying host (i.e., the OS) environment is completely

abstracted from the application. The problem that Docker solves is the "*dependency hell*".

Docker virtualizes the operating system, only our applications, and all of its dependencies are contained in one black box. This is referred to as *Dockerizing*. This

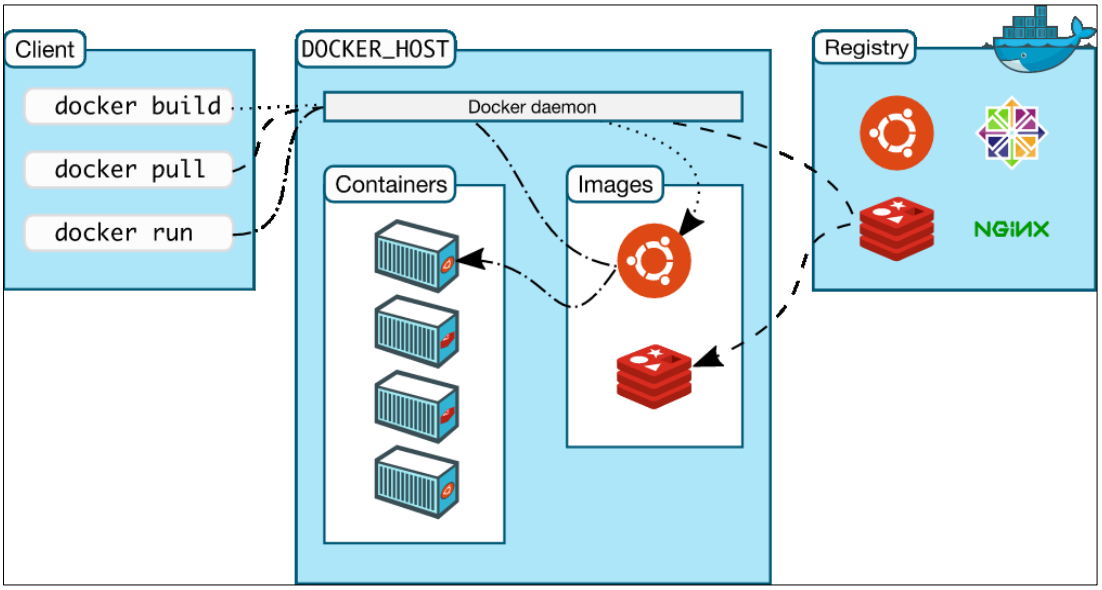

Figure 2.5: Docker architecture [26].

makes them extremely fast, portable, scalable, high density and fast deployment. The guest OS is outside of the Docker container; hence, developers do not need to worry about it. Docker is comprised with two major components. *Docker* itself, which is the open source container virtualization platform and the *Docker hub*, which is the platform for sharing and managing Docker containers. As Figure 2.5 illustrates, Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. Both the Docker client and the daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate via sockets or through a RESTful API. Docker registries hold images. You upload or download images from these public or private stores.

2.2.3 Container orchestration

Orchestration comes in handy when it comes to managing several hundreds of container nodes to scale applications in the Cloud. In a large-scale distributed setting, it is important to create a layer of abstraction that allows the developers and administrators to work collectively on improving the behavior and performance of the desired service,

rather than any of its individual component containers or infrastructure resources. This is where the container orchestration kicks in.

Kubernetes [28] is Google's approach of container orchestration, which enable handling of a large cluster of Docker containers. In a large-scale distributed environment, managing Docker nodes will become a nightmare. Therefore, Google started the initiative of building a cluster management, networking, and naming system to allow container technology to operate at Google scale [16]. Kubernetes enables automating deployment, scaling, and operations of applications containers across a cluster of Docker hosts. It is a container orchestration platform where we can use certain rules and configurations to scale applications on the fly with Docker node replication.

Jelastic [29] is another container orchestration environment which can be used to manage container nodes in a clustered environment. Through certified containers, they claimed that they could easily migrate the containers among many IaaS providers, as containers do not care what the Guest OS is (As long as the Guest OS supports the containerization). Service consumers can have many container types including Docker and Rocket. They do not expose implementation level details. However, they emphasize in their feature catalog and whitepapers that they can live migrate containers across any selected IaaS to achieve high availability. However, they do not specifically mention what are the types of services they can migrate at platform level and how they are doing it. However, they also claim that they have quite a lot of enterprise level customers who have been working on their platform to achieve a highly scalability and availability.

2.2.4 Summary

With the advent of Cloud revolution, container based virtualization has become the most feasible approach to achieve higher density and superior elasticity when it comes to spawning of virtual environments. Among the container-based technologies, Docker has become very popular lately. A large-scale distributed application with millions of container nodes requires an automated container cluster management and monitoring framework. Using a Cloud orchestration tool, we can automate the deploying and migrating of cluster container nodes, but still there are loose ends when it comes to think about our problem. As containers are closer to infrastructure than the platform, provisioning of platform level services in other nodes when migrating is not possible at

least at the container level. Even if we can achieve that in orchestration software, then again orchestration itself becomes another dependency which can intern become a single point of failure.

2.3 Service Aggregation

Service aggregation by its definition enables us to provide a common interface, which can be used to provide services across different service providers [19]. Aggregation service in the context of Cloud computing will mainly act as a hub or a dashboard which can be used to deploy different services under various service providers.

The Cloud service orchestration is a widely used concept of Cloud service aggregation, which is also studied and researched by many. Some entities have tried to provide a total eco system, which can manage several underlying Cloud providers by exposing a public subscription based API. But they are not flexible enough to let the Cloud consumers choose the underlying appropriate Cloud service provider, instead the system will automatically provision appropriate instances to get the work done. All what Cloud consumers are supposed to do is to setup and configure the parameters to scale in and out underlying services [17].

Some entities have tried to expose a web-based Cloud service aggregation API and service, which can be used to manage and monitor the underlying Cloud services [4]. All what this service does is providing a unified standard API which can be used by the Cloud consumers to get the services via a public hosted REST API. Apart from the API, they have also provided some value added features such as monitoring, information services, etc. on top of the hosted service. As they have the full control over what kind of traffic flow through their API from clients, they could easily gather that information and provide a monitoring interface, which may be very useful for audit purposes for their service consumers. The downside of this approach is simply the single point of failure. As this type of a hosted solution will impose a single point of failure to their subscribers, they need to make sure that it is up and running all the time, which is a very subtle challenge as they are also hosting this service on top of an existing IaaS provider like Amazon. Cloud service brokering does look very similar to this approach, which shows more or less the same characteristics. In this section, we will be looking into several approaches, which try to provide Cloud aggregation implementations [18].

2.3.1 IaaS Aggregator

Lee at el. [19] developed an abstraction model that will act as an aggregation layer on top of various IaaS providers to provide services at the infrastructure level. With the alarming rate of IaaS market entrants, they wanted to provide a uniform common interface and a description of IaaS services across multiple IaaS service providers through their interface. The issue they are trying to address is the provider lock-in, a.k.a. how easily can you distribute data from one provider to another [19]. The authors supported various services from the two market leaders in IaaS at that time, AWS and GoGrid. They were able to provide an IaaS proxy, which acts as a hub over the vendor-specific infrastructure. As of today, they are able to provide IaaS proxy support for AWS, GoGrid, and other service providers whose interfaces are compatible with AWS or GoGrid. They have also provided a management interface where users can monitor the resources being provisioned, their health status, etc.

IaaS aggregator has been prototyped to manage multiple IaaS resources from multiple providers [19]. This clearly backs up the problem that we are trying to solve. Although there are multiple service providers available at platform level, as most of the services they offer show a common pattern, they can be provisioned from a common interface similar to what Lee et al. [19] proposed for infrastructure level. As they have provided a web interface, they could easily track the service API calls and collect the usage, traffic information that reach their services which enabled them to provide a management interface to monitor the resources available for their customers.

However, at the meantime, they really do not provide a solution for the reliability issue that they were trying to solve in the first place, as the IaaS failure problem is now passed into their layer resulting in a single point of failure. At IaaS level, if the IaaS aggregator is down, customers can easily login to the respective IaaS provider's management portal and check out the services. This is not desirable for customers, could lead to state management problems (once the proposed layer comes up again), and it is not possible at platform level as the underlying infrastructure is abstracted from the PaaS consumer.

2.3.2 OpenCloudware

OpenCloudware is an open source initiative funded by French authorities aiming at providing an ambitious objective of enabling large-scale enterprise distributed applications to be deployed on any Cloud with minimum code level changes. OpenCloudware allows distributed application developers to think and model them using several virtual environments to assemble them, build, deploy and operate them with the PaaS layer, which will be aggregating multiple IaaS services, but the whole process is IaaS-agnostic [4].

The authors have come across many challenges when providing the support for multiple IaaS support for various PaaS services. One key challenge they emphasize on is the *"management of the lifecycle of applications across different Cloud service providers"* [4]. Under the same challenge, they also show some other key areas that should be addressed such as:

- How to hide the technical heterogeneity between the components?
- How to deploy whatever the application technology and the execution infrastructure automatically?

These are some of the valid questions that even our solution needs to address. Even though we are not providing a hosted solution that should work in a distributed manner, hiding the vendor specific complexity, heterogeneity, and automatically deploying services across different IaaS providers are two of the biggest areas of our research.

OpenCloudware implements a model called vApp as the building block of the PaaS layer, which is based on an extension of the standard Open Virtualization Format (OVF). vApp describes the node types of the virtual application, different nodes of it, the underlying relationship between the nodes and also the SLA that the end user would like to have. This descriptive model can then be used to spawn instances of the underlying IaaS to provide the PaaS-level services.

They are able to provide support for various commercially available IaaS services like Amazon, Windows, GoGrid and almost all the popular open source counterparts. As this is an open source initiative, developers can also integrate it for new IaaS providers by implementing their Federation proxies. The multi-PaaS controller will act as an API

and aggregate the IaaS provider services and deploy the vApps seamlessly according to the way we configured them.

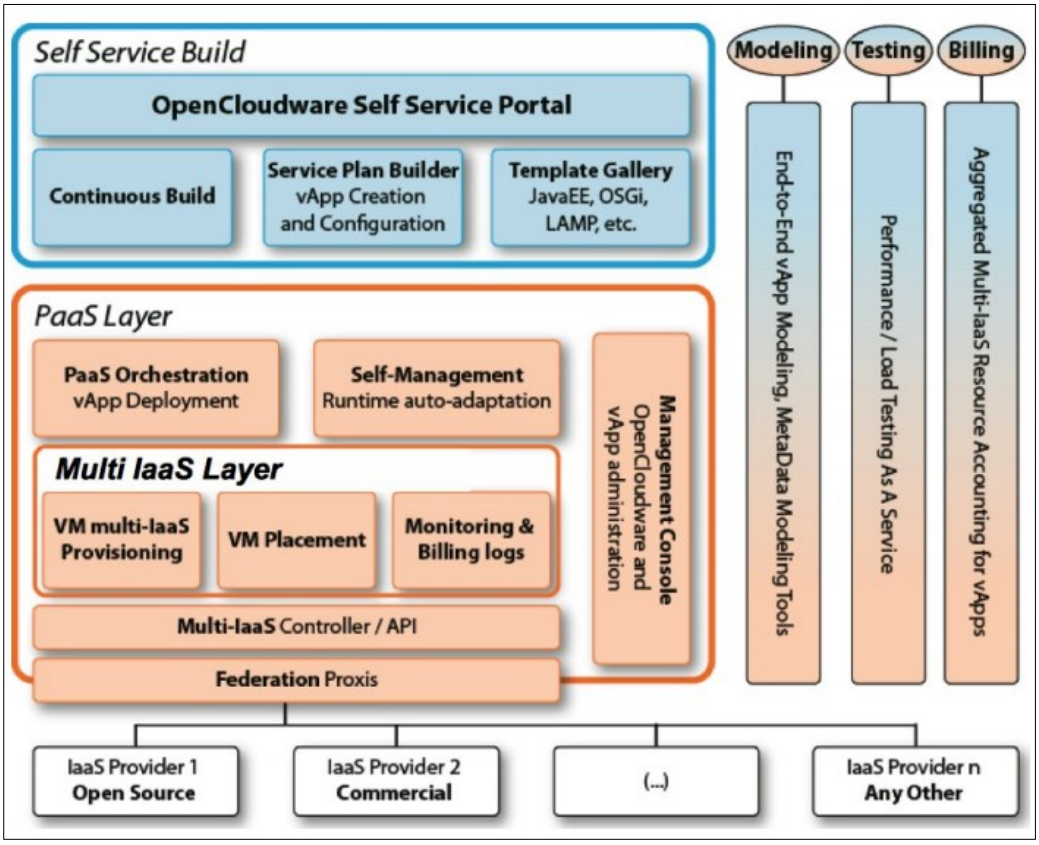

Figure 2.6: OpenCloudware high-level architecture [27].

The OpenCloudware project has a large scope and is still in development. However, certain parts of the projects are already available as prototypes. As the scope is so vast, they could not find a way to connect everything as a production environment.

2.3.3 PaaS Manager

Similar to the IaaS Aggregator discussed previously, David, Neves and Sousa [20] proposed to provide an aggregator service in PaaS. The authors attempt to provide a common interface to unify the information and management processes of applications created in PaaS environments. Their research outcome takes the form of a PaaS API aggregator, which aims at providing a solution to vendor lock-in syndrome. The authors demonstrate the proposed aggregator by connecting CloudFoundryarket leaders in PaaS, namely Java-based PaaS CloudBees, VMWare's CloudFoundry, Tier 3's Iron Foundry and Salesforce.com's Heroku.

As shown in the Figure 2.7, Range of services provided by PaaS Manager can be categorized into two groups [20]:

- Management Services: *Creating* and *managing* applications and databases. *Migrating* applications between vendors, if feasible.
- Information Services: Acquire information concerning applications and databases, Monitor applications in real-time.

PaaS Manager follows the REST-based approach. However, there is no seamless failure monitoring/handling mechanisms built into their API manager. They anyhow have included an operation called *Migration of applications among PaaS vendors*, this is when we, as developers, decide that we should move our PaaS service provider from A to B for a particular service, we can ask the PaaS Manager to check the feasibility of doing it and continue with it. Further, this migration is supported only for Applications hosted and deployed from GIT repositories. It simply creates an application in the new vendor and ask the new app to create a GIT hook and get the application deployed. This migration approach is something we may adapt to our research as we are clearly looking into failure recovery. Next, we discuss several related works on the PaaS Manager.

2.3.3.1 Cloud Service Broker

The Cloud service broker is a framework to achieve a harmonious integration between different services provided at Infrastructure and Platform level [18]. The framework comprises of three main components:

- IaaS Manager (IaaSM) – An external entity that could help distressed the workload by brokering the relationship between the Platform and Infrastructure layer. IaaSM provides a seamless API translator, which consists of three key components (*IaaS Discovery Interface, IaaS Management Interface, and IaaS Functional Interface*). This layer does the same thing as PaaS Manager, but at Infrastructure level.
- PaaS Manager (PaaSM) - This provides a common interface to unify the information and management processes of applications created in PaaS environments. This is the component that we are interested in.
- Cloud Service Broker (CSB) – This is the most important component of this framework as far as the research is concerned. CSB interconnects IaaSM, PaaSM, and other user interfaces via a service bus.

As a proof of concept, the authors are developing a prototype of the CSB and PaaS Manager working together in tandem to complete the whole architecture.

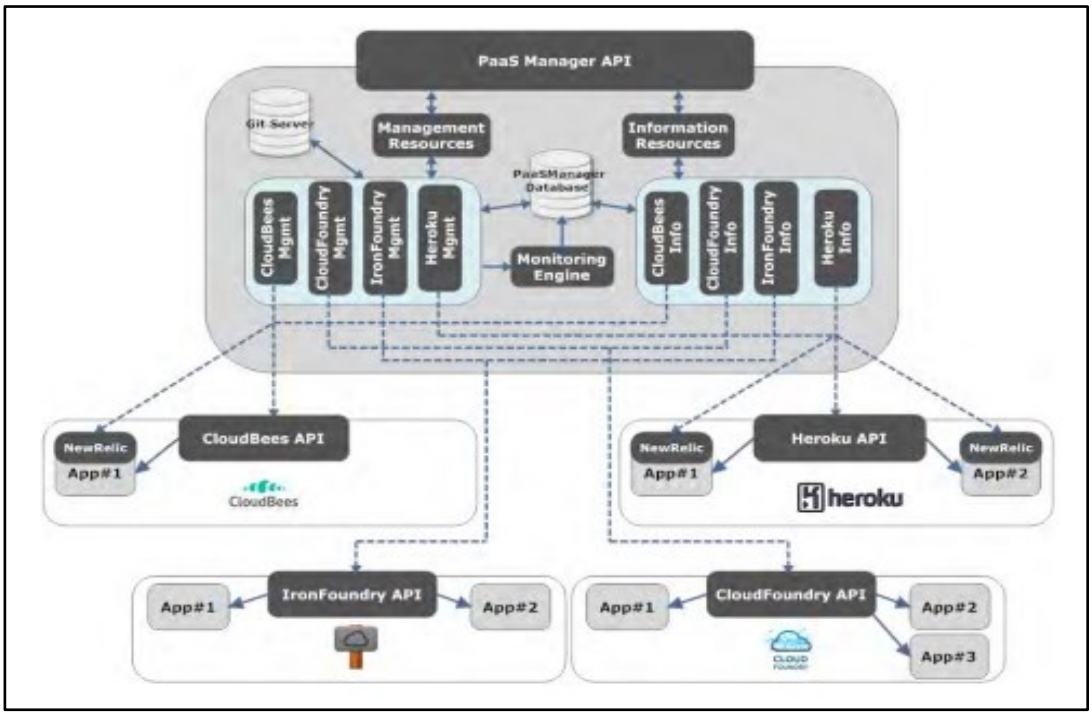

Figure 2.7: PaaS Manager architecture [20].

2.3.3.2 Interoperability and portability of Cloud services enablers in PaaS

In [21], the authors propose a distributed architecture that allows developers to create and expose services through a *Service Delivery Platform (SDP)*. They make use of the flexibility and scaling capabilities of the Cloud to enable service developers to make use of different platforms without worrying about the vendor-specific implementations. Their Cloud enabled SDP architecture, exposes a standardized API through the PaaS Manager so that the service developers can simply choose the best possible PaaS offering, which yields best possible performance and other requirements without worrying about vendor specific lock-ins.

At a very basic level, the SDP can be seen as a collection of service enablers, which are orchestrated by a Service Broker and exposed for third parties' application development in an

SOA paradigm [21]. PaaSM will make sure that API calls issued against the developer interface will be executed against the appropriate PaaS provider. They do not need to worry about the actual PaaS implementation details as the PaaSM abstracts those details into a standardized API. The authors also showed how a new service can be registered

into PaaSM and more importantly, how the services can be migrated from one PaaS to another.

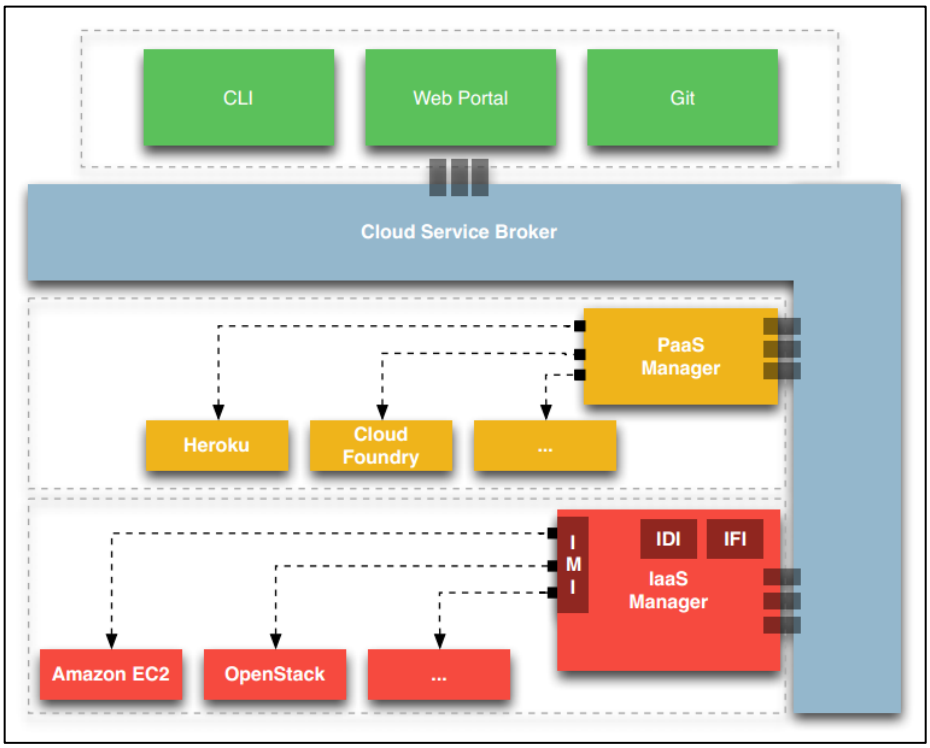

Figure 2.8: Cloud service broker Architecture [18].

2.3.3.3 Summary

PaaS Manager goes in parallel with what we are trying to achieve through our research. It answers certain parts of our research problem, but it fails to emphasize on seamless migration of PaaS services across different providers when there is a need. At the same time, PaaS Manager itself is another layer where the application developers have on, which makes it a single point of failure. However, clearly we can adapt the concepts they have been used to implement the uniform interface, which abstracts the vendor-specific implementations.

2.3.4 soCloud

soCloud is a service oriented, component-based PaaS for managing portability, provisioning, elasticity, and high availability across multiple Clouds [22]. Paraiso et al. [22] are mentioning about taking the scalability across multiple Cloud providers. Normally, in Cloud environments, we are talking about scale out and scale up (or horizontal and vertical scaling). With the enterprise level, large-scale distributed applications, applications are typically deployed across many nodes. These nodes may

reside in the same Cloud provider or across different Cloud providers. In order to support this sort of scalability, the application logic itself should not be coupled to a vendor-specific implementation. This is because the in a particular instance, the underlying infrastructure can be of any type.

soCloud discusses how to provide services for managing provisioning, elasticity, and high availability across multiple Clouds. Paraiso et al. [22] show how we can use the concept of Cloud federation to support applications deployed in a distributed-Cloud environment. Their architecture consists of two components called *soCloud master* and *soCloud agent*. Master handles node management (provisioning, deployment and replication) while the agent is where the actual application logic resides.

The authors managed to demonstrate their research with ten existing IaaS/PaaS providers, which are already available in the market, which include giants like AWS, Azure, CloudBees, Heroku, etc. They used SOA based component deployment strategy called *FraSCAti,* which has to be installed on every node where soCloud master/agent resides. An example use case deployment is shown in Figure 2.9 where soCloud PaaS provides high availability by replicating itself on different clouds. As shown in the Figure 2.9, there is one replication of the soCloud master. Then, the deployment is done in three steps. In the first step, the soCloud master is deployed in *dotCloud*. In the second step, the soCloud master (deployed in *dotCloud*) dynamically deploys another soCloud master in CloudBees. Automatically, the first soCloud master becomes leader and the second one the follower. The soCloud master leader is active, while the soCloud master follower is passive. By active, they mean the soCloud master processes the operations in the system. By passive, they refer to the standby soCloud master used as replication. One limitation of this work is the lack of support for storage, as authors focus only on computing.

If we really want to use this in our Cloud providers, we need to ask them to install all the dependencies to support soCloud architecture (Like FraSCAti, Web containers, and JVM). However, it at least shows that PaaS-level abstraction is still possible. This research is anyway aimed at highly distributed applications, which requires near perfect node deployment management and which assume every now and then node failures, which may not be the first priority of common 3-tier transaction web based application

deployed in Cloud. This research is predominantly aiming at providing a scale out solution for highly distributed applications in the cloud.

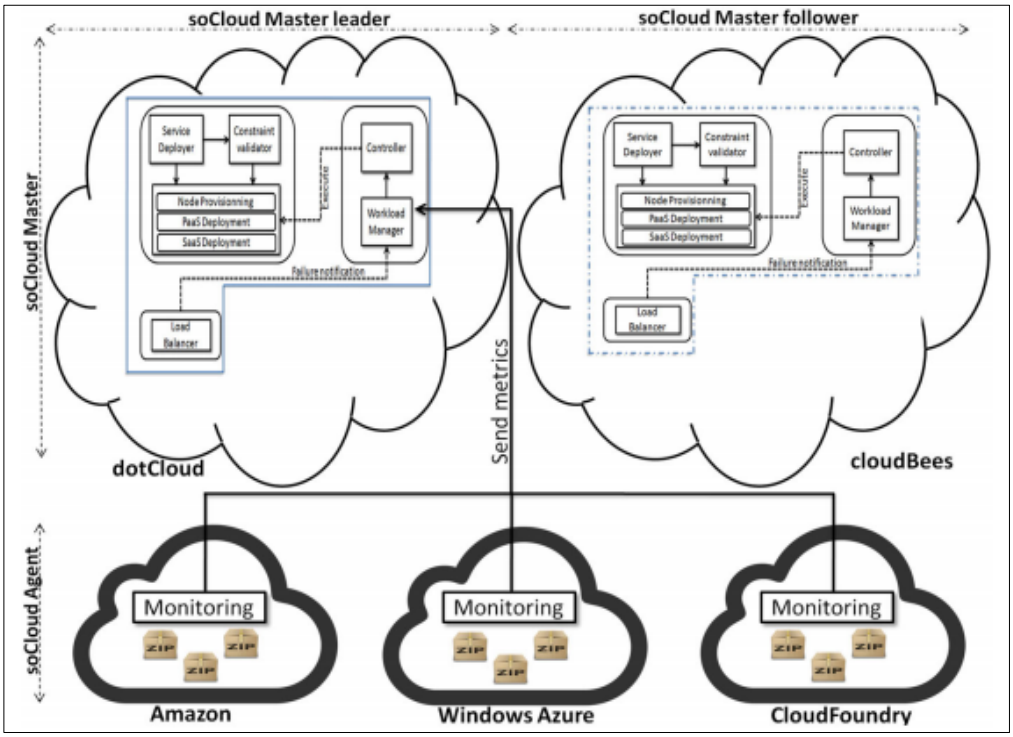

Figure 2.9: Conceptual view of a soCloud deployment [22].

2.3.5 Cloud federation

Tobia et al. [10] proposed a novel concept way back in 2011 called Federation into Cloud. The reason why this is considered as a remarkable concept is because they had this frame of mind that one day many Cloud providers will become a threat to the Cloud consumers. The authors discuss about federation at IaaS, PaaS, and SaaS levels. Although they do not provide any implementation details of a federated solution, the authors provide a high-level architecture, which can be used to provide a federated solution in Cloud. The authors also discussed different types of PaaS services available, their categorizations, and types of PaaS services likely to be federated. [10]. Our multi-cloud library architecture is inspired by this conceptual architecture. However, the authors discuss about functionalities like resource migration, resource redundancy, and combination of resource services at the multi-cloud library as the proposed library is not only meant for the PaaS layer. From the services they discussed, we are mainly focusing on the abstraction API as well as the resource migration.

If we closely look at the Figure 2.10, which illustrates the migration of a service and the impacts on the service endpoints and the thereon based application, Multi cloud library

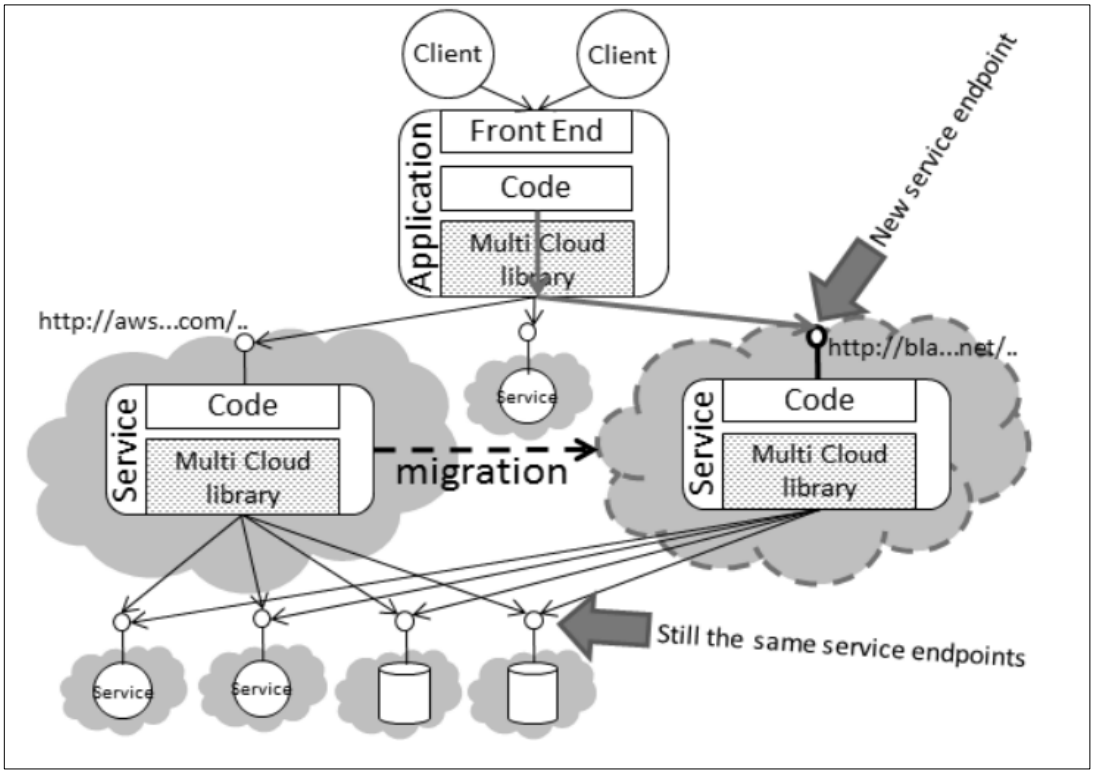

Figure 2.10: Migration scenario illustrating impact on service endpoints [10].

also follows the same design principle by providing simple configuration options to change the service endpoints. This research has played a vital role in identifying a proper convention based configuration system in PaaS Aggregator to provide the end user with a simple interface to define the underlying PaaS providers for the selected services.

2.3.6 Summary

Cloud aggregation is the only practical way to provide an implementation, which can be used to seamlessly switch among different IaaS providers when and if necessary. However, providing a hosted version of the aggregator service will inevitably make it a liability since it will become a single point of failure. Of course, if we go with that approach, as every API call is proxying through our service, we can gather lots of audit level information, thus by providing a common interface to manage and monitor the services in a web-based interface. One of the main architectural decision of the Multi Cloud library to make it as thin as possible to avoid unnecessary coupling.

CHAPTER 3

METHODOLOGY

This chapter discusses about the proposed solution and approach to implement the multi-cloud library. Section 3.1 focuses on the high-level architecture of the proposed solution. In Section 3.2 we discuss each component of the proposed architecture, algorithms, and data structures used in different components and how they contribute to different aspects of the library.

3.1 High-Level Architecture

PaaS Aggregator will be a part of the application itself, i.e., the application will utilize the functionality provided by the PaaS Aggregator, instead of vendor-specific SDKs to access the platform services. PaaS Aggregator will provide a standard API, which will abstract the platform-specific implementations. These standard API calls will be translated into appropriate vendor-specific API calls depending on the configurations and the availability of the platform. This architecture is inspired by the concept described in [10] such as providing a resource API to access underlying services in a unified manner and also a management API to safely migrate consumers among service providers. It also discusses how important it is to take the redundancy services out of the application logic to provide better interoperability in the application architecture. Since this can only be done by getting rid of vendor-specific implementations from the application, the multi-cloud library can be taken as the placeholder of these dependencies.

High-level architecture and the integrating modules that support the defined operational processes of the proposed PaaS Aggregator is illustrated in Figure 3.1. When Service 3 cannot access its primary configured PaaS provider (i.e., PaaS Provider 3), it automatically switches to PaaS Provider 2 to fulfil the request. Services shown inside the PaaS Aggregator are the service APIs, which are used to access the underlying PaaS services from the respective providers. At the same time, it uses a log (depending on the service) to store the subsequent updates so that when the primary provider comes online, it can replay the missing updates to sync the data stores. Synchronizing process may take place asynchronously as well, depending on the availability requirement.

The proposed library will be responsible for:

- Routing API calls based on the currently selected and active PaaS environment.
- Switching the PaaS provider for a particular service, if it is not accessible in the currently selected provider due to service outage or service unreachable issues.
- Providing different kinds of failover mechanisms such as active-active and active-passive. This is important as in active-active scenario, the Cloud consumer should always maintain an active service instance which is costly. To provide an active-passive setup we plan to use an update-based logging mechanism, which can be used to recover the services up to the position where the services can be resumed.

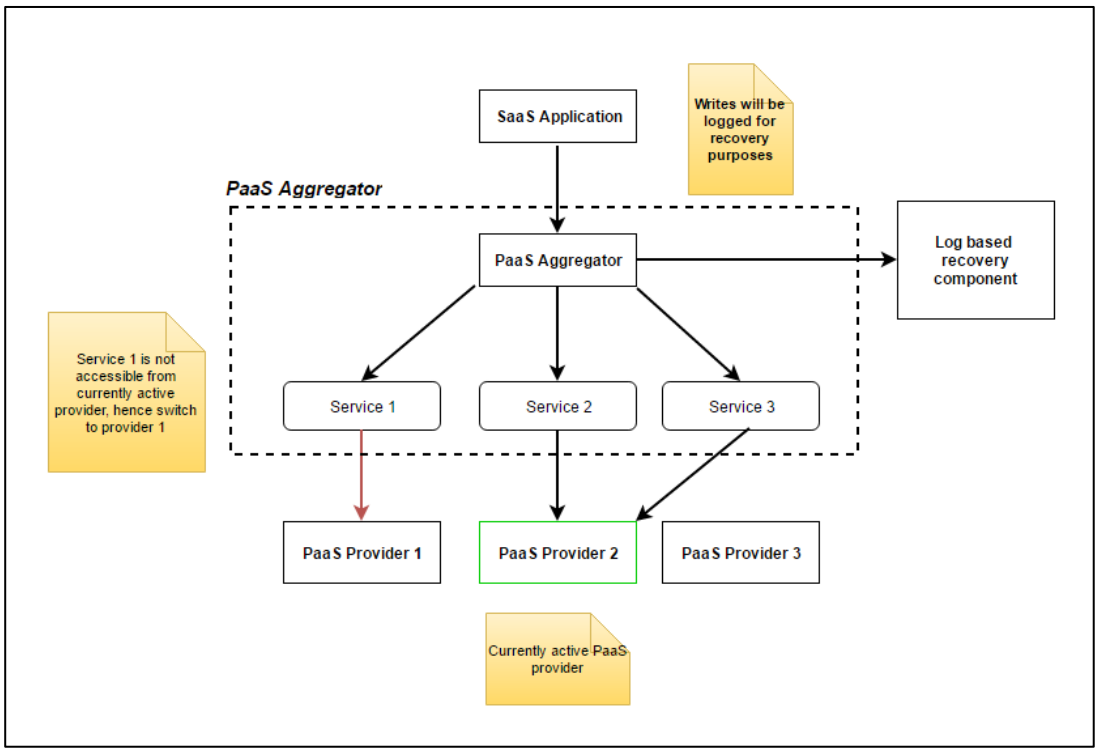

Figure 3.1: Proposed high-level architecture.

Proposed architecture illustrated in Figure 3.1 cannot be used for migrating compute services (e.g., Web sites and scheduled workers) across different service providers. Because the PaaS Aggregator is an integral part of the service layer of the application, this architecture assumes that the environment it resides is available all the time. For the services we chose to demonstrate the effectiveness of the PaaS Aggregator, we can efficiently use this approach. PaaS Aggregator will be implemented by wrapping different SDKs in a library with well-defined simple interfaces. There will be endpoints defined in the framework to extend the functionality with regards to multiple services

in PaaS and also multiple vendor support. Depending on the consistency and availability level that the application demands, we can use the secondary provider (shown in Figure 3.1) with a low hardware configuration to reduce the cost. For example, if the secondary is supposed to be used as a read-only service in case of a sudden failure in the primary service, we can use a secondary service with relatively lower specification. If a secondary is not required at all, only a primary can be provided in the configurations, with the objective of extending to multiple services for better availability in the future.

3.2 Solution Overview

The PaaS Aggregator architecture has a modular design that allows the entire system to remain fully operational even if some PaaS vendor API is not operating correctly. Although the context of this research is to provide the ability to access services in Azure and AWS, they are not the only two PaaS providers available in the market. Therefore, an important design principle that is visible all across the PaaS Aggregator architecture is the room for extensibility. As shown in Figure 3.2, PaaS Aggregator solution is comprised with four main components, namely PaaS Aggregator Core, PaaS Aggregator Cache, PaaS Aggregator Database, and PaaS Aggregator Storage. PaaS Aggregator Configuration Provider is responsible for defining the configurations of the selected PaaS providers. These modules are expected to be exposed as nuget packages [38] so that only the required dependencies can be easily downloaded and referenced. Depending on the service accessed, relevant vendor-specific SDK will be used to access the underlying service. Next, we discuss the details of these four main components.

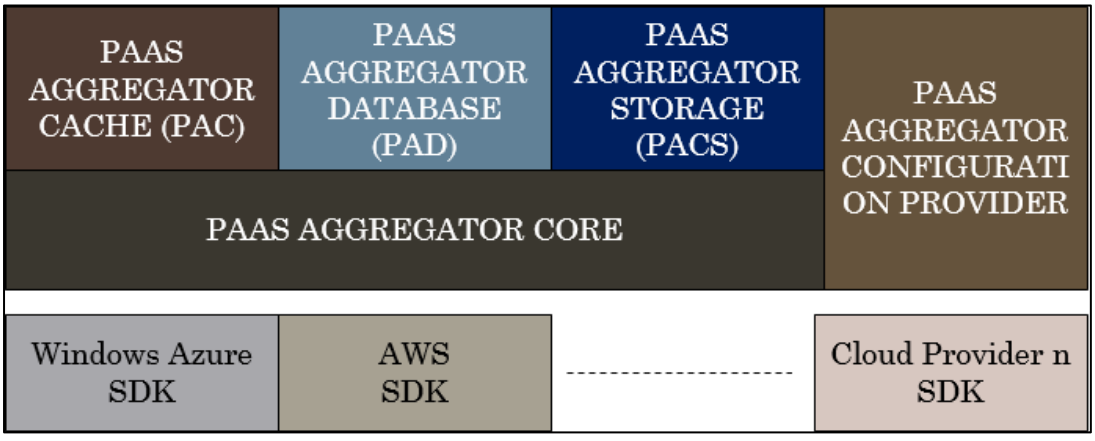

Figure 3.2: PaaS Aggregator – components.

3.2.1 PaaS Aggregator Core

As the name implies, this is the core of the PaaS Aggregator library. To utilize any of the API methods provided by PaaS Aggregator, this component has to be referenced. This component provides the following two main functionalities:

- Facades

  PaaS Aggregator adapts façade design pattern [39]. Not only the service APIs that abstract vendor-specific implementation, but also all the extensible library functionalities should be exposed as facades in the core. The main idea to separate out the service API facades from the respective services themselves is to provide a logical interface to extend the PaaS Aggregator to support additional service providers. Details on how to extend these facades to provide support for other PaaS providers are discussed in respective sections where the service APIs is detailed.

  Log service storage accessor interfaces are also included in the core. Log service storage accessors are abstracted into a façade so that the service consumers could decide what kind of underlying log storage provider to use depending on the service they are consuming in PaaS Aggregator. For example, a NoSQL storage backend can be used as the logging provider for the database service in PaaS Aggregator. It is a matter of providing storage method implementations according to the façade and inject it into the appropriate factory to resolve the service. PaaS Aggregator will make sure that the injected storage backend will be used (if provided with). Logging will be extensively discussed below in respective sections where the service APIs is detailed.

- Configurations

  PAC also parse the PaaS provider-specific configurations and injected them into specific services used in the system. PaaS Aggregator expects the configurations are placed in a JSON file in accordance with the given format in the root directory of the currently executing assembly. The skeleton of the configuration file is shown in Figure 3.3 and the file should be named as config.json.

  Configuration root object contains three array variables called *caches*, *databases*, and *storages*. As the name implies, each variable holds configured

service provider credentials and other information that is essential for the PaaS Aggregator to communicate with service providers. There has to be one primary provider configured for each service which will be given the highest priority when trying to fulfil a request. Each configuration is discussed in detail under each service API section, respectively.

In our Proof of Concept (PoC) implementation all of this sensitive information is kept in plain text format. They should be encrypted using some kind of an encryption mechanism or to utilize a different mechanism such as *Azure Key Vault* [30].


```
{
   "databases":[
      {
         "primary":true,
         "name":"azure-primary",
         "connection-string":"",
         "provider":"azure",
         "mode":"passive"
      }
   ],
   "caches":[
      {
         "primary":true,
         "connection-string":"",
         "name":"redis-primary",
         "provider":"azure"
      }
   ],
   "storages":[
      {
         "primary":true,
         "connection-string":"",
         "master-container":"paasaggregatordocs",
         "name":"azure-storage",
         "provider":"azure-blob"
      }
   ]
}
```


Figure 3.3: Configuration JSON schema.

3.2.2 PaaS Aggregator Cache (PAC)

This is the cache accessor component of PaaS Aggregator. If a SaaS application is interested in utilizing cache services provided by PaaS providers, this particular module needs to be referenced. Our PoC PaaS Aggregator supports services offered by *Redis cache* from Windows Azure and *DynamoDb* from AWS. Both the cache providers are key-value pair NoSQL storages where the cached object will be stored in JSON format with a unique key to refer to it.

3.2.2.1 Configuring Cache Provider

From the supported cache providers in PaaS Aggregator, what provider(s) to use in the application is configured in *config.json* file. There is a dedicated section in config.json to configure cache accessor details as shown in Figure 3.4. Under each cache provider configuration, following four mandatory fields should be filled:

- *name* – This is the name given to a particular cache configuration. This name must be unique among the cache providers configured.
- *provider* – This denotes the underlying PaaS provider name and the service flavor that it provides. The two cache providers available in our PoC PaaS Aggregator are *azure-redis* and *aws-dynamo*. This is used by the *CacheAccessorFactory* to instantiate the appropriate cache accessor to serve the requests.
- *primary* – This denotes whether the cache provider is the preferred accessor that should be used every time when it is accessible. If this is set to false, it will not be utilized when the primary cache provider is accessible and capable of serving the requests. config.json can contain only one primary cache provider.
- *connection-string* – This is where the cache accessor platform-specific security information should be kept. For Azure Redis provider, primary connection string copied from Azure management portal should be placed here. For AWS DynamoDb provider, there are three types of information (*awsaccesskey*, *awssecretaccesskey*, and *region*) that should be pasted here separated by using a pipe symbol (|). Additional cache providers introduced later may adopt a different convention when configuring the platform-specific connection information.

3.2.2.2 Cache Accessor Façades

Cache Accessor Façades are located at PAC. There are two types of facades that can be found in PAC related to this.

- *ICacheAccessor* – This abstracts all the necessary function definitions, which in turn standardize the cache accessors. *CacheProviderFactory* provides the application with an instance, which implements *ICacheAccessor* depending on the configurations passed into it and the availability of the underlying resources.
- *CacheAccessorModel* – Every object that will be stored in the cache should be inherited from *CacheAccessorModel*. This is an abstract class, which holds one virtual property called *Key*. Key primarily represents the cached object reference.

Following section provides a detail description of the cache accessor API and how it can be used to manipulate data in cache.

```
"caches":[
   {
      "primary":false,
      "connection-string":"<azure-redis-connection-string>",
      "name":"redis-primary",
      "provider":"azure-redis"
   },
   {
      "primary":true,
      "connection-string":"<awsaccesskey>|<awssecretaccesskey>|<region>",
      "name":"dynamo-secondary",
      "provider":"aws-dynamo"
   }
]
```

Figure 3.4: Cache accessor configuration in config.json.

3.2.2.3 ICacheAccessor API

As shown in Figure 3.5, ICacheAccessor is a template interface that exposes four main functionalities. For each function, it expects the implementations to expose their *async* version as well. This is particularly important as the public API guideline from Microsoft [31] recommend that for the sake of performance and scalability, we should expose *async* versions of our API methods.

*CacheAccessorRedis (*in *PaaSAggregator.Cache.Azure.Redis)* and *CacheAccessorDyn*

*amoDb (*in *PaaSAggregator.Cache.Aws.DynamoDb)* are the two implementations of those facades. API methods Store and BatchStore will write cache information into the underlying provider while *RemoveItem* will delete the item from cache, if exists. By design, objects stored in the Redis cache will have a Time-To-Live (TTL) value of 2-hours, i.e., if an object is not removed within 2-hours, it will automatically removed from the store. There is no such restriction in AWS DynamoDb. Items will reside in the underlying storage until they are removed by the consumer.

```
1  public interface ICacheAccessor<T> where T: CacheAccessorModel {
2
3      void Store(T item);
4      Task StoreAsync(T item);
5
6      void BatchStore(IEnumerable<T> items);
7      Task BatchStoreAsync(IEnumerable<T> items);
8
9      void RemoveItem(T item);
10     Task RemoveItemAsync(T item);
11
12     T GetItem(string key,
13               Func<string, T> cacheMissAlternateGet = null,
14               bool insertInCacheOnAlternateFound = true);
15     Task<T> GetItemAsync(string key,
16                          Func<string, Task<T>> cacheMissAlternateGet = null,
17                          bool insertInCacheOnAlternateFound = true);
18
19 }
```

Figure 3.5: ICacheAccessor Façade.

One important aspect in CacheAccessor compared to other two services in PaaS Aggregator is that, it does not support a logging mechanism. This is because of the way how the cache is supposed to behave. Applications use a cache to optimize repeated access to data held in a data store. However, it is usually impractical to expect that cached data will always be completely consistent with the data in the data store. Applications should implement a strategy that helps to ensure that the data in the cache is up to date as far as possible, but can also detect and handle situations that arise when the data in the cache has become stale. In other words, as shown in Figure 3.5, PaaS Aggregator implements the cache aside pattern; thus, by providing the flexibility to the data layer of the application to provide an alternate method to get executed when there is a cache miss occurred. Algorithm 3.1 shows how the Cache Aside Pattern is implemented in PaaS Aggregator.

Architecture of the Cache Accessor in PaaS Aggregator is shown in Figure 3.7. As mentioned earlier, when it comes to write operations, PaaSAggregator will simply write to the first available cache provider configured, and it follows cache aside pattern to act

upon when there is a cache miss. Even though this behavior is optional in the function API, it is highly recommended to follow the pattern for better data consistency and availability.

Algorithm 3.1. Cache Aside Pattern

```
1: procedure CacheGetAsync<T>(key,fnCacheMissAlternateGet)
2:     value ← Cache.TryGetValue(key)
3:     if value == NULL then                          ▷ Cache Miss
4:         value ← fnCacheMissAlternateGet(key)
5:         Cache.AddItem(key, value)
6:     end if
7:     Return value
8: end procedure
```

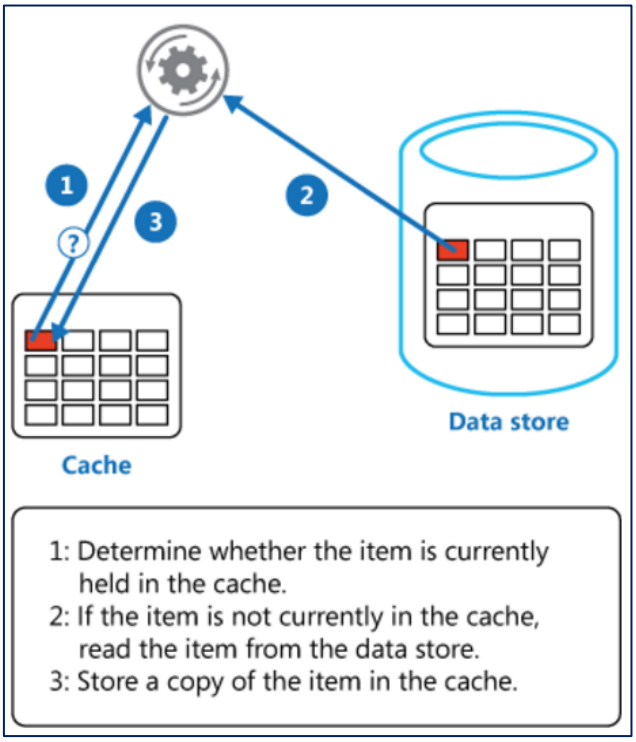

Figure 3.6: Cache aside pattern [32].

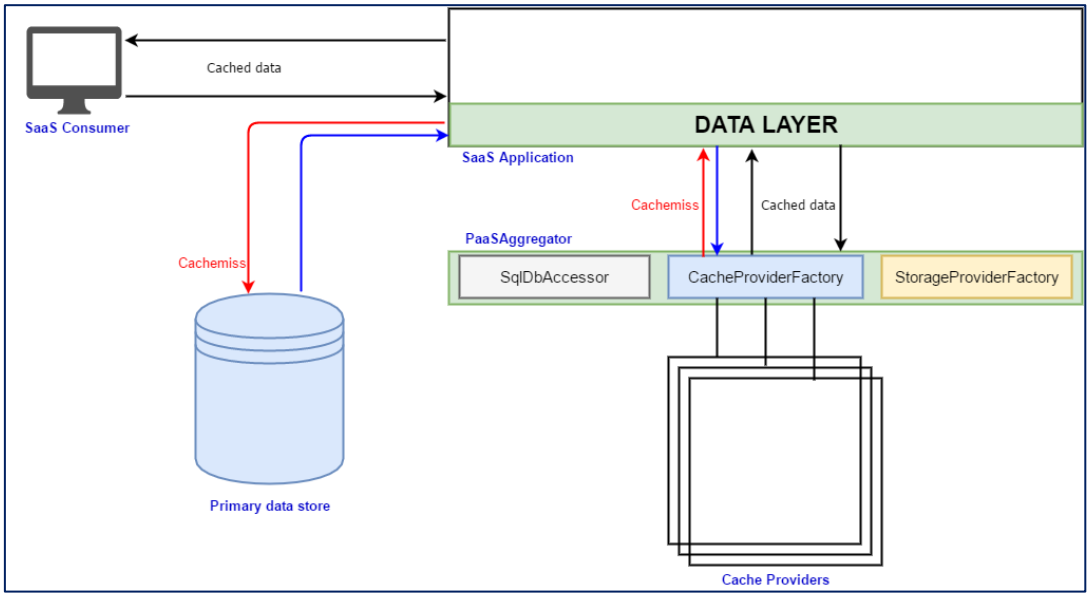

Figure 3.7: CacheAccessor Architecture - PaaS Aggregator.

3.2.2.4 PaaS Aggregator Cache Accessor – Azure Redis

PaaS Aggregator provides an implementation of *ICacheAccessor* for the SaaS applications developed on Windows Azure. There are several storage providers in Azure that can be used as a Cache (Redis, Table storage, DocumentDb, and Blob). Among those services, Azure Redis Cache is based on the popular open source Redis cache. It gives access to a secure, dedicated Redis cache, managed by Microsoft and accessible from any application within Azure. To use this a user has to create a node in Azure Redis under his/her resource group in the subscription. When the required resources are available, it is just a matter of extracting the access key in the portal and use it in one of the cache configurations (under the connection-string) in config.json. By default, TTL value of each item stored in Azure Redis Cache is set to 2-hours. PaaS Aggregator will convert the objects passed into the API to a JSON before storing them.

3.2.2.5 PaaS Aggregator Cache Accessor – Amazon DynamoDb

From the cache providers available on Amazon (Redis, DynamoDb, S3, etc.), PaaS Aggregator provides an implementation to use Amazon DynamoDb to store objects as key-value pairs. When configuring the application to use DynamoDb, PaaS Aggregator expects following three fields to be included in the connection-string:

- AWS Access Key
- AWS Secret Key
- Region name (e.g., ap-southeast-1)

As Amazon has different region names based on the service, it is important to extract the correct region name and paste it. These three fields should be joined by a pipe (|) and put the string in connection-string field.

3.2.2.6 Introducing New Cache Accessors

When it comes to extending the *ICacheAccesor* to provide additional cache providers, there is more to it than just implementing the exposed methods. We need to update the *ICacheAccessor* dependency resolver logic to instantiate an object of the new type based on the name configured in config.json. If there is custom platform specific, access information that are required, they should be extracted from the connection-string property described above.

3.2.3 PaaS Aggregator Database (PAD)

This is the component where the *dbAccessor* resides in PaaS Aggregator. If a SaaS application is interested in utilizing database services provided by PaaS providers, this particular module needs to be referenced. Our PoC PaaS Aggregator supports services offered only by Microsoft SQL Server. As long as there is a version of SQL server in any underlying PaaS provider, users can use the PaaS Aggregator to access it in a consistent manner, providing both high availability and enough logging mechanisms to make database layer consistent throughout, regardless of the database that is currently connected.

3.2.3.1 Configuring Database Provider

As PaaS Aggregator only supports SQL Server, all the configurations will look identical. As shown in Figure 3.8 config.json has a section to configure database accessor information. Under each database provider configuration, there are three mandatory fields and one optional field that should be filled:

- *name* – This is the name given to a particular db configuration. This name must be unique among the database providers configured.
- *provider* [optional] – This denotes the underlying PaaS provider name. Our PoC PaaS Aggregator is shipped with the support for *azure*, *aws*, *appharbour*, and *local*. At this juncture, this provider is not really in use as we only support for one flavor of the database engines.

- *primary* – Denotes whether the db provider is the preferred accessor that should be used every time when it is accessible. If this is set to false, it will not be utilized until the primary database provider is accessible and capable of serving the requests. There can be only one primary db provider in config.json.
- *connection-string* – This is the connection string information to the SQL Server database instance. The connection string typically contains the server information such as database name and required credentials to connect to the server.

3.2.3.2 Database Accessor Façades

Two types of Database Accessor Façades are located at PAC, namely:

- *IDbAccessor* – This abstracts all the necessary function definitions, which in turn standardize the dbaccessor. *SqlDbAccessor* in *PaaSAggregator.Database* namespace provides the application with an instance that implements *IDbAccessor* depending on the configurations passed into it and the availability of the underlying resources.
- *IDbLogAccessor* – This abstracts all the methods required by the PaaS Aggregator to log all the write operations. Every write operation will be persisted in any of the implementation of *IDbLogAccessor* injected into DbAccessor API. PaaS Aggregator is equipped with an implementation of *IDbLogAccessor* in case there is no any custom implementation provided. This out of the box implementation will utilize the server file storage to persist logs.

Next, we provide a detail description of the dbaccessor API and how the data will be stored.

3.2.3.3 IDbAccessor API

As shown in Figure 3.9, IDbAccessor is an interface that exposes two main functionalities. For each functionality, it expects the implementations to expose their *async* version as well. All the operations result in an update on the backend database is supposed to be directed to *Execute* API method as it is important to log these update operations to maintain consistency among different database instances in case of a failure in the primary database. There is no logging involved during the read operations. The *Read* operation will return a *SqlDataReader*, which can be used to extract data out.

The *Write* operation will return the number of rows affected as a result of the command executed in the backend. Our PoC PaaS Aggregator is only capable of handling SQL Server based data stores. That is the reason why there is a dependency on SQL-based data types in *IDbAccessor*. However, this abstraction can be updated in future so that it can be used to support any backend data store.

```
"databases":[
   {
      "primary":true,
      "name":"azure-primary",
      "connection-string":"<database-connection-string>",
      "provider":"azure",
      "mode":"passive"
   },
   {
      "primary":false,
      "name":"azure-secondary-1",
      "connection-string":"<database-connection-string>",
      "provider":"azure",
      "mode":"passive"
   },
   {
      "primary":false,
      "name":"aws-secondary-2",
      "connection-string":"<database-connection-string>",
      "provider":"aws",
      "mode":"passive"
   }
]
```

Figure 3.8: Database accessor configuration in config.json.

3.2.3.4 DbLogAccessor API

This is the interface that is utilized by the DbAccessor in PaaS Aggregator to log all the write operations sent to it. As shown in Figure 3.10, it exposes three main functionalities to read, write, and list the log files.

```
1  public interface IDbAccessor
2  {
3      Task<SqlDataReader> ReadAsync(string commandText, params SqlParameter[] parameters);
4
5      SqlDataReader Read(string commandText, params SqlParameter[] parameters);
6
7      Task<int> ExecuteAsync(string commandText, params SqlParameter[] parameters);
8
9      int Execute(string commandText, params SqlParameter[] parameters);
10 }
```

Figure 3.9: IDbAccessor Façade.

It is very important from the PaaS Aggregator point of view that the underlying provider for *IDbLogAccessor* is very robust and durable. That is the main reason why this abstraction is exposed outside for the SaaS application designers to decide on such a provider. Our PoC PaaS Aggregator supports one out of the box implementation, in which it utilizes the server file storage to persist the logs (*StorageLogAccessorLocal*). In case of no implementation is provided into the DbAccessor, PaaS Aggregator will use *StorageLogAccessorLocal* and the logs can be located at "\App_Data\Logs\Database\" or "\temp\PaaSAggregator\Logs\ Database\" depending on the application type (web or desktop based).

```
1  public interface IDbLogAccessor
2  {
3      string Read(string fileName);
4
5      Task<string> ReadAsync(string fileName);
6
7      bool Write(string fileName, string content);
8
9      Task<bool> WriteAsync(string fileName, string content);
10
11     string[] ListFiles();
12
13     Task<string[]> ListFilesAsync();
14 }
```

Figure 3.10: IDbLogAccessor Façade.

Additional implementations of *IDbLogAccessor* can be added to the PaaS Aggregator in future. Until then, they should be implemented by the PaaS Aggregator consumers be implementing the interface and injected into DbAccessor. As an important note to the *IDbLogAccessor,* the content passed into the *Write* method and the result expected from *Read* method are JSON serialized version of type *ExecuteSkeleton,* which is described below. *FileName* passed into IDbLogAccessor is based on the current server time's Unix timestamp.

3.2.3.5 DbAccessor Log Item Skeleton

Each database insert, update, and delete operation is logged in PaaS Aggregator. They will be persisted in the form of a JSON serialized string as shown in Figure 3.11. Both the command and the parameters will be available in each log item. Name, value and the SQL db type of the parameter are the three attributes persisted in each parameter.

When it is required to restore from these log items, they will be read and invoke the *IDbAccessor Write* method for each log item.

Figure 3.11: DbAccessor Log Item Skeleton.

3.2.3.6 IDbAccessor Architecture

As shown in the Figure 3.12, each database access operation, regardless of the operation type, has to first acquire a connection with the respective database specified in the config.json. Highest priority will be given to the primary database specified in the config.json.

To use the logging capability of DbAccessor in PaaS Aggregator, users need to introduce a table called ___*Checkpoints* which consists of a couple of columns to hold the checkpoint timestamps. Each write operation will be logged in the injected log storage provider and at the same time, an entry will be inserted into the ___*Checkpoints* table. Timestamp used in the corresponding checkpoint will be converted to an integer value and use that as the name of the log.

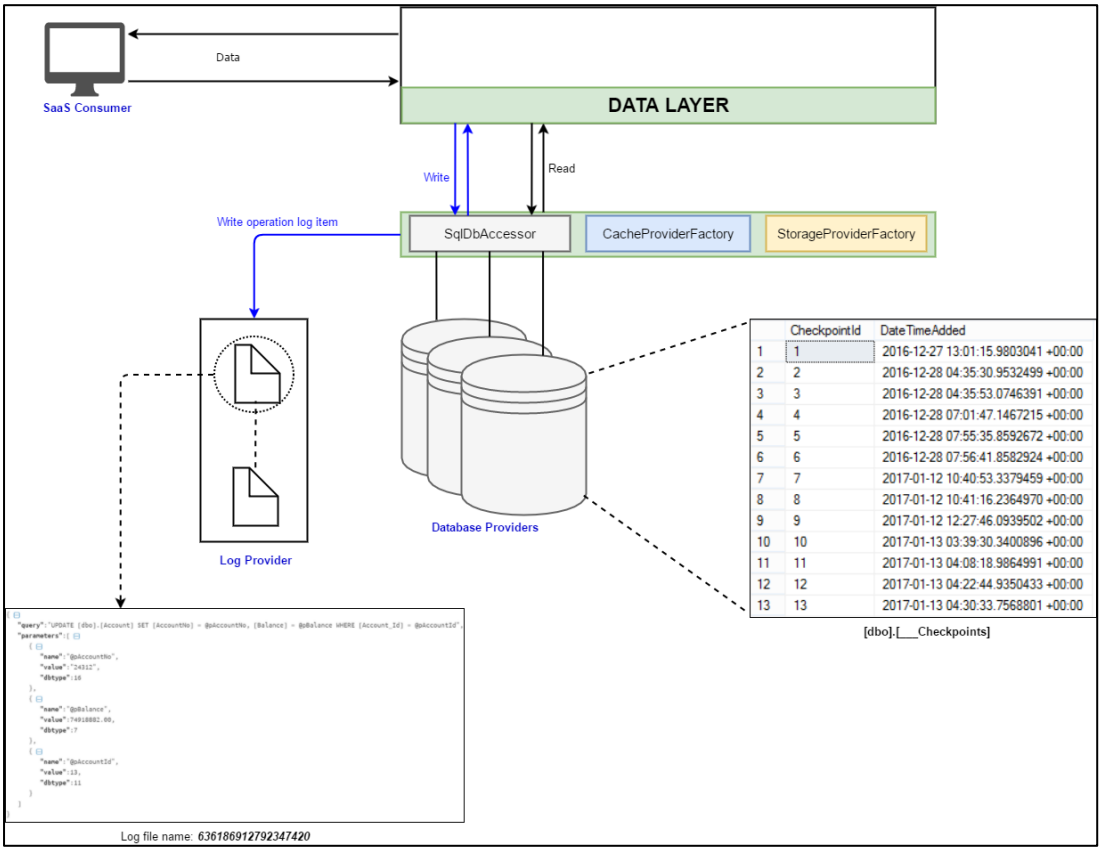

Figure 3.12: PaaS Aggregator DbAccessor Architecture.

If the connection cannot be established during an operation, PaaS Aggregator will automatically try to switch to one of the secondary databases listed in the config.json. Before executing the command in the secondary database, the system will check whether the database is in sync status.

This is done by comparing the latest checkpoint entry in the database with the latest log file name persisted. If they mismatch, the system will start finding out the matching log file for the latest checkpoint entry found in the ___*Checkpoints* table. Then it will start replaying the log items up until the latest log is located. Algorithm 3.2 shows how the system initializes an active connection to one of the databases specified in config.json. According to the Algorithm, it gives the highest priority to primary database provider. If the primary is not accessible, it goes through the other configured providers to find an active connection. Upon finding a successful connection, it checks whether that database is in sync by comparing the latest log item stored in the log storage and the latest checkpoint. If the database is not in sync, the log items will be executed before releasing the database connection to the consumer.

Algorithm 3.3 shows the synchronization process by replaying logs on new connection established. Taking the latest checkpoint into consideration, it collects all the log items to be executed by comparing the log item name to the checkpoint name. Once the log items are collected, they are executed in the order they were logged to maintain consistency.

Algorithm 3.4 shows the implementation of Write operation. First, the write operation will be executed on the database. Then a checkpoint will be added into the ___*Checkpoints* table. Then a log item will be pushed to the injected log accessor. These three operations are executed in one transaction to maintain atomicity. This will enable the system to keep all the databases in sync before they are ready to be used.

Because the consistency is very important in RDBMSs, the approach taken in the PaaS Aggregator to synchronize the data stores is aggressive. Therefore, it is recommended to keep a background service to take care of the synchronizing among the configured databases periodically. This will result in minimizing waiting time when switching among databases. At the same time, it is recommended to come up with a strategy to remove outdated logs if they are not required. For example, as the databases are being backed up periodically, it is not important to store outdated logs. Outdated unnecessary logs will consume a lot of space both in the log storage as well as in the database storage.

3.2.4 PaaS Aggregator Storage (PAS)

This is the component where the storage accessor resides in PaaS Aggregator. If a SaaS application is interested in utilizing storage services provided by PaaS providers, this particular module needs to be referenced. Our PoC PaaS Aggregator supports services offered by Windows Azure Blobs and Amazon S3.

3.2.4.1 Configuring Storage Provider

There is a section in config.json to configure storage accessor information as shown in Figure 3.13. Under each storage provider configuration, following three mandatory fields and two optional field should be filled:

- *name* – This is the name given to a particular storage configuration. This name must be unique among the storage providers configured.

Algorithm 3.2. Get An Active Connection From Configured Providers

```
1: procedure GetActiveConnectionAsync
2:     configs ← GetConfigurations()
3:     primarydbconfig ← configs.databases.primary
4:     conn ← GetConnection(primarydbconfig)
5:     if IsAlive(conn) then
6:         if CurrentActiveConnection ≠ conn then
7:             ReplayLogs(conn)
8:         end if
9:         Returns conn
10:    else
11:        alternatedbconfigs ← configs.databases.primary ≠ TRUE
12:        while alternatedbconfigs.Any() do
13:            conn ← GetConnection(alternatedbconfigs.next)
14:            if IsAlive(conn) then
15:                if CurrentActiveConnection ≠ conn then
16:                    ReplayLogs(conn)
17:                end if
18:                Returns conn
19:            end if
20:        end while
21:    end if
22: end procedure
```

Algorithm 3.3. Sync Database Using Logs

```
1: procedure ReplayLogs(connection,latestcheckpoint)
2:     lognames = GetLogNames() ORDER BY name DESC
3:     while dofilename > latestcheckpoint
4:         logitem ← deserialize(ReadLog(filename))
5:         logstack.push(logitem)
6:     end while
7:     while logstack.Any() do
8:         logtobereplayed ← logstack.pop()
9:         ExecuteAsync(logtobereplayed)
10:    end while
11: end procedure
```

Algorithm 3.4. Execute Commands With Logging Enabled

```
1: procedure ExecuteAsync(command,parameters[])
2:     conn ← GetActiveConnection()
3:     AWAIT ExecuteCommand(command, parameters)
4:     AWAIT AddCheckpoint(conn)
5:     AWAIT AddLogItem(command, parameters)
6: end procedure
```

- *provider* [optional] – This denotes the underlying PaaS provider name. Our PoC PaaS Aggregator is shipped with the support for *azure-blob* and *aws-s3*.
- *primary* – This denotes whether the storage provider is the preferred accessor that should be used every time when it is accessible. If this is set to false, it will not be utilized until the primary storage provider is accessible and capable of serving the requests. There can be only one primary storage provider configured in config.json.
- *connection-string* – This is where the connection string information to the storage provider is kept. The connection string typically contains access keys as well as other environment specific access details.
- *master-container* – This is the root container where the documents will be uploaded to. This value can be realized as the base container in Windows Azure Blob storage account or the root bucket in the AWS S3 account.

3.2.4.2 Storage Accessor Façades

Following two types of Storage Accessor Façades are located in PAS:

- *IStorageAccessor* – This abstracts all the necessary function definitions, which in turn standardize the storage accessors. *StorageProviderFactory* will basically provide the application with an instance which implements *IStorageAccessor* depending on the configurations passed into it and the availability of the underlying resources.
- *IStorageLogAccessor* – This abstracts all the methods required by the PaaS Aggregator to log all the storage write operations. Every write operation will be persisted in any of the implementation of *IStorageLogAccessor* injected into StorageAccessor API. PaaS Aggregator is equipped with an implementation of *IStorageLogAccessor* in case there is no custom implementation provided. This out of the box implementation will utilize the server file storage to persist logs.

3.2.4.3 IStorageAccessor API

Figure 3.14 shows the storage API façade, which exposes functionalities to work with storage items in cloud. The current version of the storage API is mainly targeted towards SaaS applications, which are having regular uploads and downloads of static binary content. The API does not yet expose any functionality to carry out real-time updates to

storage items. Shown below is a detail description of API methods provided in *IStorageAccessor*. Each functionality exposed under *IStorageAccessor* contains both sync and async version for better performance and scalability. Methods UploadStream, DownloadStream, DeleteItem, and ListItems are serving the purpose of handling storage items IsInSync flag denotes the current status of the storage provider.

Figure 3.13: Database accessor configuration in config.json.

```
1  public interface IStorageAccessor
2  {
3      bool UploadStream(StorageAccessorModel info);
4      Task<bool> UploadStreamAsync(StorageAccessorModel info);
5
6      StorageAccessorModel DownloadAsStream(StorageAccessorItemInfo info);
7      Task<StorageAccessorModel> DownloadAsStreamAsync(StorageAccessorItemInfo info);
8
9      bool DeleteItem(StorageAccessorItemInfo info, long? dateTimeAddedTicks, bool avoidCheckpointing = false);
10     Task<bool> DeleteItemAsync(StorageAccessorItemInfo info, long? dateTimeAddedTicks, bool avoidCheckpointing = false);
11
12     bool Exists(StorageAccessorItemInfo info);
13     Task<bool> ExistsAsync(StorageAccessorItemInfo info);
14
15     List<StorageAccessorItemInfo> ListItems(StorageAccessorItemInfo info);
16     Task<List<StorageAccessorItemInfo>> ListItemsAsync(StorageAccessorItemInfo info);
17
18     bool IsInSync { get; }
19
20     Dictionary<string, StorageEventLogItem> GetLatestEventLogItem();
21     Task<Dictionary<string, StorageEventLogItem>> GetLatestEventLogItemAsync();
22 }
```

Figure 3.14: IStorageAccessor API.

Unlike PAD, *IStorageAccessor* will not perform on the fly synchronizing when switching across different storage providers that are configured in config.json. The reason behind that is the cost of operation to restore object streams. Instead, *IStorageAccessor* exposes a flag called *IsInSync*, which is set when the storage is accessed. Algorithm 3.5 shows the approach to decide the sync status of the current storage provider. This algorithm will query for the latest log item stored both in master log provider as well as storage log provider. If the latest log item names match, it is an indication that the selected storage provider is in sync.

As seen in Figure 3.15, items accessed through IStorageAccessor are wrapped in either of the following models:

- *StorageAccessorItemInfo* – Holds item's basic information. This abstraction is added to the PAS to only retrieve items' metadata.
    - *Name* – Display name of the item.
    - *FolderHierarchy* – Path to locate the storage item (virtual directory structure).
    - *Metadata* – Other important information about the storage item.
- *StorageAccessorModel* – If the requirement is to retrieve all the information regarding the items stored, users need to use this type as this include the stream as well.

Algorithm 3.5. Checking Whether The Storage Provider Is In Sync

```
    procedure CHECKSTORAGEISINSYNC()
2:      latestmasterlog ← logaccessor.getlatest()
        if latestmasterlog == NULL then
4:          Returns TRUE
        end if
6:      latestcheckpoint ← getcheckpoints() ORDER BY name
        if latestcheckpoint == NULL then
8:          Returns FALSE
        end if
10:     Returns lastestmasterlog.logidentifier == latestcheckpoint.name
    end procedure
```

3.2.4.4 IStorageLogAccessor API

This is the interface that is used by the Storage accessor in the PaaS Aggregator to log all the write operations sent to it. As seen in Figure 3.16, it exposes two main functionalities to retrieve, write the log files.

```
1  public class StorageAccessorItemInfo
2  {
3      public Dictionary<string, string> Metadata { get; set; }
4      public List<string> FolderHierarchy { get; set; }
5      public string Name { get; set; }
6  }
7
8  public class StorageAccessorModel
9  {
10     public StorageAccessorItemInfo ItemInfo { get; set; }
11     public Stream Stream { get; set; }
12     public long? DateTimeAddedTicks { get; set; }
13     public bool? AvoidCheckpointing { get; set; }
14 }
```

Figure 3.15: StorageAccessor models.

As in PAD, the log accessor in PAS also expects the log storage to be provided by the API consumer. If an external implementation is not provided for the storage accessor, it will use the out of the box implementation (*StorageLogAccessorLocal*) which utilizes the server's local file storage and will save the log items in either "\App_Data\Logs\Storage\" or "\temp \PaaSAggregator\Logs\Storage\" depending on the application type (web or desktop based).

Unlike PAD, *IStorageLogAccessor* in PAS only persists the metadata about a particular storage item and the operation performed. While the *IDbAccessor* uses a table in the database for checkpointing purposes, *IStorageAccessor* maintains a couple of folders (or containers) in the underlying storage provider to store log items, as well as the actual data stream.

```
1  public interface IStorageLogAccessor
2  {
3      void AddLogItem(StorageLogItem log);
4      Task AddLogItemAsync(StorageLogItem log);
5
6      StorageLogItem GetLatest();
7      Task<StorageLogItem> GetLatestAsync();
8
9      StorageLogItem Get(int skip);
10     Task<StorageLogItem> GetAsync(int skip);
11 }
```

Figure 3.16: IStorageLogAccessor API.

If the storage provider is Windows Azure, you will find a container called "*storage-logs*" while in AWS S3, you will find a folder called "*storage-logs*" where the checkpoints will be added. As shown in Figure 3.17, information persisted in a particular checkpoint will contain following attributes:

- *Key* – Display name of the storage item.
- *Operation* – Operation performed on the item (*Added*, *Modified* or *Deleted*).
- *Datetime* – Operation timestamp in UTC.


```
{
   "key":"4000 TC 4.xlsx",
   "operation":"added",
   "datetime":"2017-03-21T05:18:41.6495815+00:00"
}
```


Figure 3.17: A Storage Checkpoint (Storage log item).

These storage checkpoints are JSON serialized strings of type *PaaSAggregator.Core.ConfigModel.StorageEventLogItem*. As shown in Algorithm 3.5 these checkpoints will be checked against the master log items to decide whether the storage is in sync.

Apart from the checkpoint, PAS sends log items to *IStorageLogAccessor, which* should be persisted in a reliable manner for synchronizing purposes. These log items are called *master log items*. Figure 3.18 shows the information stored in a master log item. In addition to the information stored in a storage log item, following information are also included in a master log item:

- *LogId* – This is the converted integer value of the Datetime attribute, which is actually the name of the storage log item.

- *Provider* – This is the identification of the underlying storage provider on which the operation took place. As of now, this can contain either "*azure-blob*" or "*aws-s3*".

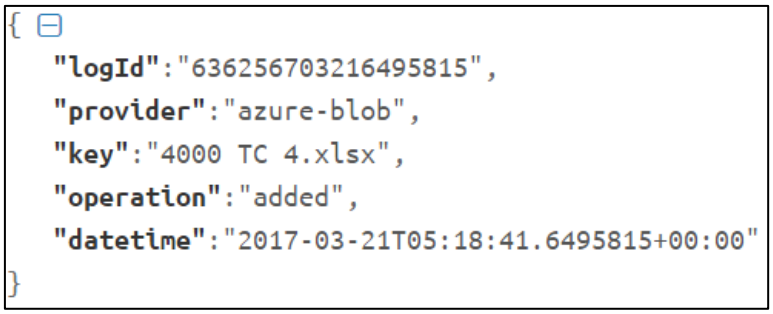

Figure 3.18: Master log item.

3.2.4.5 IStorageAccessor Architecture

As shown in Figure 3.19, each storage item access request requires an active connection to a configured storage provider in config.json. Highest priority is given to the *primary* storage provider. *StorageProviderFactory* will hold the responsibility of spawning the appropriate instance of the accessible storage provider.

As shown in the architecture diagram (Figure 3.19), each write operation will end up creating two log items, one in the storage itself and one log item in master log storage. Both the log item names should be the same and the name is the Unix timestamp of the server's current time. An important decision taken during the architecture of the PAS is that the existing implementation of *IStorageAccessor* do not impose a restriction on the operations performed, even if the storage is not in sync. As there is a flag to indicate the sync status, that decision is outsourced to the API consumer.

Hence, switching across different storage providers will not trigger a syncing process as the cost of that operation may be high. However, PaaS Aggregator provides a utility (*StorageSynchronizer*), which can be used in a background service to run in a periodic manner. This utility is capable of synchronizing the storage providers configured in config.json. The algorithm used in the utility is shown in Algorithm 3.6. This algorithm will scan through the storage configurations and compares each configured storage providers sync status with respect to the latest log item stored in master log. If the

considered storage provider is not in sync, then the log items that need to be executed are retrieved by calling Algorithm 3.7.

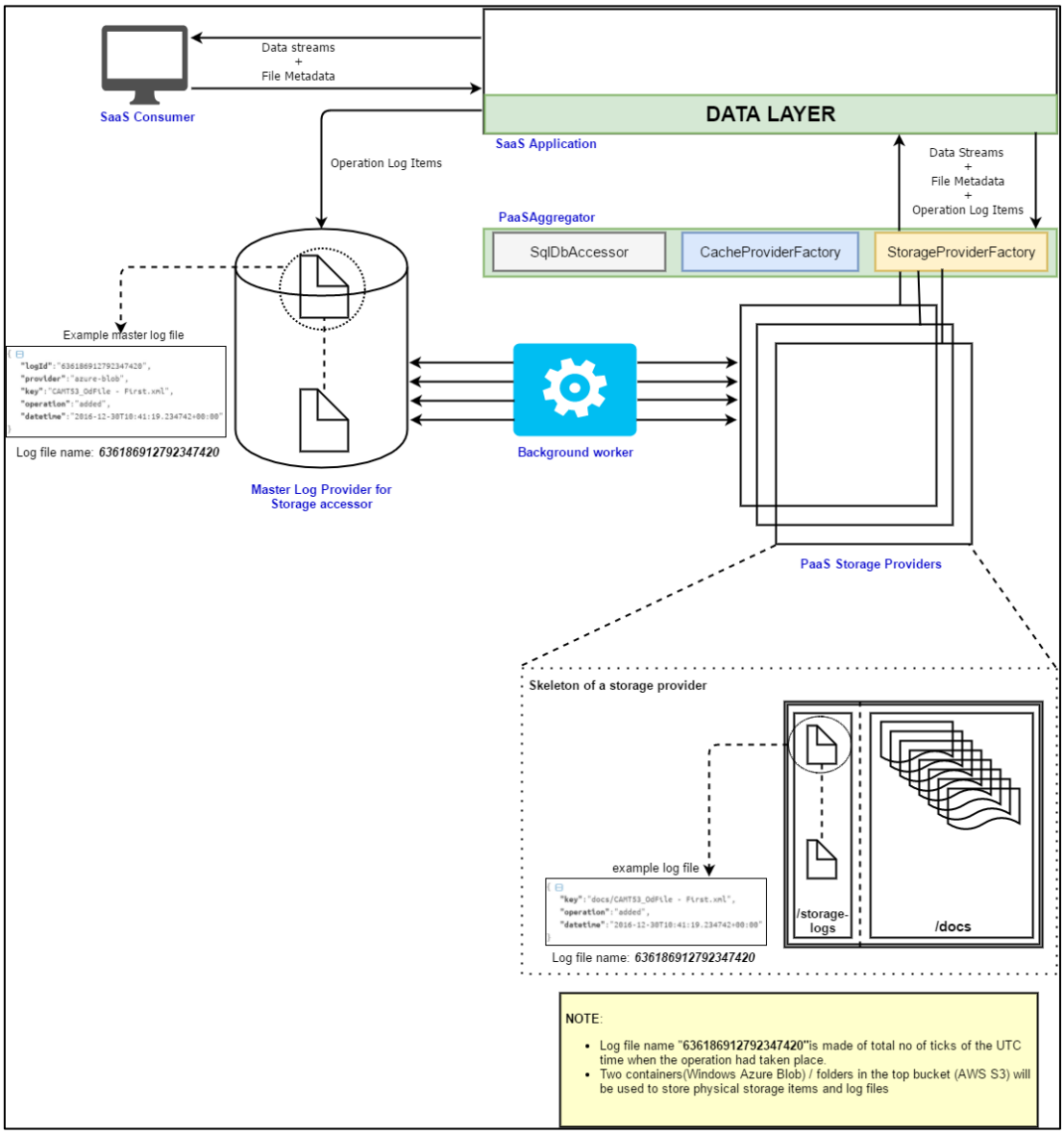

Figure 3.19: PaaS Aggregator Storage Accessor Architecture.

Once the log items are collected, *PerformOperation* will be called on each log item for the selected storage provider. The *PerformOperation* method call will invoke methods provided in the *IStorageAccessor* depending on the operation specified in the log item to manipulate the log items between the source and destination storage providers. The *Getmasterlogitemsuntil* shown in Algorithm 3.7 will query the *IStorageLogAccessor* and retrieve the items to be replayed in the destination storage accessor by taking the datetime stamps into consideration. It will go through log items stored in the master log accessor and collects the logs into a stack until it encounters a log item which has the

matching name as the one provided in. Upon satisfying this condition, it will return the collected log items to the *DoSync*.

3.3 Summary

PaaS Aggregator overall solution architecture is designed after carefully studying Cache, Database, and Storage API's provided by Windows Azure and AWS SDKs. PaaS Aggregator follows a component based architecture, where each service it provides, can be referenced without depending on all the library dependencies. PaaS Aggregator Core defines all the facades and models that will be common to all the components while PAC, PAD, and PAS implement functionalities abstracted in Core. PAD and PAS provide additional logging facades where the consumer should inject an implementation of it. PAC follows Cache Aside pattern where a cache miss will end up retrieving data all the way from the origin and save them in the cache.

Algorithm 3.6. Storage Item Synchronizer Background Service

```
1:  procedure DoSync(storagelogaccessor, storageconfigs)
2:      while storageconfigs.length > 0 do
3:          destinationstorageaccessor ← storageconfigs.next
4:          if destinationstorageaccessor.IsInSync == FALSE then
5:              lateststoragelogitem ← destinationstorageaccessor
6:                                          .getlatesteventlogitem()
7:              logitemstobereplayed ← Getmasterlogitemsuntil(
8:                                          lateststoragelogitem)
9:              while logitemstobereplayed.length > 0 do
10:                 masterlogitem ← logitemstobereplayed.pop()
11:                 sourcestorageaccessor ← Getstorageaccessor(
12:                                          masterlogitem,
13:                                          storageconfigs)
14:                 PerformOperation(sourcestorageaccessor,
15:                                  destinationstorageaccessor,
16:                                  masterlogitem)
17:             end while
18:         end if
19:     end while
20: end procedure
```

PaaS Aggregator is designed in such a way that, if the API consumer decides one day to get rid of PaaS Aggregator, it is not going to be a tedious effort to rewrite the data access layer. Whenever possible all the functionalities defined in PaaS Aggregator are provided with an async counterpart, which will not make it a performance bottleneck during I/O operations.

Algorithm 3.7. Retrieving Master Log Items To Be Replayed

```
1: procedure GETMASTERLOGITEMSUNTIL(LOGID, MASTERLOGACCESSOR)
2:     do
3:         nextitem ← masterlogaccessor.get(index)
4:         logstack.push(nextitem)
5:     while nextitem ≠ NULL && nextitem.logidentifier ≠ logid
6:     Returns logstack
7: end procedure
```

Whenever it is possible, enough endpoints are provided in the PaaS Aggregator to extend the functionality to get better throughput and reliability. At the same time, required known exception types are defined in each component to handle erroneous situations. For maintaining consistency across different service providers, required flags, and utilities are also provided together with the API. All the core components are implemented in self-containing components so that each service can be individually used without referencing to everything.

CHAPTER 4
PERFORMANCE EVALUATION

To demonstrate the effectiveness of the proposed multi-cloud library, we performed an extensive set of performance evaluations so that we can demonstrate that the PaaS Aggregator will not post any major overhead when it comes to performance. We developed a set of test scenarios that executed on top of a popular dataset [33] under a considerable workload.

Section 4.1 and 4.2 focus on test environments and setup, respectively. Sections 4.3, 4.5, and 4.6 present the test results for each service offered by PaaS Aggregator. Section 4.4 presents the test results collected when there is an infrastructure failure in database provider where it resulted in PaaS Aggregator switching to the secondary provider. Section 4.7 presents a summary of the overall performance.

4.1 Workload

As the initial problem definition that we are trying to solve came from a hosted web application due to a cache service outage back in 2014, we used the same kind of an environment for the performance evaluation. The web application was a single page application where the front end was mostly written in JavaScript. There were REST APIs exposed in ASP.Net Web API, which are responsible for handling simple transactions. Those transactions include database transactions as well as uploading/downloading of documents as binary content. Logged in users' session information were stored in Azure distributed cache for the performance. User base was from the financial domain and maximum users concurrently users was around 1,000. Average workload was around 750 to 800 concurrent users. Most users used the application during the end of a particular fiscal year. Consequently, the workload that we are trying to impose on the environment is a typical load/stress test on a web server. We used JMeter [35] to play the workload against the web application. First, we generated JMeter scripts to simulate a set of concurrent users who send requests to the web server depending on the type of test scenario. With respect to storage access, we assumed that the users will use the web server for static file accessing by uploading, downloading, and listing down the files.

To test the PAD, we used the popular AdventureWorks [33] dataset, which is heavily used in .Net-base applications. Adventure Works Cycles, the fictitious company on which the AdventureWorks sample databases are based is a large, multinational manufacturing company. The company manufactures and sells metal and composite bicycles to North American, European, and Asian commercial markets. While its base operation is located in Bothell, Washington with 290 employees, several regional sales teams are located throughout their market base. The dataset is mainly a transactional-based retail dataset.

- Test cases:
    - Requests to access ONLY the database (MSSQL Server)
    - Requests to access ONLY the cache provider (Amazon DynamoDB)
    - Requests to access ONLY the storage provider (Azure Blob)
    - Requests to access ALL the above mentioned services randomly

Each of the above mentioned tests were carried out against two different data layers integrated into the API server. One data layer used the PaaS Aggregator to access resources while the other used a vendor-specific SDK's to do so. To simulate the web application workload, we used JMeter to send requests from 1,000 concurrent threads (users) with a ramp up time of 10 seconds. These numbers were selected based on the statistics that we collected over the last three years. For each test case mentioned above, following statistics were collected from the two types of data layers:

- Transaction Throughput vs. Threads

  This shows total server's transaction throughput for active test threads. In other words, it shows the statistical maximum possible number of transactions based on number of users accessing the application. The formula for total server transaction throughput is as follows:

$$\frac{Active\ threads * 1\ second}{1\ thread\ response\ time}$$

Equation 4.1: Total Server Transaction Throughput

- Response Time vs. Time

  This shows the web server's response time to each request sent. Naturally, server takes longer to respond when a lot of users requests it simultaneously, as well as the type of service it has been requested for.

4.2 Experimental Setup

We used a 3-tier web application (see Figure 4.1) where the front end is a REST based API server which serves JSON objects to consumers. This API server was developed in ASP.Net Web API 2.0 and hosted in IIS 8.0. The web application was hosted as a web site in Windows Azure where the app service plan is set up to use the basic tier 2 (B1) pricing model. This app service plan is equipped with a single Virtual Core and 1.75 GB of memory. Auto scaling rules were set up to scale out the number of web application instances to two, if the current average CPU utilization of the app service plan exceeds 80% or the average memory utilization exceeds 85%. The reason for using these values is based on the statistics we collected over five years based on our workload on the web application where the incident took place in 2014. As shown in the figure, the API server is configured to use services from three service providers, namely Windows Azure, AWS, and AppHarbor.

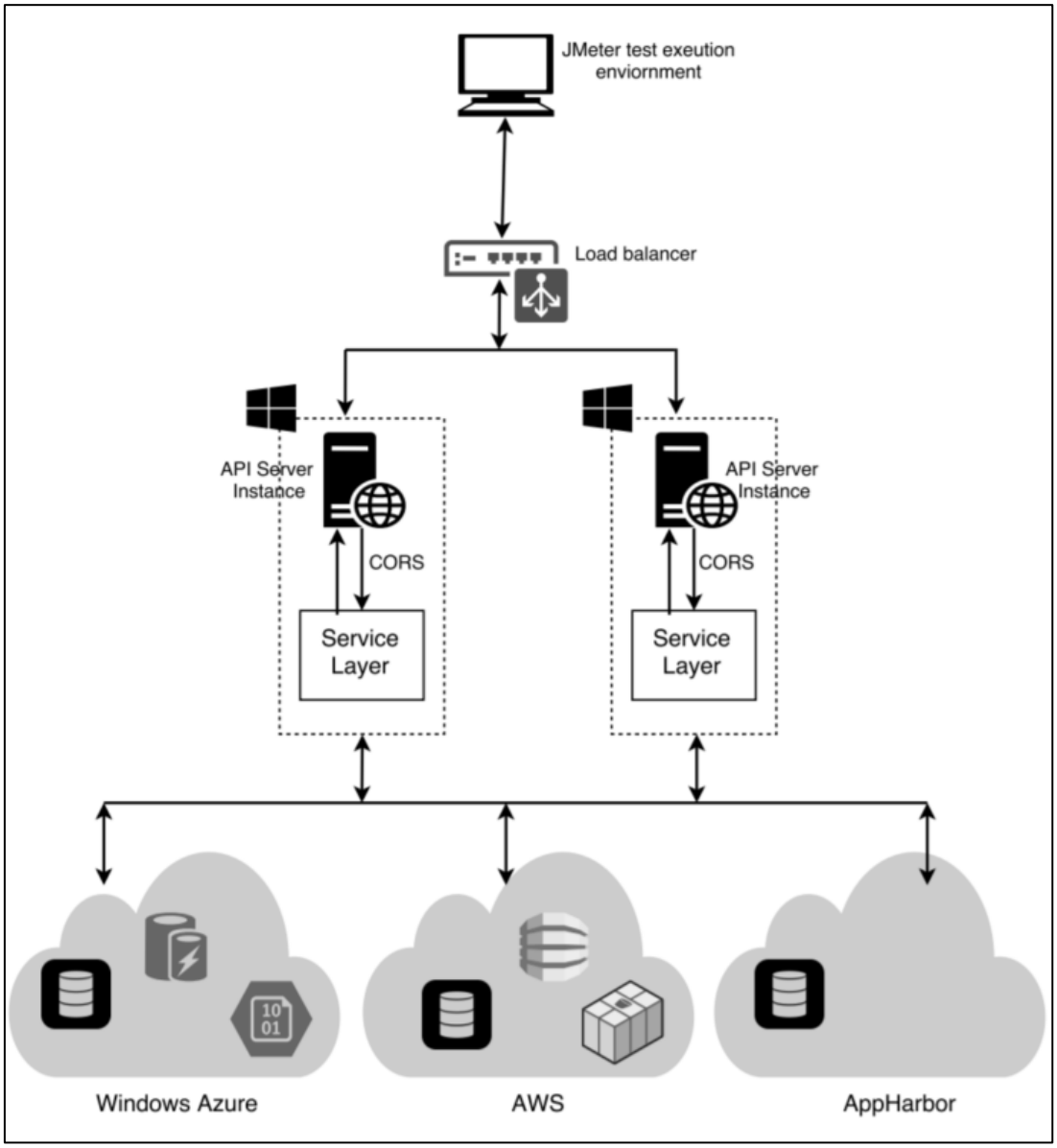

Figure 4.1: Experimental setup including the JMeter workload generator.

The service and data access layer resides in the same environment (as this is the same set up that we have in current live environment). Service layer handles the business logic while the data access layer handles database transactions, cache access, and storage access for simple file manipulations. The data access layer is developed in such a way that we can switch between an implementation which uses direct vendor-specific SDKs to access the services and another implementation which will utilize PaaS Aggregator to access the services. A configuration stored in the server decides which data access layer to be used and then injected the appropriate implementation into the service layer. Initially, the API server was deployed with the first version of data access layer and carried out the performance test scenarios and then change the configuration to utilize the vendor-specific APIs and collected the statistics.

Performance test case execution environment was Apache JMeter 3.1 installed on a machine with an Intel Core i7-5600 @ 2.60 GHz (4 CPUs), 16 GB RAM, Intel HD Graphics 5500, 1 TB of secondary storage. Average download and upload bandwidth available for the JMeter node during the test case execution were 14 Mbps and 3 Mbps respectively.

4.3 PAD Performance

We consider Throughput vs. Time and Response duration vs. Elapsed Time as performance matrices. One important note worth mentioning here is that PaaS Aggregator is not aiming at improving the performance of the data layer. The idea behind the performance evaluation on PaaS Aggregator is to check whether incorporation of into the data layer would introduce any additional overhead on overall application performance.

To carry out performance test scenarios on *dbaccessor*, we extracted various query executions and included in the service layer of the API server. These queries contain simple SELECT statements as well as complex queries, which involves joining of several number of tables to retrieve data. More than 45% of the queries got executed during the test were write operations as PaaS Aggregator might under-perform during these type of operations, as it needs to run back up operations for each write.

Same set of test scenarios were executed against the two aforementioned data access layers and statistics were collected. As an implementation of *IDbLogAccessor*, we incorporated Azure Blob as the primary log storage for the DbAccessor in API server. Table 4.1 shows the JMeter test thread group parameters and the other set up parameters that were used.

Table 4.1: DbAccessor test setup parameters.

Parameter	Value
Number of threads (Users)	1,000
Ramp-up period (s)	10
Loop count (Number of time each thread gets executed)	3
Number of read ops per thread	7
Number of write operation per thread	6

4.3.1 PAD - Throughput comparison

As a step during the JMeter test script compilation, we added a graph result listener and extracted the throughput graph in both occasions. Figure 4.2 shows the throughput graph when the data layer is configured to use the direct vendor-specific SDKs to access the database service, while Figure 4.3 shows the throughput of PaaS Aggregator enabled data access layer. Throughput, in this scenario is the ability of the API server to handle heavy load.

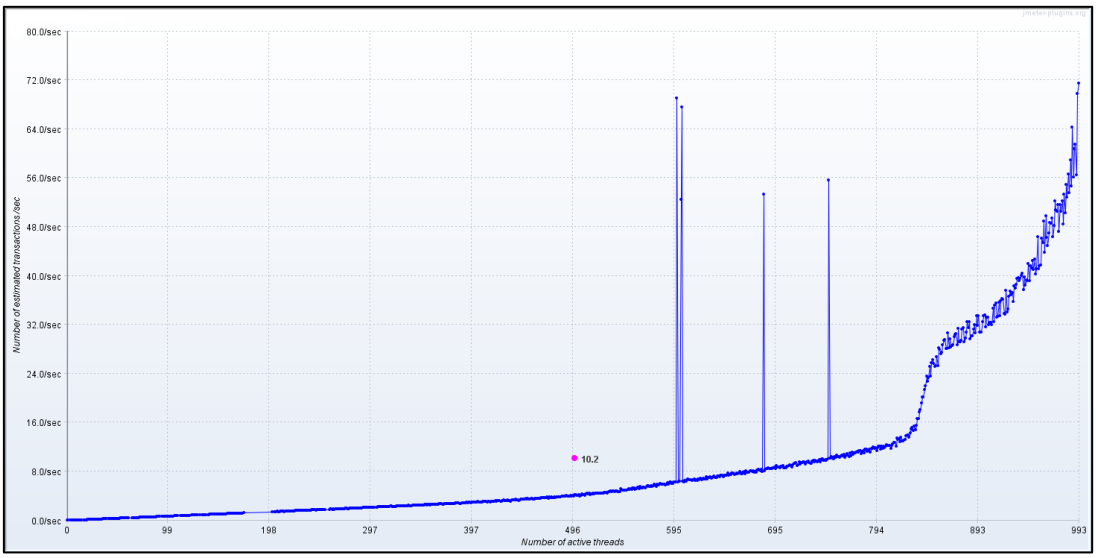

Figure 4.2: PAD Performance – Transaction Throughput vs. Time (accessing through vendor-specific API).

With the high number of active threads were created during the initial phase of the tests, both versions show a relatively high throughput (Transactions per second). However, as the number of active threads decreases, the throughput decreases. But according to Equation 4.1 (Section 4.2), ideally the transaction throughput should decrease in a linear manner. However, as we can see, this is not the case up until the number of active threads reach a value around 600. This is due to the App service configuration that we have in Azure. Although we set up a scale out rule to spawn out another instance when the resource utilization is high, the resources available for a given instance was not enough to handle the workload. This is understandable when we observed the measurements provided by Azure regarding the health of the App service (see Figure 4.6). We could observe during the initial phase that the CPU utilization of the instances was almost 100%, which resulted in queuing of requests during the time.

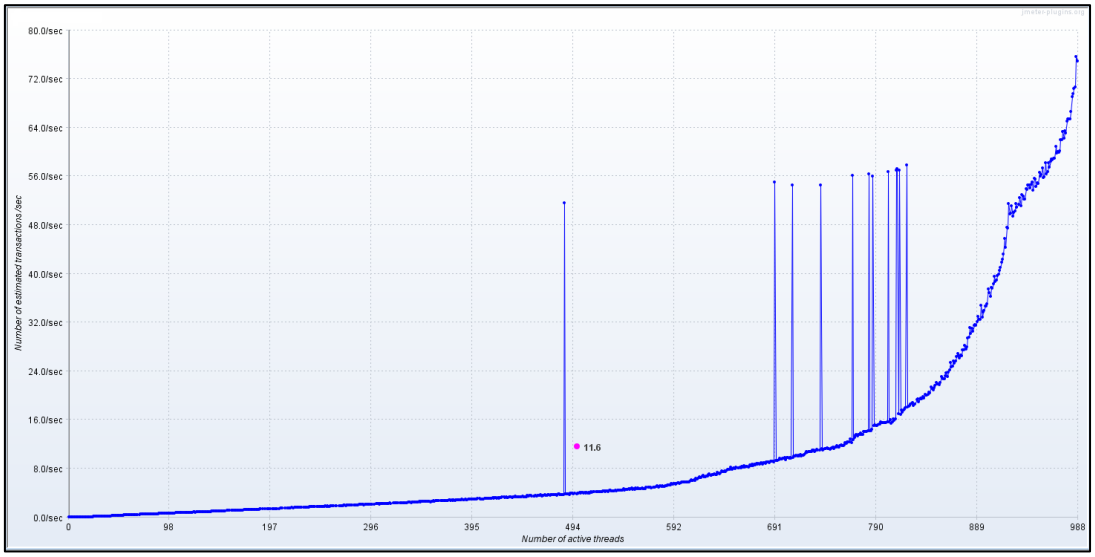

Figure 4.3: PAD Performance - Transaction Throughput vs. Time (accessing through PAD).

Anyhow, if we compare the two graphs, transaction throughput is not much different between the two implementations. Indeed, at the initial stages, the average throughput exceeded when the data layer was configured to use the PAD. Throughput values can be affected by the network bandwidth and delays, as well as JMeter internal processing delays. However, on average, both the implementation showed a near identical throughput distribution over the test period.

4.3.2 PAD – Response times comparison

The same characteristics are visible in the Response time graphs in Figure 4.4 and 4.5. Response times for the API calls displays similar characteristics which suggests that the introduction of PAD into the data layer does not add much of an overhead to the application performance. As can be seen in both figures, overall performance of the application is mainly affected by the few complex database update operations throughout the testing period. At the meantime, the overall response times are 1.56% higher for read operations and 8.90% higher for write operations in Figure 4.5 compared to Figure 4.6.

This is possible due to the extensive logging that is built into PaaS Aggregator. From one side, PaaS Aggregator needs to keep checkpoints for each write operation. This is done by inserting new checkpoint entries to ___*checkpoints* table. This is not necessarily a heavy operation. However, if there are unnecessary outdated checkpoints

available in the table that may affect the checkpoint read operations. Some of the measures that we can take to avoid this are to:

- Introducing a non-clustered table index.
- Backup the database regularly and keeping a background housekeeping service to delete checkpoints, which are older than a configured threshold.

Another improvement that we can propose as a future work is to abstract the check pointing service and inject a custom implementation, which might be a much faster version than using a SQL table.

Another contributing factor for the high response time, especially for write operations, is the performance in the implementation of *IDbLogAccessor*. For each write operation, PaaS Aggregator sends a JSON containing the command and parameters to the log accessor. It is the log accessor's responsibility to implement its functionalities as efficient as possible for the overall performance of the data layer. For example, if the service layer and data layer reside in South East Asia datacenter, and the log accessor storage is located in a data center in West Europe, it will definitely have an impact on the performance. Therefore, it is important to keep the database server, business layer, and the log accessor as close as possible for better performance. However, with the time response time improved, due to the number of threads the server needed to respond dropped and as a result, the two instances could handle the load much better.

Another important observation is that during the same period the CPU utilization reached almost 100% (see Figure 4.6), which must have had an effect on the increasing latency during the time. Even though there is a scaling-out rule specified for the app service, as the instance size is basic, it had reached those figures.

4.4 PAD Performance – Switching across different providers

As the *dbaccessor* supports on-the-fly synchronizing among the providers when switching across them, it is important to test how it performs under a relatively high load. To test this scenario, we created three SQL server databases in Windows Azure, Amazon RDS, and AppHarbor. Table 4.2 shows the JMeter thread group parameters as well as the other test set up parameters. The regions we chose for the three databases are East Asia, West Europe, and East US.

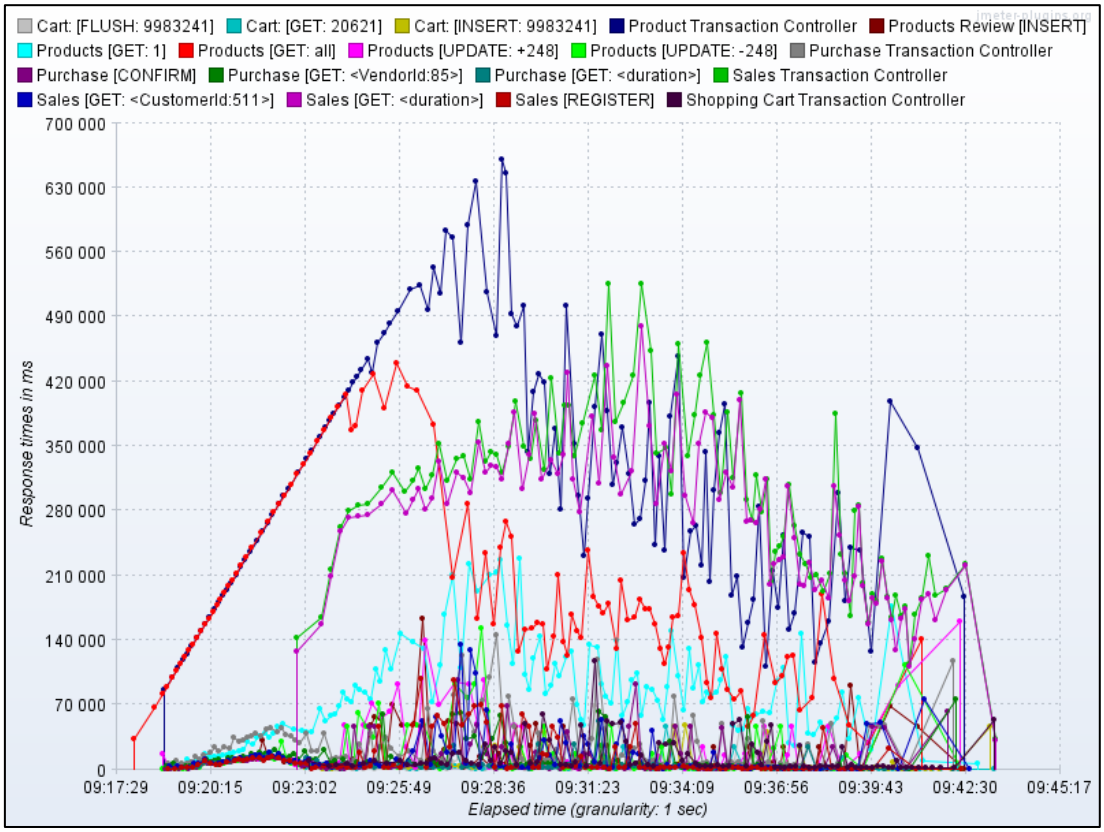

Figure 4.4: PAD Performance – Response time vs. Time (accessing through vendor-specific APIs).

SQL Server databases created in Azure and Amazon are more or less having the same performance levels. They are created in SQL Servers installed in Windows Server 2012 R2. Both the servers have near identical hardware. However, the AppHarbor database was created in their free tier. It does not have the same configuration level when it comes to hardware resources. Nevertheless, the idea was to start the JMeter test script execution from the SQL Azure database and change the config.json by logging into the Azure deployment environment to use the Amazon RDS database as the primary database in the middle to simulate a database-switching scenario. Likewise, the system will be switched to the database hosted in AppHarbor towards the end of the test execution. Therefore, as we can see the performance overhead during the switching between two service providers is mainly due to the amount of data that should be synchronized and the hardware resources available in the active provider.

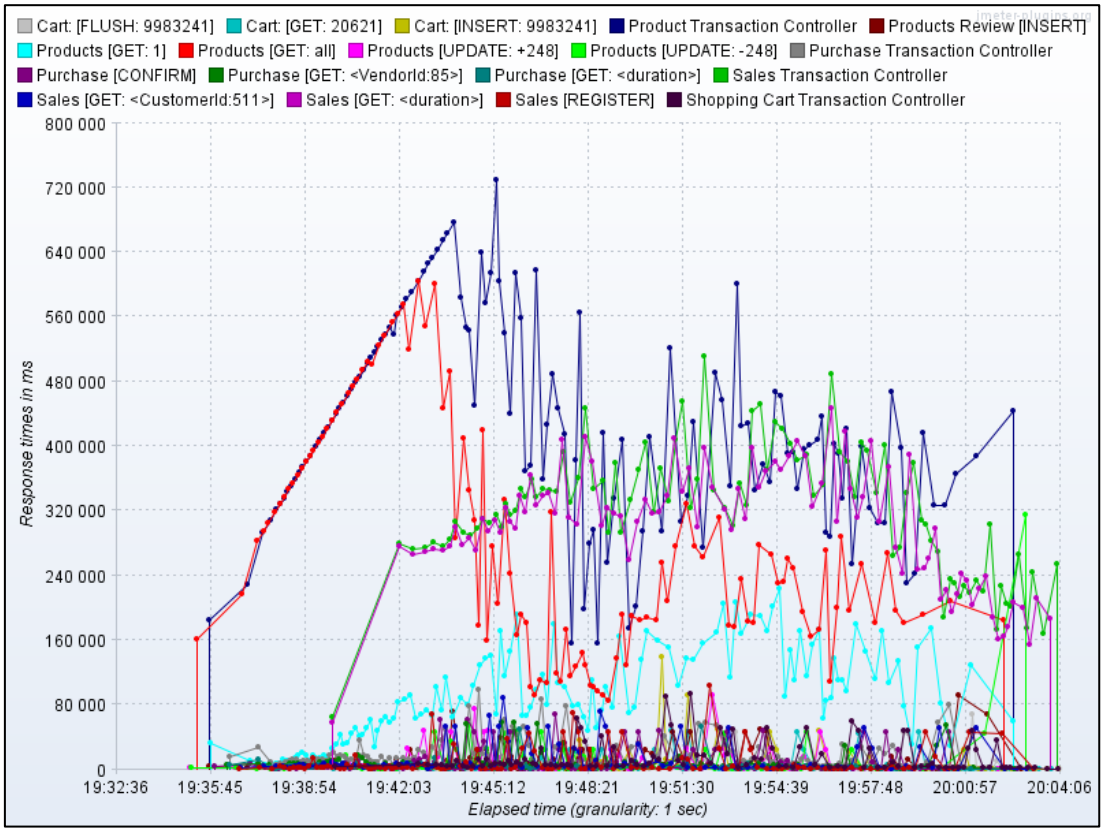

Figure 4.5: PAD Performance – Response time vs. Time (accessing through PAD).

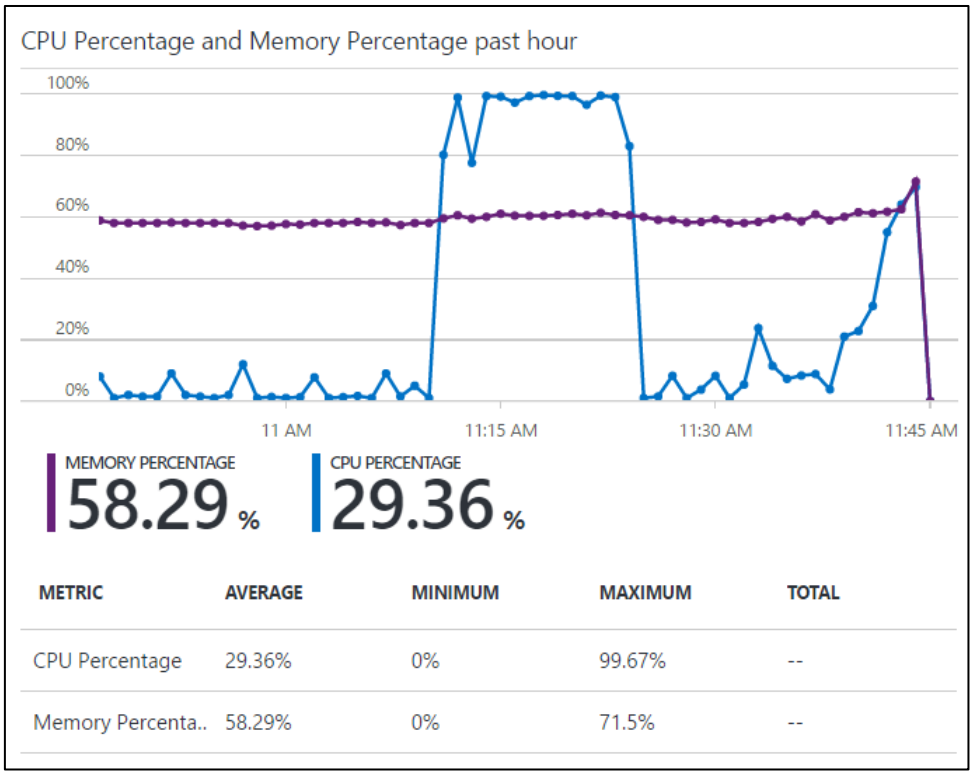

Figure 4.6: Resource utilization of the App service plan.

Table 4.2: DbAccessor test setup parameters – switching between databases.

Parameter	Value
Number of threads (Users)	1500
Ramp-up period (s)	10
Loop count (Number of time each thread gets executed)	3
Number of read/write requests per thread	14

As we can see in Figure 4.7, first switching from Windows Azure to Amazon happened around 20:36 where it took some time to replay the logs stored in Azure blob storage to sync the database hosted in the Amazon. Switching from Amazon to AppHarbor took place around 20:38. At that time, JMeter had already executed 41,358 samples out of 63,000 test samples. Out of 41,358 API calls, more than 18,600 (45%) calls trigger an update in the database, which causes the PaaS Aggregator to push a log item for each. That is the main reason why the response time increase sharply around that time. Replaying these logs in a relatively slow SQL server made the application wait until the syncing operation completed. Even after the database was fully synced and capable of handling the rest of the requests, response time does not reverted back to previous state because of the hardware configuration in SQL Server. Another observation from Figure 4.6 is the continuous increase of response time of the request *Products Review* from 20:38 for some time even after the synchronization was completed. If we consider this specific request, it is composed of number of complex queries which updates six tables in the database. This is visible in the graph as the response time for this specific operation is relatively high compared to other operations from the beginning. As the SQL server switched at 20:38, this operation has become even slower to execute in AppHarbor hosted database which is got basic hardware resources allocated to the SQL server it resides. That is the reason why it kept climbing even though the storage synchronization completed. As we discussed in the previous section, this is the reason why we recommend maintaining a background service, which does the synchronizing of databases asynchronously.

4.5 PAC Performance

For the performance testing we configures Caching in the hosted API server to store logged in user sessions. Upon each login request, an authorization object is created by retrieving information from the database. This is a complex object, which represents the current logged on user's privileges. A successful login will end up storing this type of

an object in JSON format. API server is configured to use a DynamoDb instance created in East-Asia region.

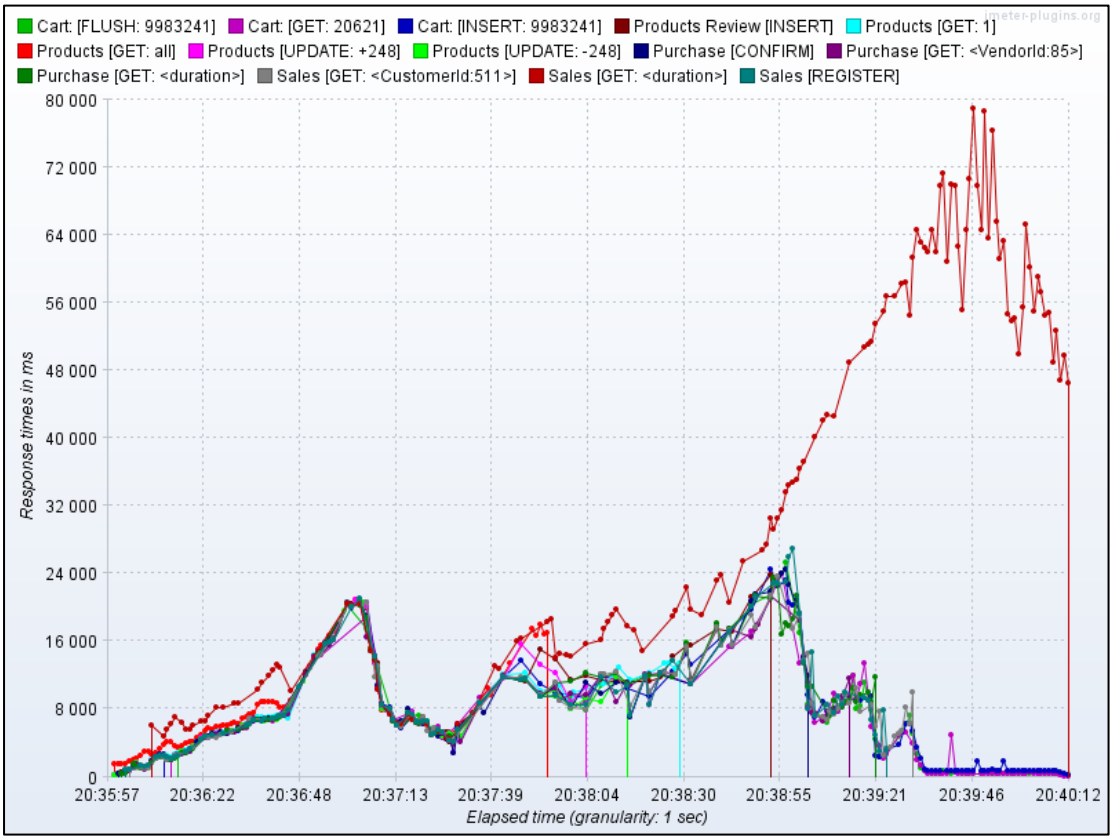

Figure 4.7: PAD – Switching between providers - Response time curve.

One version of the Cache accessing logic in the data access layer used the AWS SDK directly while the other version was configured to use the Dynamo Db as the cache provider. The latter version used the *IDbAccessor* to access the database and create the rights map. It also utilized the cache-aside pattern to recreate the cache objects, if there is a cache miss. JMeter script to test the PAC performance comprised with three requests per thread. These requests would:

- Log the current user in, create an entry in the cache for the current user.
- Accessing the current user's privileges already stored in the cache.
- Logged out the user, thus removing the cache entry.

These requests were scattered across different threads intentionally to simulate a cache miss scenario. Table 4.3 shows the JMeter thread group parameters as well as the other test set up parameters used in the analysis.

4.5.1 PAC – Throughput comparison

Figure 4.8 and 4.9 show the transaction throughput variation with the active threads. As the data layer which does not use PAC and Cache Aside pattern, it simply returned a NULL value

avoiding the application to fail. However, the version that uses the PAC (see Figure 4.9) and utilizes the *IDbAccessor* in PaaS Aggregator as well to retrieve the information from the backend database which get cached. Because of this, the transaction throughput curve does not look as smooth in the beginning as the other version.

Table 4.3: CacheAccessor test setup parameters.

Parameter	Value
Number of threads (Users)	1000
Ramp-up period (s)	10
Loop count (Number of time each thread gets executed)	3
Number of read requests per thread	3

We believed that most of the login requests must have happened towards the beginning, which causes the curve to fluctuate. Absolute throughput value fluctuated at times when the application tries to recover from cache misses by retrieving the data again and trying to recreate the cache and persist. However, the overall average transactions per second the system could handle remained at 101.7 per second, which is a good indication that PAC does not add any overhead on the cache accessing layer of the API server.

Although PAC-enabled version of the data layer has to do more work, it regained the throughput during the latter part of the test and increased the throughput considerably as a result of less number of cache misses. We are not doing any additional logging in PAC, any additional overhead should have been from PAD. As we realized in the previous section, there are ways to minimize that overhead and improve the overall throughput of the *CacheProvider*.

4.5.2 PAC – Response time comparison

Figure 4.10 and 4.11 show the response time when the cache is accessed via the PAC and directly through vendor-specific APIs. As we can see, response times fluctuated when the PaaS Aggregator is configured in the data access layer. The reason for this fluctuations, we believe, is the characteristics of the test scenario we selected. The test

scenario has a direct dependency on PAD when logging the current user in through the membership provider.

This additional dependency adds up the extra response time, which we observed in Section 4.3.2. This causes the overall response times on PAC to have a relatively higher value compared to its counterpart, which is shown in Figure 4.10.

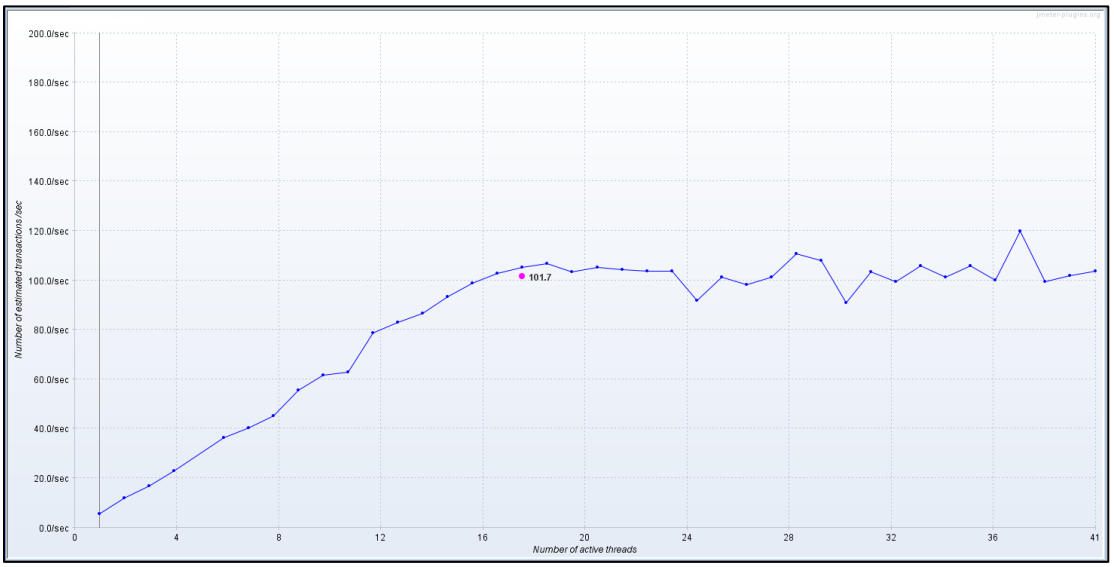

Figure 4.8: PAC Performance – Transaction Throughput vs. Time (accessing through vendor-specific API).

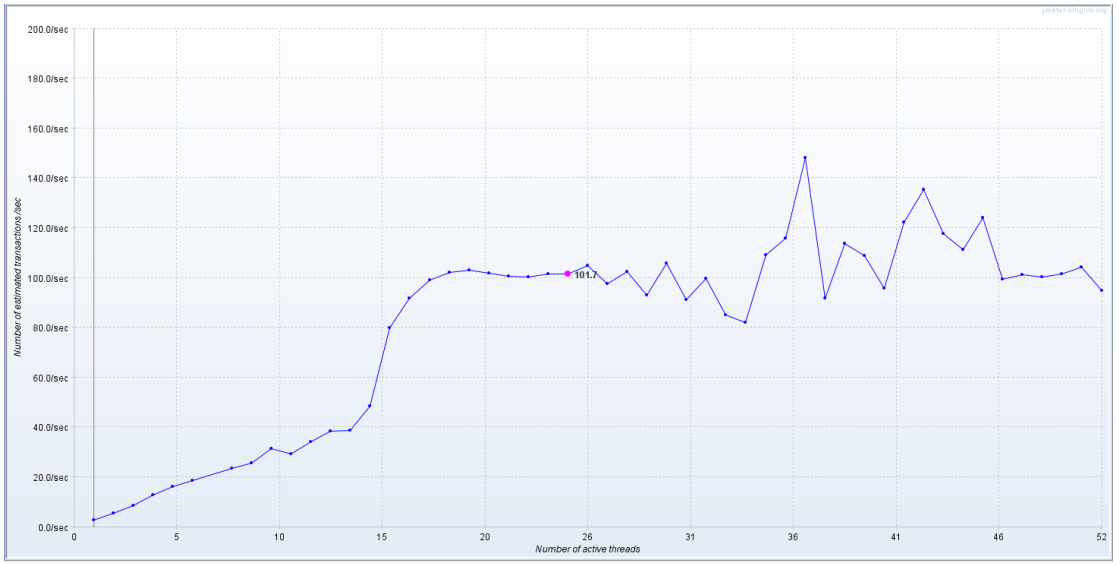

Figure 4.9: PAC Performance – Transaction Throughput vs. Time (accessing through PAC).

Even though, the version which uses the PAC shows lots of fluctuations over the time, when we look at the overall average line for all the three operations show more or less identical behaviors. Therefore, as PAC is just a wrapper over the vendor-specific SDKs, application will not experience any major performance bottlenecks, by introducing it to

the data layer. Another observation in Figure 4.11 is a cache miss occurred around 20:29 resulting in an I/O operation to the underlying database to fetch and store the data.

Figure 4.10: PAC Performance – Response time vs. Time (accessing through vendor-specific APIs).

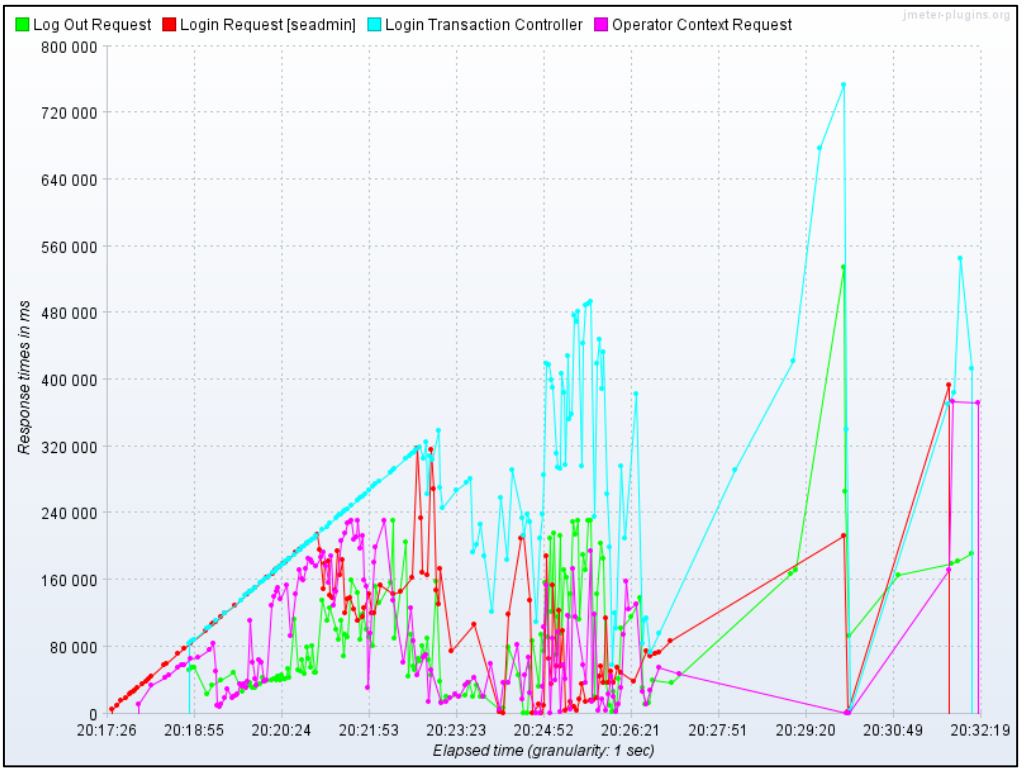

Figure 4.11: PAC Performance – Response time vs. Time (accessing through PAC).

4.6 PAS Performance

PaaS Aggregator storage provider test scenarios were executed on the Windows Azure platform using the Azure Blob service. One version of the data access layer in API server was configured to use the Azure SDK to access static contents stored in the blob while the other version initiated an instance from PaaS Aggregator *StorageProviderFactory* that ultimately access the blob based on the configurations.

An implementation of *IStorageLogAccessor* was injected into the PAS, which also utilized the blob to store master log items. They were stored in the same storage account, but in a separate container. Table 4.4 shows the JMeter parameters used when executing test scripts. JMeter scripts were added to send four types of requests against the API server as follows:

- Upload some binary content to the server
- Download some predefined binaries available in the blob
- List down file names stored in the blob
- Delete some items from the blob, if available

Table 4.4: StorageAccessor test setup parameters.

Parameter	Value
Number of threads (Users)	1,000
Ramp-up period (S)	10
Loop count (Number of time each thread gets executed)	3
Number of read requests per thread	4
Number of files sent in upload request	20

4.6.1 PAS - Throughput comparison

Figure 4.12 and 4.13 show the throughput data collected over the period of test execution. When the storage is accessed without using PaaS Aggregator, we can see that the overall transaction throughput is around 30% less compared to the version which accesses the storage via vendor-specific APIs . We believe following factors affecting this fluctuation:

- Every time an instance of *IStorageAccessor* is initiated from PaaS Aggregator, it checks whether the currently accessible storage is in sync. The way it does this is by comparing the latest master log and latest local log item stored. No query functionality is provided in the Azure blob SDK to retrieve the last blob

item added into a container. The only way to achieve this is to load all the blob item metadata to the memory and query the metadata in memory. Hopefully Microsoft will provide a better query facility in their future SDK versions to facilitate this. Furthermore, it would have been better if we introduced the functionality into *IStorageProvider* to check for this sync status, if only the current storage provider is switched to another one due to inaccessibility.

- The other possible performance bottelenck is the choice of *IStorageLogAccessor* storage provider. As it is the API consumer's responsibility to choose a reliable and efficient provider for this, it is very important to choose the provider wisely from the point of view of the network infrastructure (i.e., at least within the same region as the main storage), as well as the SDK capability to provide enough capability to query the data stored in it.

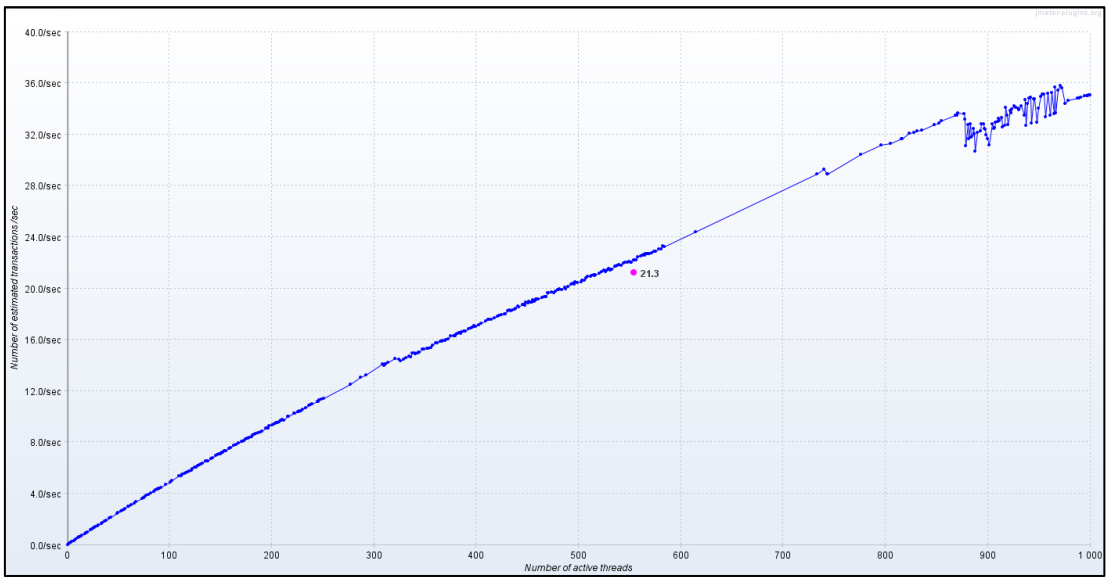

Figure 4.12: PAS Performance – Throughput vs. Active threads (accessing through vendor-specific API).

4.6.2 PAS – Response Time Comparison

Figure 4.14 and 4.15 show the response for file upload and download operations. While the PaaS Aggregator enabled version shows an increase in response time, the other version shows an evenly distributed latency. This clearly emphasizes the two possible performance overheads that we discussed in Section 4.6.1 which affected the response times as well. These performance data could have been much better for PaaS Aggregator, if the log storage layer was configured to use Amazon S3 over Windows Azure Blob storage as S3 supports metadata querying via Amazon SimpleDB [34],

which would have made the querying very efficient. When the latest log item stored in the master log provider is queried by the StorageFactory (as there are no metadata querying facility provided by Azure SDK until now) it has to retrieve all the blob item properties to the server memory and then query for the latest item. This is very costly; hence, affects the overall performance of the request. Therefore, we have learnt that by incorporating an efficient log storage provider into the PaaS Aggregator, the possible minor bottlenecks that we saw during the test scenarios could easily be removed.

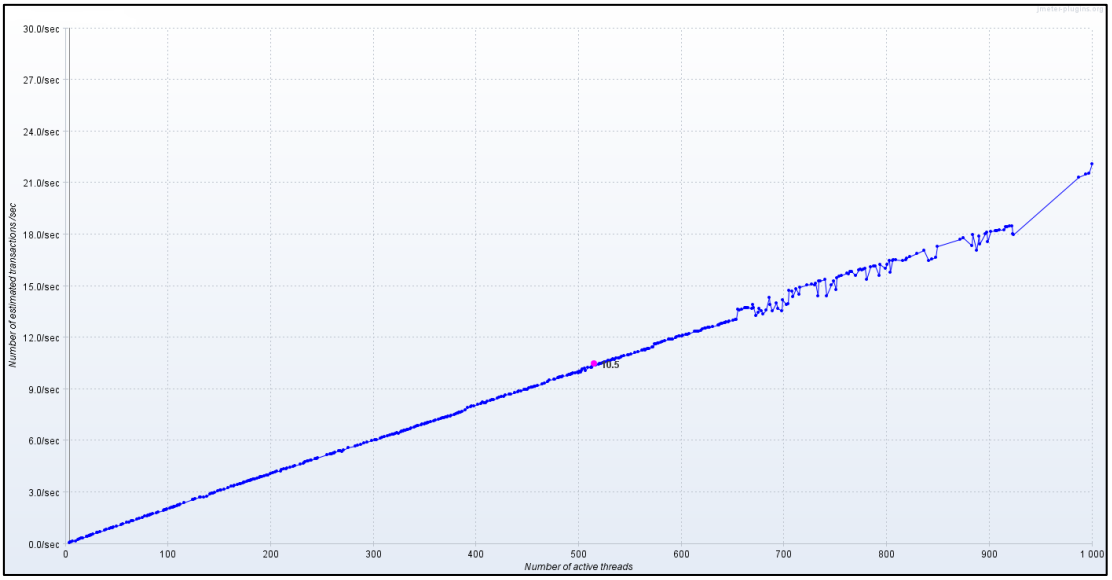

Figure 4.13: PAS Performance – Throughput vs. Active threads (accessing through PAS).

4.7 PaaS Aggregator – Overall performance

So far, we tested each service individually to investigate whether they introduce any performance bottlenecks to the data access layer. I1 this Section, we test how the application performs when all the provided services are accessed through PaaS Aggregator. To test the overall performance, we agglomerate all the types of requests issued to test each service individually and prepared a composite workload that issues a mix of requests randomly against the server. Table 4.5 shows the JMeter test setup parameters used in the evaluation. Even though the number of write operations are set to 11, each file upload request will be issued with 20 binary files.

Table 4.5: PaaS Aggregator – overall performance test setup parameters.

Parameter	Value
Number of threads (Users)	1,000
Ramp-up period (S)	10
Loop count (Number of time each thread gets executed)	3
Number of read requests per thread (Database, Storage and Cache)	9
Number of write requests per thread (Database, Storage and Cache)	11

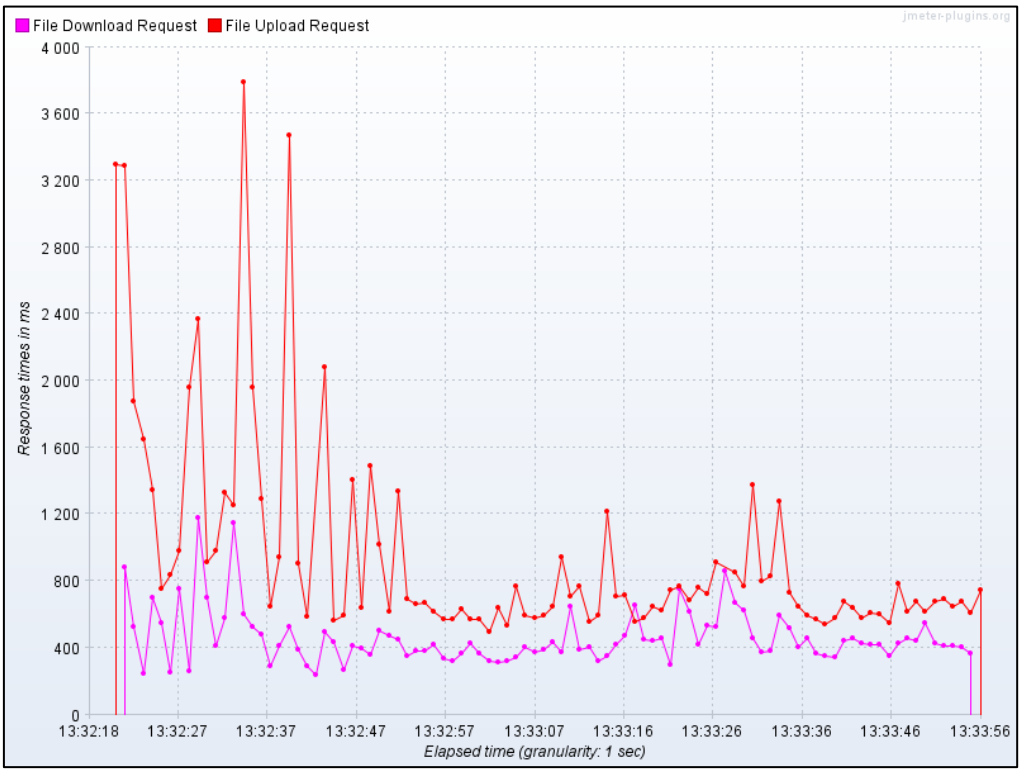

Figure 4.14: PAS Performance – Response time vs Time elapsed (accessing through vendor-specific APIs).

4.6.3 PaaS Aggregator – Overall Throughput Comparison

Figure 4.16 and 4.17 show the transaction throughput comparisons when the services are accessed through vendor-specific APIs and PaaS Aggregator, respectively. Three lines shown in each graph show the transaction throughput variation of each service during the period of analysis. Maroon, Blue, and Green colors show the throughput variations of PAC, PAD and PAS, respectively.

Clearly the throughput values achieved for *CacheAccessor* was much higher compared to the other two accessors. This is because there were no CPU or memory intensive work going on while accessing Cache.

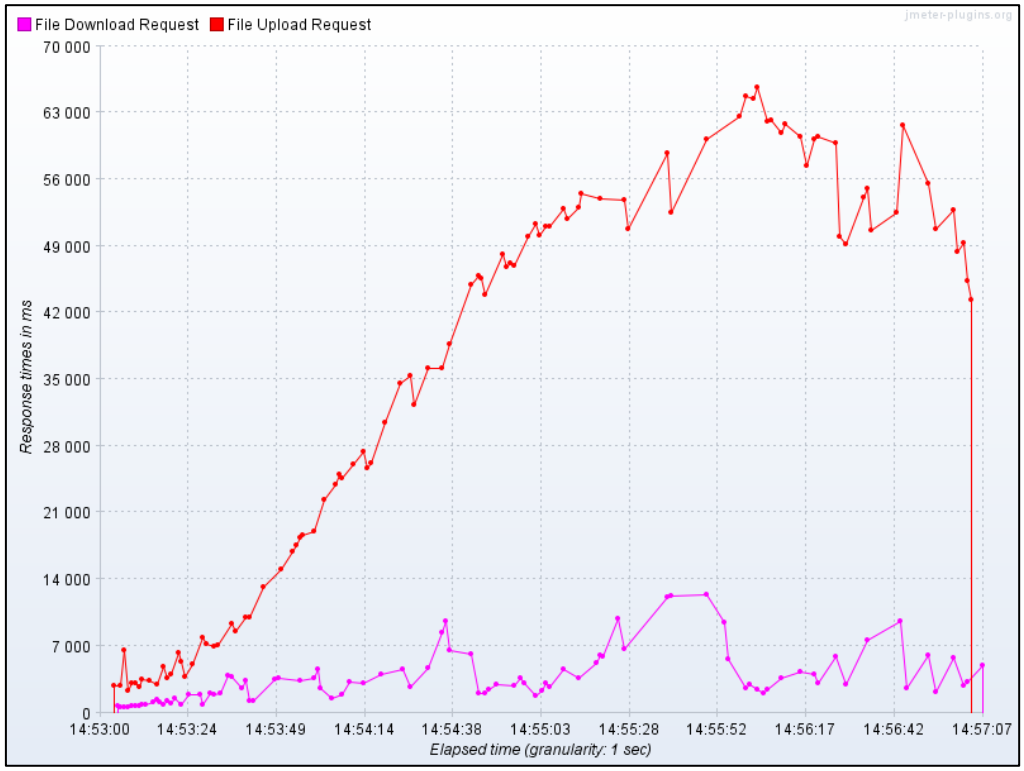

Figure 4.15: PAS Performance – Response time vs Time elapsed (accessing through PAS).

Moreover, most of the login and log out requests must have been issued during the start and end which resulted in a relatively low throughput values during those times. As the throughput line of PAC stands out with high throughput values in the graph, the fluctuations of the other two lines were not noticeable in this graph. However, we can see that the overall transaction throughput values are around 6 to 8% higher when accessing the services through vendor-specific APIs. The main reason for this is the choice of log accessor storage provider and the absence of periodic synchronization service as we discussed in previous sections.

4.7.1 PaaS Aggregator – Overall Response Time Comparison

As seen in Figure 4.18 and 4.19, the overall response times, when accessing services through PaaS Aggregator are around 20% higher in average when compared to the version which uses cloud vendor-specific APIs. The main contributing factor for these numbers is the file upload request. The response time for this particular request, in average, is around 40% higher in comparison. Moreover, some of the high volume data retrieval requests contributed considerably to the overall response time distribution (network latency may have also impacted).

Except those two types of requests, response time distributions of all the other requests show identical behaviors with closer response times. We believe, by choosing an efficient storage provider for the services in PaaS Aggregator, these response times can be drastically reduced, thus by reducing the overall response times.

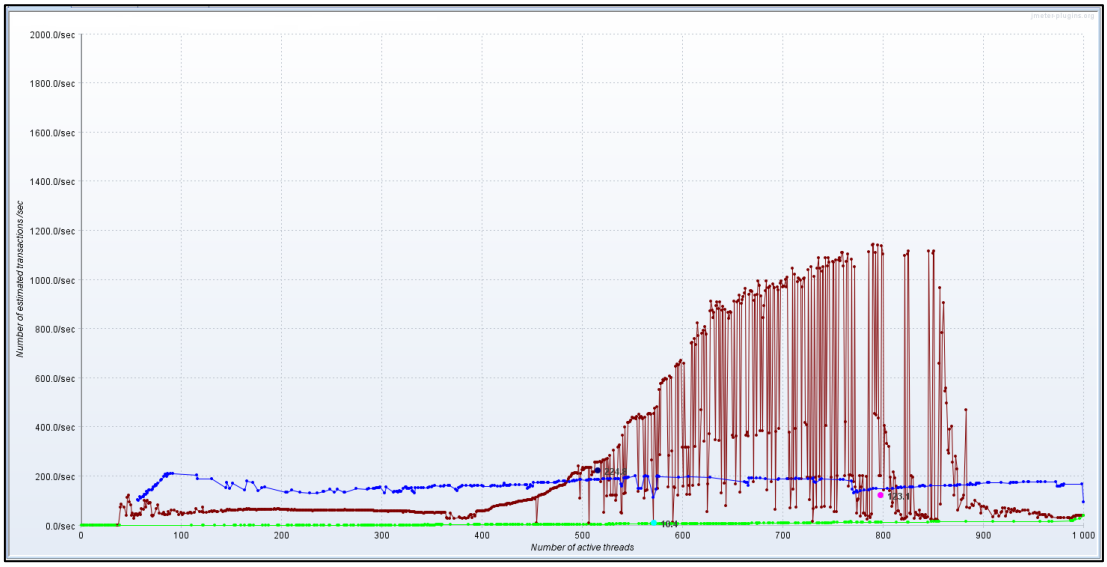

Figure 4.16: PaaS Aggregator Performance – Transaction Throughput vs. Time (accessing through vendor-specific API).

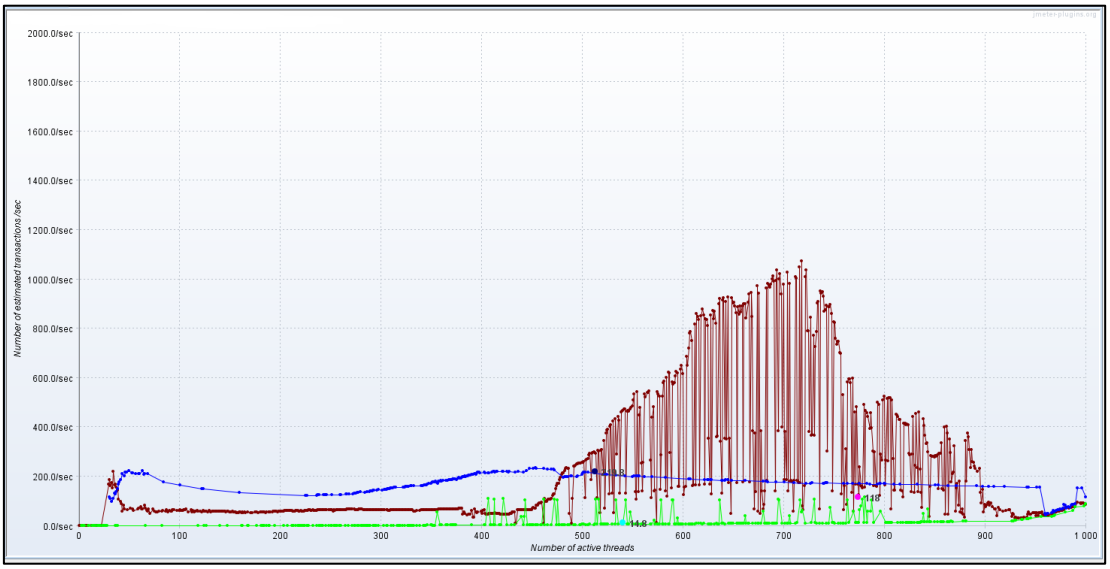

Figure 4.17: PaaS Aggregator Performance – Transaction Throughput vs. Time (accessing through PaaS Aggregator).

4.8 Summary

We investigated how PaaS Aggregator performed under a stressed workload when comparing with a conventional SaaS application that is tightly coupled with a given

provider. We also discussed how the proposed multi-cloud library would perform when the current active provider is not accessible any more. This on the fly synchronizing is only available in PAD as of now, but there are enough provisions added into PAS, which will enable us to maintain consistency over time. We observed that the performance overhead that PAD and PAC bring into the data access layer is considerably low while there was a considerable performance overhead introduced by PAS due to its extensive logging mechanisms.

As we figured out during some of the test scenarios, the overall throughput and response times may be affected by the extended logging functionality that is built into PaaS Aggregator. We discussed how inappropriate selection of log storage providers might slow down the requests, thus by affecting the overall performance of the application. We also discussed it is advisable to maintain required background services in your SaaS environment to periodically sync the environments using the utilities provided in the PaaS Aggregator itself. We also saw that overall performance of the system when simulating an environment, which access all the resources through PaaS Aggregator is more or less identical, if we configured the system to access the resources otherwise.

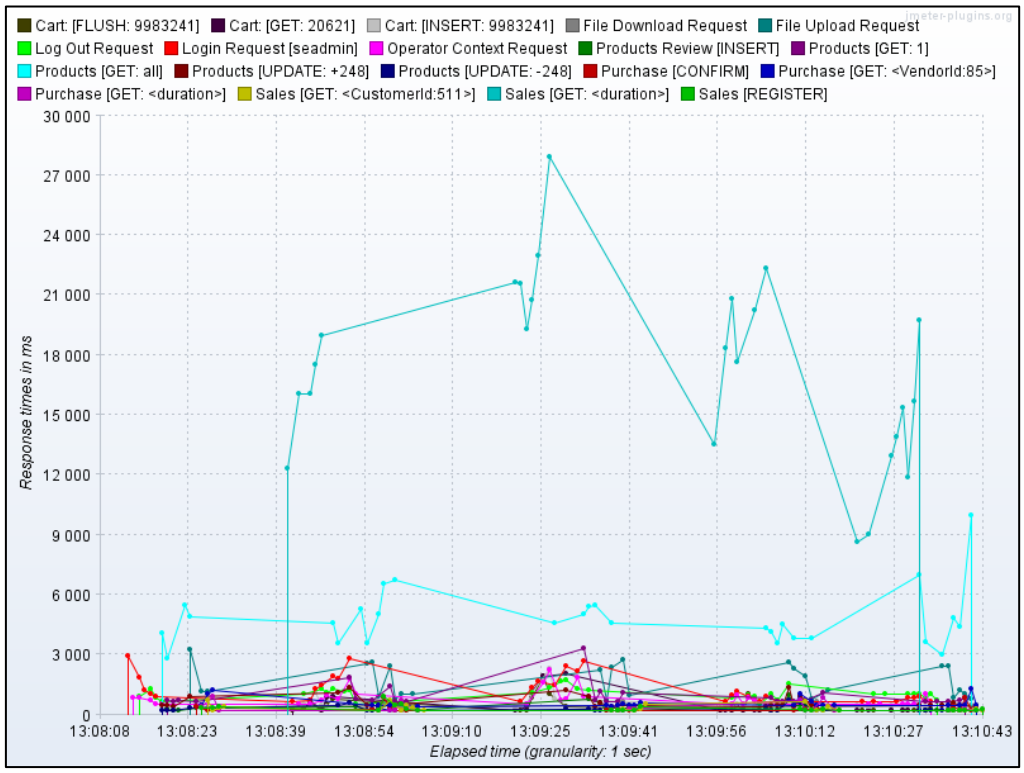

Figure 4.18: PaaS Aggregator Performance – Response Time vs. Time (accessing through vendor-specific API).

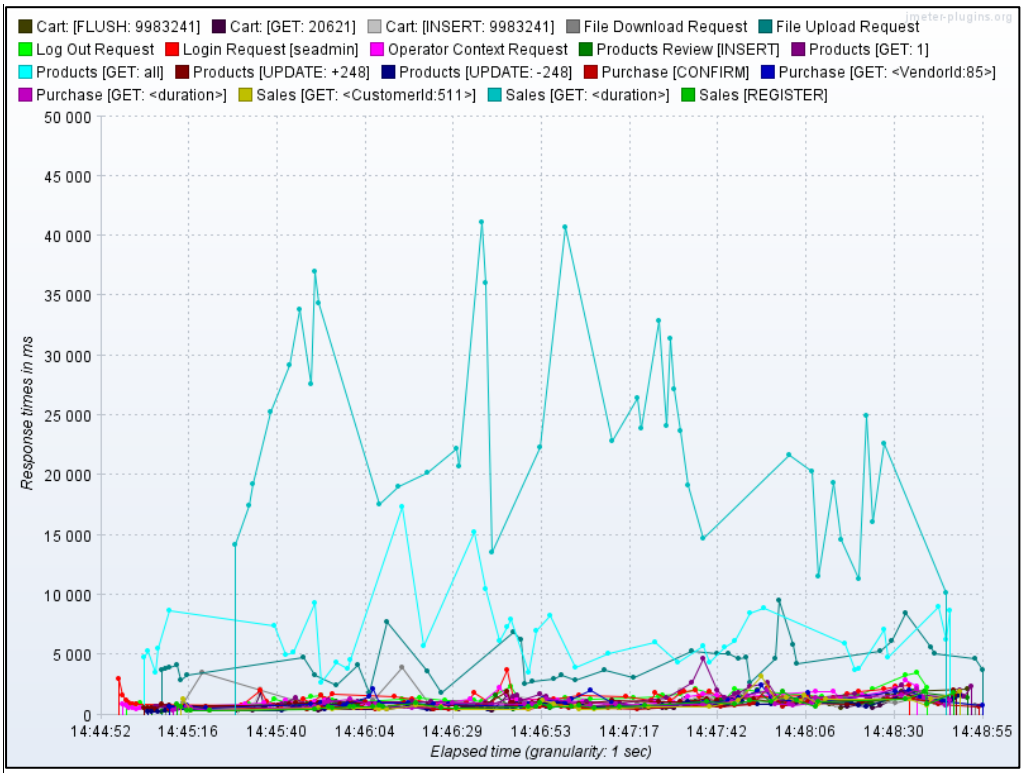

Figure 4.19: PaaS Aggregator Performance – Response Time vs. Time (accessing through PaaS Aggregator).

CHAPTER 5

CONCLUSIONS

5.1 Summary

We proposed a simplified approach to enhance the availability of SaaS applications by providing a framework to facilitate the SaaS application developers to utilize different types of services without worrying about their corresponding vendor-specific implementations. The framework is developed in the form of a library so that it eventually becomes a part of the client application. The reason why this framework is exposed as a library instead of a hosted REST API is to get rid of the vulnerability of single point of failure. Apart from the simplified and unified service definitions, the framework is also capable of migrating among different providers for a given service when and if required. It is also equipped with required utilities to keep the information consistent among different providers. Although the current framework is only capable of handling limited number of platform services, the model can be adapted and extended to support many services in future. Our ambition is to invite platform-level service providers as well as open source contributors to provide adapters that can be plugged into the framework so that at some point, developers will never be required to worry any vendor-specific SDK at all. They can utilize this framework to do the implementation once and then decide which PaaS provider to use based on a single configuration file.

We presented an overall architecture of the PaaS Aggregator and how it works. One of the key characteristics of the proposed architecture is the ability to switch among different configured providers for a given service and the required information that it provides for synchronizing purposes. Another important design decision that we have built into in PaaS Aggregator overall architecture is the focus on extensibility. This is very important as new PaaS providers can later provide implementations for their APIs. We also discussed how most of the logging functionality is outsourced as a responsibility of the SaaS application for the betterment of performance and reliability of the overall application. It is essential to choose a storage provider which provides those characteristics (especially metadata querying), as it can affect the overall performance of the application. We also looked into how the background

synchronization among different providers should be done in *dbaccessor* and *storageaccessor*. On the fly synchronization in database provider is available in PaaS Aggregator and as of now, it cannot be switched off as the consistency is vital when it comes to RDMSs. However, this process is not possible while accessing storage service as the binary data synchronization is costly. Instead, PaaS Aggregator provides information to decide whether the currently accessible storage is in sync or not. SaaS applications can decide the way forward based on that. For example, if the storage is not in sync, SaaS application can use the storage provider in read only mode until the storage is back online. At the same time, PaaS Aggregator provides a utility to synchronize storage providers defined in the configuration file. In order to make the switching among different providers smooth, it is recommended to carry out synchronization in a background service asynchronously.

As the solution for the problem at hand is to provide a thin layer into the application data layer, it was important to verify that we do not introduce considerable overhead to the application, thus by reducing the overall performance. Therefore, extensive stress tests were carried out to test all the three services individually, as well as together. We compared the transaction throughput of the server and response time of the requests with and without the proposed PaaS Aggregator. Throughput and latency characteristics of the proposed aggregator reduced marginally compared to the native application, due to extensive logging available in PaaS Aggregator, if the log provider is not chosen wisely. This can be overcome by choosing a log storage provider that supports metadata querying. We observed that using Windows Azure Blob (which is not capable of providing metadata querying) as the storage provider in PAS resulted in 40% of transaction throughput drop. Moreover, it is recommended to choose the log storage provider to be in the same region as the main service provider. Nevertheless, when we tested the overall performance characteristics when all the services are accessed as a mix, it turned out that the application overall performance averaged out over the time period thus remained identical. Therefore, we can state that the reliability and availability aspects overweigh the minor overheads that PaaS Aggregator introduces into the applications.

In this way, our effort in providing a thin layer of abstraction in the data layer by means of a library is providing us with positive results when it comes to alleviating vendor

lock-in syndrome in PaaS layer and improving the overall availability of the application. Although PaaS Aggregator provides these possibilities with three of the most popular services used in PaaS layer, we can easily extend the same architecture to support more vendors for the existing services as well as adding more services into it.

5.2 Research Limitations

Our main objective was to provide a thin abstraction layer that is capable of providing a unified API, thus by getting rid of tight coupling in SaaS applications to a particular PaaS provider. However, by providing a library we assume that the hosted application will always be online to migrate the data layer from provider to another. However, there are other platform services like compute, schedulers, traffic managers, etc., which cannot be migrated to a totally different service provider by this kind of a library solution. Therefore, our PoC PaaS Aggregator implementation is capable of taking care of platform services as long as the environment it resides in is stable and available. In fact, solutions like Jelastic [29], OpenCloudWare [27], and PaaS Manager [20] are capable of handling those platform services but these solutions are REST based hosted services. Therefore, they are vulnerable to single point of failure, defeating their own purpose of providing high availability. Consumers will have to depend on their services, thus by creating another coupling in between SaaS and Platform layers. One of the outcomes of this thesis is not only to provide an abstraction layer, which is capable of providing the simplified interfaces which are provided by the hosted providers like PaaS Manager [20], but also not to introduce any single point of failure risk into the SaaS application by doing so.

PAD is only capable of handling operations in MSSQL databases. There is no support for other database engines in PaaS Aggregator. Even though this seems like a possible future work, this is tricky when it comes to synchronizing across different providers, as well as query execution against different database engines. Furthermore, as we saw in Section 4.6.2, if we choose an inefficient master log storage provider for PAS, it will affect the overall performance of the PaaS Aggregator. Therefore, we can consider that unavailability of an optimized master log storage provider built into PaaS Aggregator is another limitation.

5.3 Future Work

Our work can be extended along the following directions:

Extending PaaS Aggregator to support more programming environments

As of now, PaaS Aggregator is implemented using .Net framework limiting its use to .Net supported languages. The reason why we started with a .Net library is to support the existing SaaS application which is .Net based. It will be advantageous, if we could provide support for mainstream languages such as Java and NodeJs. Even with .Net-based library, there are some improvements that we can still provide. For example, *dbaccessor* in PaaS Aggregator is only capable of issuing SQL queries through ADO.Net interface. With the rising popularity of ORM tools, we need to focus on adding compatibility with ORMs like Entity Framework, Dapper.Net, and nHibernate. Furthermore, we are planning to deploy the PaaS Aggregator components as Nuget packages where the library can be referenced without any hassle and manage the versions in a smooth manner. There will be four Nuget packages at the end, which look like, PaaSAggregator.Core, PaaSAggregator.Database, PaaSAggregator.Cache, and PaaSAggregagor.Storage. They can be referenced separately as long as the dependent assemblies are available during the runtime.

Adding more services and provider support

It is useful to extend the solution to support for other platform services like Network traffic manager, CDN, Active Directory, Service bus, etc., in PaaS Aggregator. This requires some research into various types of functionalities provided by different service providers and coming up with a standardized API. Also at the same time, we should constantly extend the functionalities already available in existing supported services. For example, the storage provider is only capable to handling simple binary content upload and downloads, we need to extend it to support complex functionalities like appending for log file manipulations.

Our PoC solution need to be extended to work on various other service providers beyond Microsoft Azure and Amazon. Other service providers may contain different API conventions, which need to be taken into account when providing support for them. These should happen without diluting the important architectural decisions taken during the initial implementations.

It is also important to come up with a mechanism to handle upgrading of vendor specific SDKs inside PaaS Aggregator. As of now, it refers to the latest versions, but with the time, we need to come up with a strategy to smoothly upgrade those SDKs.

Providing vendor-specific functionalities that cannot be abstracted

As the PaaS Aggregator abstracts the vendor-specific implementations, it only exposes common functionalities that are available across majority of the PaaS providers. However, different cloud providers may provide specific features that are not available in other providers. Those features are expected not to be exposed through the API in PaaS Aggregator by design. Therefore, we need to come up with a ponying mechanism to allow the applications to utilize these vendor-specific features, if required.

Providing an implementation which uses provider native language

PaaS Aggregator acts as a mediator in the data layer of the application where the API consumers need to adhere to the contracts provided in it. Therefore, if an application already using AWS SDK is planning to migrate to PaaS Aggregator, the data layer needs to be changed so that the correct contracts are invoked. If this API routing could take place after the vendor-specific methods are called, then the amount of work that needs to done by the PaaS Aggregator consumers will become less. It the meantime, it may improve the performance in the application as we are using the native language of the cloud provider.

References

[1] R. Harms and M. Yamartino, "The economics of the cloud," [Online]. Available: http://www.microsoft.com/presspass/presskits/cloud/docs/The-Economics-of-the-Cloud.pdf. [Accessed: 20-Feb-2016]

[2] M. Armbrust et al., "Above the clouds: A Berkeley view of cloud computing," Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS 28.13 (2009): 2009.

[3] C. David, P. Neves, and P. Sousa, "PaaS manager: A platform-as-a-service aggregation framework," Computer Science and Information Systems, vol. 11, no. 4, 2014, pp. 1209-1228.

[4] S. Kolb and G. Wirtz, "Towards Application Portability in Platform as a Service," in Proc. 8th Symp. Service-Oriented System Engineering, April 2014.

[5] Intel IT Center, "Platform as a Service (PaaS) Drives Cloud Demand," Intel Whitepaper, August 2013, [Online].Available:http://www.intel.com/content/dam /www/public/us/en/documents/white-papers/cloud-computing-paas-cloud-demand -paper.pdf

[6] W. Kim, S. D. Kim, E. Lee, and S. Lee, "Adoption issues for cloud computing," in Proc. 7th International Conference on Advances in Mobile Computing and Multimedia, ACM, 2009.

[7] E. Anderson and D. M. Mitchell, "Hype cycle for cloud computing," Gartner Inc., Stamford (2011): 71.

[8] Why PaaS growth is disproportional to other sectors. (2014, Oct 30). [Online]. Available: http://research.gigaom.com/2014/10/why-paas-growth-is-disproportional-to-other-sectors/

[9] L. Wang, G. Von Laszewski, M. Kunze, and J. Tao, "Cloud computing: a perspective study," J New Generation Computing 28, no. 2 (2010): 137-146.

[10] T. Kurze, M. Klems, D. Bermbach, A. Lenk, S. Tai, and M. Kunze, "Cloud federation," in Proceedings of the 2nd International Conference on Cloud Computing, GRIDs, and Virtualization (CLOUD COMPUTING 2011), IARIA, Sep 2011.

[11] C. Kanaracus, "PaaS Market to Reach $14 Billion by 2017, IDC Says," InfoWorld (November 8, 2013). [Online]. Available: http://infoworld.com/d/cloud-computing/paas-market-reach-14-billion- 2017-idc-says-230440

[12] “Cloud-Related Spending by Businesses to Triple from 2011 to 2017.” IHS (press release) (February 14, 2014). [Online]. Available: http://press.ihs.com/press-release/design-supply-chain/cloud-related- spending-businesses-triple-2011-2017

[13] D. Sullivan “PaaS Providers List: 2014 Comparison and Guide.” Tom’s IT Pro (January 31, 2014) with additions. [Online]. Available: http://www.tomsitpro .com/articles/paas-providers,1-1517.html

[14] R. Cowan, “Tortoises and hares: Choice among technologies of unknown merit,” The Economic Journal, Jan. 1991. [Online].
Available: http://links.jstor.org/sici?sici=0013-0133(199107)101%253A407%25 3C801%53ATAHCAT%253E2.0.CO%253B2-S

[15] S. Soltesz et al., “Container-based operating system virtualization: a scalable, high performance alternative to hypervisors,” ACM SIGOPS Operating Systems Review. Vol. 41, No. 3, ACM, 2007.

[16] A. Verma et al., “Large-scale cluster management at Google with Borg,” in Proc. 10th European Conference on Computer Systems, ACM, 2015.

[17] T. Aubonnet and N. Simoni, “Self-Control Cloud Services”, 2014 IEEE 13th International Symposium on Network Computing and Applications (NCA), pp. 282-286, 2014.

[18] C. Goncalves, D. Cunha, P. Neves, P. Sousa, J. P. Barraca, and D. Gomes, “Towards a Cloud Service Broker for the Meta-Cloud,” in Proc. 12th Conferencia sobre Redes de Computadores, November 2012, Aveiro, Portugal.

[19] B. S. Lee, S. Yan, D. Ma, and G. Zhao, “Aggregating IaaS Service,” In Proc. 2011 Annual SRII Global Conference, Connecting Services to Science & Engineering, San Jose, California, Mar. 2011.

[20] D. Cunha, P. Neves, and P. Sousa, “PaaS manager: A platform-as-a-service aggregation framework,” Computer Science and Information Systems vol. 11, issue 2, 2014

[21] D. Cunha, P. Neves, and P. Sousa, “Interoperability and portability of cloud service enablers in a PaaS environment,” in Proc. 2nd International Conference on Cloud Computing and Services Science, SciTePress, pp 432–437.

[22] F. Paraiso, P. Merle, and L. Seinturier, “soCloud: A service-oriented component-based PaaS for managing portability, provisioning, elasticity and high availability across multiple clouds,” Computing, 2014

[23] Kavis Technology Consulting, “Responsibilities in the Cloud,” [Online] Available: http://www.kavistechnology.com/blog/responsibilities-in-the-cloud/

[24] C. Lin, “Scale a web app in Azure App Service,” [Online] Available: https://azure.microsoft.com/en-gb/documentation/articles/web-sites-scale/

[25] “Web Application Hosting,” [Online] Available: https://aws.amazon.com/architecture/

[26] “What is Docker’s architecture?,” [Online] Available: https://docs.docker.com/engine/understanding-docker/

[27] “OpenCloudware,” [Online] Available: http://www.opencloudware.org/bin/view/Discover/OpenCloudware

[28] D. Bernstein, “Containers and Cloud: From LXC to Docker to Kubernetes,” Cloud Computing, IEEE, vol. 1, no. 3, pp. 81-84, 2014.

[29] Jelastic: Multi-Cloud PaaS and CaaS for Business, [Online] Available: https://jelastic.com

[30] Key Vault: Enhance data protection and compliance, [Online] Available: https://azure.microsoft.com/en-us/services/key-vault/

[31] API implementation guidance, [Online] Available: https://docs.microsoft.com/en-us/azure/best-practices-api-implementation, 2016

[32] Cache aside pattern explained, [Online] Available: https://i-msdn.sec.s-msft.com/dynim g/IC709568.png

[33] Microsoft Corporation, “AdventureWorks Sample Databases,” [Online] Available: https://msdn.microsoft.com/en-us/library/ms124501(v=sql.100).aspx

[34] “Indexing and Querying Amazon S3 Metadata with Amazon SimpleDB,” [Online] Available: https://aws.amazon.com/items/1465?externalID=1465

[35] “Testing and measuring performance with Apache JMeter,” [Online] Available: https://github.com/apache/jmeter

[36] Microsoft Corporation, “Windows Azure Status History,” [Online] Available: https://azure.microsoft.com/en-us/status/history/

[37] Lessons learned from recent cloud outages (2013). http://tinyurl.com/qz5maey

[38] Nuget: Package Manager for the Microsoft development platform. https://www.nuget .org/

[39] A. Shalloway and J. Trott, “The Façade Pattern,” in *Design Patterns Explained Simply*, vol. 2, 2002, pp 87 – 93.