Achieving Performance for Compute Intensive Services in Cloud SaaS

JAWAD MUSTAFA
Achieving Performance for Compute Intensive Services in Cloud SaaS

Uppnå prestanda för beräkningsintensiva tjänster i Cloud SaaS

Jawad Mustafa

jawadmu@kth.se

Master of Science Thesis

External Supervisor: Mats Bergström
Examiner and Internal Supervisor: Johan Montelius

Software Engineering of Distributed Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

June 18, 2016
List of Acronyms

CIS  Compute Intensive Services
IaaS  Infrastructure as a Service
PaaS  Platform as a Service
SaaS  Software as a Service
AWS  Amazon Web Services
API  Application Programming Interface
SLA  Service Level Agreement
EC2  Elastic Compute Cloud
ELB  Elastic Load Balancing
ASG  Auto Scaling Group
CF   Cloud Formation
LC   Launch Configuration
AMI  Amazon Machine Images
S3   AWS Simple Storage Service
CRT  Cache Routing Table
EU   European Union
Abstract

Enterprise business applications are following a rapid trend to move their solution model from specific organizational users to global access. This creates opportunities for organization to expand geographically with partners and resellers but it also increases simultaneous user requests. These solution models are commonly based on many data-set states and a web applications that perform multiple tasks in its work-flow including compute intensive requests to separate Compute Intensive Services (CIS). This research is based on these solution models and a special type of CIS that build and reuse in-memory cache to reduce response latency. Performance factors like additional simultaneous requests and cache building requests can increase response latency if not enough CIS are available to handle load peaks. Additional compute services can be added to the infrastructure but such solutions increase cost and these additional services are not required all the time.

Main goal of this research is to study and design an architecture to achieve cost-effective performance for solution model of CIS. First, a study have been performed on dedicated servers approach, to find impact of these performance factors. Next, a prototype Software as a Service (SaaS) architecture has been presented which detects and reduces load peaks created by performance factors. SaaS architecture has been designed by using cloud computing products of Amazon Web Services (AWS). Few supplementary components have been identified and developed during research to overcome shortcomings of standard cloud products. It aims to reduce load peaks with scalability and elasticity. Experiments have been performed on SaaS architecture to find its advantages and limitations for solution model of CIS.

An essential part of this research are two solution proposals, which are based on designed SaaS architecture. First solution proposal has been made for multi-tenant architecture because multi-tenancy can help to enhance cost-effective performance. The second solution proposal has been made to achieve low latency response by optimizing usage of in-memory cache. This optimization can help enterprises to change data-set states more often and achieve predictable low latency. It also adds flexibility in SaaS architecture to reduce number of required servers.
Sammanfattning


Huvudsyftet med denna forskning är att studera och utforma en arkitektur för att uppnå en kostnads effektiv prestanda för denna lösning modell ac CIS. För det första har en studie utförts på tillvägagångssätt dedikerade servrar för att hitta effekten av dessa resultatfaktorer. Därefter har en prototyp ac SaaS arkitektur presenterats som detekterar och minskar belastningstoppar som skapas på grund av prestandafaktorer. SaaS arkitektur har utformats med hjälp av molnprodukter av ac AWS. Få kompletterande komponenter har identifierats och utvecklats under forskning för att övervinna bristerna hos vanliga molnprodukter. Det syftar till att minska belastningstoppar med skalbarhet och elasticitet. Experiment har utförts på SaaS arkitektur för att hitta sina fördelar och begränsningar för lösning modell ac CIS.

En väsentlig del av denna forskning är lösningsförslag som bygger på utformade SaaS arkitektur. Första lösningsförslag har gjorts för flera organisationer arkitektur eftersom flera organisationer kan bidra till att förbättra kostnads effektiv prestanda. Det andra förslaget lösningen har gjorts för att uppnå låg respons svar genom att optimera användningen av cache i minnet. Denna optimering kan hjälpa företag att ändra data-set tillstängande oftare och det kan bidra till att ge förutsägbara låg latens för slutanvändaren från ac CIS. Det kan också lägga till flexibilitet i SaaS arkitektur för att minska antalet nödvändiga servrar.
Acknowledgements

First of all, thanks to ALLAH, the most merciful and the most beneficent for giving me strength, courage and wisdom that allowed me to complete this thesis.

I would like to express my gratitude to Johan Montelius for giving me an opportunity to work under his supervision. His valuable guidance and feedback helped me to pass through all problems and phases of this thesis project. I would also like to thank Mats Bergström and other senior members for their valuable technical support and guidance throughout the project.

Finally, I would like to thank my family for their encouragement and support during whole thesis project.
List of Figures

1.1 using compute intensive service ............................ 6
1.2 cache building impact on latency .......................... 8
1.3 work load graph ............................................. 9

2.1 servers of web application and compute intensive service ... 15
2.2 in-memory cache usage ....................................... 16
2.3 in-memory cache latency ..................................... 18
2.4 multiple infrastructure graph ................................. 19
2.5 scale-up and scale-out ....................................... 23
2.6 scale-out and compute intensive demand .................... 24
2.7 multi-tenant architecture ..................................... 24

3.1 each region load ............................................... 27
3.2 region specific demand with in-memory cache .............. 28
3.3 single infrastructure for all regions ........................ 29
3.4 single infrastructure demand ................................. 30

4.1 ASG Scaling ...................................................... 34
4.2 saas basic architecture ........................................ 36
4.3 queue-time(MMC) ............................................. 38
4.4 scaling service colaboration ................................. 39
4.5 aws conceptual support high availability .................. 40
4.6 SaaS Architecture for CIS .................................... 43
4.7 saas experiment single scaling, fast scale-out .............. 45
4.8 saas experiment multiple scaling, lazy scale-out .......... 45
4.9 saas experiment heavy cache building ...................... 46
4.10 multi tenant usage ........................................... 49
4.11 state heavy proposal .......................................... 53

B.1 CRT Table 2 .................................................. 62
Contents

1 Introduction 5
  1.1 Performance Factors of Compute Intensive Services for Enterprise Web Application 7
  1.2 Research Approach and Methods 10
  1.3 Motivating Scenario 11
  1.4 Boundaries and Assumptions 12

2 Background 13
  2.1 Load Trends of Enterprise Web Application 13
  2.2 Load Balancing For Compute Services 14
  2.3 In-Memory Cache Usage and Benefits 15
  2.4 Performance Impact of In-Memory Cache 17
  2.5 Performance Impact of Simultaneous Requests 19
  2.6 Infrastructure Choices for Compute Intensive 20

3 Experiments and Analysis of Dedicated Servers 26
  3.1 Study of Region Specific Servers 27
  3.2 Study of Single Infrastructure Servers 29
  3.3 Evaluation and Limitations 31

4 SaaS Architecture Design And Performance Analysis 33
  4.1 Applicable Cloud IaaS Products 33
  4.2 Basic SaaS Architecture 35
  4.3 Supplementary Features and Components 35
  4.4 Implementation 40
  4.5 Architecture and Runtime View 42
  4.6 Experiment and Solution Evaluation 44
  4.7 Evaluation and Proposal for Multi Tenant (Future Work) 49
  4.8 Evaluation for State Heavy Scenarios 51
  4.9 Achieving Low Latency with SaaS Scalability (Future Work) 52

Appendices 58
Chapter 1

Introduction

Enterprise and business applications use web platform to provide organizational and global access to business data and services. When organizations and businesses grow with regional offices, resellers and partners then load on these web applications increases intensely. A huge number of users can register and use web application for business data and services. This demands a need to handle large number of users which is usually performed by using load balancing and clustering techniques [2]. The variance in workload also increases because number of simultaneous user requests can increase and decrease depending on working hours of each office or region [1]. These performance factors should be evaluated as a main part of architecture solution of these web application. [13].

These web applications solve different business scenarios that can have simple data requests or complex problems such as engineering calculations, analytical computation and solutions for combinatorial optimization. A simplified user interface is provided for users to get solutions of these problems. Integrated or directly connected CIS are used by web application to handle complex problems and provide solutions. These services require computational and memory resources to complete execution of requested problem and return solution. Organizations have multiple infrastructure options for these web applications and CIS. Highly distributed and large-scale infrastructures are commonly used which include on-premise, fixed servers or cloud hosting and Infrastructure as a Service (IaaS) with scalability services.

This research focuses on cache optimized CIS. These CIS build in-memory state using data-sets of incoming problems. This in-memory state is built by performing necessary disk read operations during initial non-cached requests. For preceding requests if a cache hit take place then only execution is performed by reusing in-memory cache. Cache hits can reduce response time five to ten times or even more depending on type of CIS. Due to cache hits a service can handle more requests because it do not need to perform any disk-read operations. In-memory optimizations allows web applications
to use CIS more frequently and it also reduces infrastructure requirements. This in-memory cache optimization works good if there are small number of users web application have very few simultaneous requests i.e. 1 to 10. For a large scale enterprise web application these simultaneous requests can increase till 100s or 1000s. This makes it difficult to decide required infrastructure resources for cache optimized CIS. For example, if at any time number of non-cached requests increase and all CIS go in blocked state then many requests can go in waiting state.

An increase in waiting time can cross wait time tolerance of end-users and reduce system performance for end-user. If we assume maximum expected response latency as 1sec then wait time tolerance can be 3sec. Today, there are different opinions about wait time tolerance and it varies from couple of seconds till 30sec [8]. In spite of that users expect a predictable performance to get use to system and latency with less variance will get higher adoption.

These performance factors and computational nature of these enterprise web applications using CIS makes it hard to select appropriate infrastructure. It is difficult to find peaks of workload which also makes it hard to find optimum infrastructure to keep low cost and achieve high performance. This demands study and designing of an architecture with infrastructure services to support flexibility and low cost. Infrastructure should support scalability to add and remove resources according to workload [1]. In order to find optimized infrastructure of enterprise business applications using CIS there is a need to study impact of these performance factors [3].

In this research study and experiments have been performed to exploit performance factors of compute intensive services for infrastructure options. A SaaS based implementation has been presented using cloud IaaS to reduce impact of performance factors by achieving scalability. Next sections describes performance factors of CIS and research approach to design scalable infrastructure services for solution model of web applications and CIS.
1.1 Performance Factors of Compute Intensive Services for Enterprise Web Application

Large scale web application that use CIS can have performance impact due to uncertain request types and simultaneous requests. Each request from user is accepted in a separate thread of web application and available compute intensive service is used to get solutions for user requests as shown in figure 1.1. This separates request handling from computational execution and allows web application to accept all requests without blocking state. All computational execution is performed by CIS. This research considers CIS which are optimized to use in-memory cache.

In order to build and use in-memory cache, a compute intensive service verifies state identifier of data-set for each request. When a service receives a request of a specific state of a data-set for the first time then it performs disk read operations and it builds state in memory. In order to improve performance the compute intensive service keep this state in fast memory. This in-memory cache can be reused by CIS for preceding similar requests. If a cache hit is found for given identifier of data-set state then only execution step is performed and response is sent back to web application with low latency. If a service cannot find in-memory cache for given identifier of data-set state then service need to perform disk read operations and state building. A service can go in blocked state when it performs disk read operation and builds cache. Due to blocking state of services, web application may put next calls in waiting state if all services are found busy. Mostly this service blocking problem is solved by running multiple services and by using a load balancer that maintains list of available services.

For example; if there is a specific data-set $DS_A$ with state identifier 0101A then a user will get higher response latency if it is the first request sent to service S001 due to cache build action for state of data-set $DS_A$. All preceding requests for data-set $DS_A$ with state 0101A will get low response latency from service S001 because in-memory cache will be reused. If another user sends a request for the same data-set $DS_A$ and state identifier 0101A and request is routed to service S001 then in-memory cache will be reused and this user will also get response with low latency. Below is an example of request sequence:

- User-1 send request-1001 for $DS_A$ with state identifier 0101A. Service perform complete execution and send response back to User-1. $Latency = 1000ms$

- User-1 send request-1002 for $DS_A$ with state identifier 0101A. Service performs only execution by reusing cache and send response back to User-1. $Latency = 100ms$

- User-2 send request-2001 for $DS_A$ with state identifier 0101A. Service
performs execution by reusing cache and send response back to User-1.

*Latency* = 100ms

It is not guaranteed that a service will always have in-memory cache ready for given data-set and state identifier. It is also not guaranteed if request will be routed to a service which have in-memory cache ready. This variance of request types makes performance unpredictable. User calls that reuse cache will have low response latency and they can help to reduce or eliminate waiting time but if we get too many requests that require in-memory cache building for data state then response latency can keep on increasing due to waiting time. This performance factor of in-memory cache also has an impact on infrastructure cost because less number of services are required if most requests reuse cache.

Graph in figure 1.2 explains response latency difference that can happens due to increase and decrease of requests that require in-memory cache building. It shows impact of cache building at start of every day. For this example scenario we have 20 services running and incoming requests are *highest* = 800/sec between 10:00 – 14:00. In-memory cache is built during initial requests for data-set state. This in-memory cache is then reused for execution of preceding requests if service receive calls for similar data-set state. Running fixed number of servers is not typically enough due to performance factor of in-memory cache availability. In peak load situations due to cache availability if not enough services are there then waiting time will increase and it can become difficult to get predictable performance.

Another important performance factor is increase in number of simultaneous user requests. In enterprise and business applications users are from different regions of the world. These users have distinct time zones and working hours. For example in below scenario where users are from East Asia, US and Europe, we can observer simultaneous increase and decrease according to working hours and sales campaign.

- Peak hours for east Asia (UTC+9) = 23:00 till 09:00
During working hours system will receive higher number of requests and in other time slots we expect low number of requests. Each region does not need servers/resources 24hr. Overlapping working hours represent peak load, for example 08:00 – 09:00 and 14:00 – 18:00 when working hours for one region are still not finished but working hours of another region has been started. Different business scenarios also increases concurrent system access, for example a sales campaign or addition of more users to the system. Graph in figure 1.3 explains peak load for a specific region i.e. EU. It has an example scenario of a sales campaign in Europe between 09:00 till 13:00. It shows that we have peak load of 500 requests whereas average load during work hours is from 100 till 200 concurrent user requests. This increase of simultaneous requests in working hours can include cache building requests which can increase performance impact. If number of requests increase and all services are found busy then all next calls will go in waiting state and waiting time will keep on increasing.

Due to these performance factors response latency can increase and load peaks can appear. Organisations have a demand to handle these load peaks with low cost by running required services on sufficient number of servers. Unpredictable performance problem can become an obstacle for organisation to use features of compute intensive services with more freedom for regional and community users. For example, allowing open user registration or use system with more requests and speed during sales campaign.

Multi tenant SaaS providers that use compute intensive services can get even higher impact of performance and cost due to these performance factors. An increase or decrease of simultaneous user requests or in-memory cache availability can effect waiting time and server usage more rapidly. A big infrastructure requirement raises to support multiple organizations each with hundreds of concurrent user requests in peak load time. It is difficult for SaaS providers to tolerate big variance in performance because this will effect end-user experience and they may not prefer unpredictable latency.
1.2 Research Approach and Methods

Main focus of this research is to design and evaluate infrastructure services to reduce impact of performance factors related to CIS. The infrastructure should handle and reduce load peaks created by uncertain cache building requests and simultaneous user requests. Today, multiple infrastructure options are common for enterprise applications, this includes on-premise infrastructure, fixed cloud or hosted servers, IaaS, hybrid cloud and cloud SaaS infrastructures. This research work will begin with experiments of compute intensive services on dedicated servers to find cost and performance impact. Dedicated servers can be from on-premises or hybrid cloud and they are considered advantageous in many organizations for security and performance. Evaluation and experiments on dedicated servers will help to differentiate scenarios when such infrastructure is advantageous and when we get undesired performance or cost overheads. These experiments and evaluations will be based on single tenant or single organization aspect. In main part of this research a prototype SaaS architecture will be presented and evaluated to reduce impact of performance factors that are related to compute intensive services.

SaaS architecture will be built using cloud computing products and for simplicity all servers will be used from cloud computing provider. Cloud computing platforms are used because they have more out of the box products to support SaaS architectures. It has created an important shift in designing of IT infrastructure services. Cloud IaaS providers have introduced many different types of resources that can be acquired for shorter or longer period of time. Web application should plan required resources to provide good performance with low cost. Flexibility of resource usage and infrastructure services to manage on-demand resources has made cloud computing more popular than other options [4]. There is almost zero upfront cost and flexibility helps to achieve automation and auto-scaling [5].

AWS IaaS products have been used as underlying technology to build single tenant and multi tenant SaaS architecture by using motivating scenario defined in next section. Many standard input metrics are available in AWS for scaling alarms and scaling policies e.g. CPU utilization [6]. However these standard metrics are not enough to achieve performance for CIS. Cache optimized compute intensive services used by enterprise web application have special nature due to performance factors such as, in-memory cache and variance in simultaneous user requests. There is a need to study these performance factors and identify bottleneck components in system architecture. Cloud IaaS products can be customized by using supplementary methods to achieve scalability and performance [5]. These supplementary methods will be investigated in this research for CIS over cloud IaaS.

Proposed SaaS architecture will be evaluated for different data-set state sizes and multi tenancy. There are possibilities of incompatible scenarios
that may not achieve desired performance in proposed architecture. Additional application specific or architectural methodologies will be proposed for these incompatible scenarios to achieve balanced performance and cost in SaaS. There are many technical challenges involved in SaaS development. One of them is multi-tenancy, which allows single instance of software to serve multiple organizations by accommodating their unique requirements through configuration at the same time [7]. The proposed solution will be evaluated for multi tenant SaaS solution considerations and advantages.

1.3 Motivating Scenario

In this research we have considered a web application that has supplementary open user access on enterprise level. This web application uses compute intensive services to handle requests. Each request contains a dataSetName, stateId and dataValues of the problem to get a solution. To illustrate actual workload pattern of simultaneous requests for compute intensive services we need to find request frequency. If we consider that each active user makes request every 5th second with frequency (F) = 0.2/sec:

- Registered Users (U) = 10000
- Active users in peak working hours (P) = 2500
- Peak requests load ‘P-Load’ = (P) * (F) = 2500 * 0.2 = 500 requests/sec

Latency of response depends on data-set prepare time and in-memory cache state. For simplicity of this research below cases have been considered. This latency can be much higher in many other scenarios.

- Response latency when no cache hit happens and cache has to be built = 1000ms
- Response latency when there is a cache hit and only execution step is performed = 100ms

If cache hit happens then latency to deliver response has been used as 100ms, this can be called as cached request. If there is no-cache and application service has to build and deliver the response then it can take 1000ms, this can be called as non-cached request. These are recommended response latencies in web application used in this research but there can be several cases with higher response latencies. Experiment and evaluation will also be performed for higher latency services in this research.

Performance of a web application has been evaluated by comparing response time with wait time tolerance. In this research wait time tolerance has been considered as 3sec. First reason to choose 3sec threshold is that web users typically remain focused for 3secs for each request however it can
vary for different systems and it can be higher then 3sec. Second reason is that highest response latency of a single request in this motivating scenario is 1000ms so threshold latency has been taken three times higher then highest possible latency.

1.4 Boundaries and Assumptions

Main goal of this research is to evaluate and present infrastructure services for a SaaS architecture. This research does not consider internal structure of compute intensive services however if required, proposals will be made for CIS. Compute intensive service considered in this research loads data-set state from disk and it prepares problem state of a specific state in fast memory one time only. Only execution step is performed for incoming requests if cache hit happens and state identifier matches. Data-sets can handle many different problems related to same date. For example for 500 requests:

- DataSet 1 can handle problem request 1 - 100
- DataSet 2 can handle problem request 101 - 300
- DataSet 3 can handle problem request 301 - 500

In-memory cache is maintained by each service for only those states of data-sets that has been received by the service. A service may have to build cache if it has not received a specific state identifier of a data-set before. CIS may also restart to empty its in-memory cache if it do not have more memory available for incoming requests with unique state identifiers of data-sets.
Chapter 2

Background

Web applications have became a mandatory part of enterprise success since last few decades. Enterprise data and services are used around the clock to provide open access for customers and increase sales. Public access to web application and services like compute intensive services demands high infrastructure resources. Challenges like infrastructure resource usage and cost blocks extensive usage of web application and it can obstruct enterprises to take full advantage of these services [1]. An enterprise or business web application need to support hundreds of concurrent requests from thousands of registered users. This include direct enterprise users as well as API calls to share real-time information and processing [9]. These requests are received by a central web server that runs web application, database and compute services. This Web server can accept requests by allocating separate thread to each request but it can become bottleneck if each request also need some computational and memory resources.

These bottlenecks are typically removed by separating CIS from web application and by using a load balancer where multiple compute intensive services run on separate servers. With addition to this separation, low response latency is achieved by optimizing cache usage of CIS. In-memory cache is built and reused by services which helps to handle more requests by same number of services. Next sections of this chapter will explain load trends, load balancing and in-memory cache details. In-memory cache building and its reuse is part of service inter-logic and these sections will only explain benefits and limitations of in-memory cache usage.

2.1 Load Trends of Enterprise Web Application

In common enterprise architectures, enterprise users are connected to single web application and common database. Thousands of enterprise users become registered to the web application and during working hours they can make hundreds of simultaneous requests. Due to infrastructure demands
of CIS, usually enterprise users have restricted access to compute intensive services. This stops enterprise to take full advantage of compute intensive features. Organisations can acquire more intelligence and more users can perform intelligent operations if they can get access to CIS with a simplified or guided layer. For CIS providers it is also important to support Application Programming Interface (API) requests where API requests can increase and decrease execution load at various times [9].

In order to solve these infrastructure and scalability problems different load balancing and in-memory cache optimizations are used. Many CIS need to run together to handle simultaneous requests. A basic load balancer can be used inside or with web application to maintain list of available services and use them to share load. This basic load balancing can remove bottleneck from web application but it is not sufficient to achieve performance.

2.2 Load Balancing For Compute Services

In enterprise web application, hundreds of simultaneous users requests can occur. In large and global enterprise web application this number can raise till thousands of simultaneous requests. In order to handle these many concurrent requests large number of CIS need to run together. A basic CIS load balancer can be created and used inside or with web application. This load balancer maintains list of available services and it use them to distribute load. This basic load balancing can remove bottleneck from web application. Multiple compute intensive services can run on a separate server with their own computational and memory resources. These services are used by web application for incoming requests to share load. System architecture with basic load balancing and multiple services may work fine for limited number of simultaneous user requests. Large number of simultaneous user requests can have significant variance due to working hours and business campaigns that can effect performance of web applications. There are two possible solutions to solve this scalability problem, scale-up and scale-out [5].

- scale-up, add more processing and memory resources to servers
- scale-out, add more servers for compute intensive services

Adding more memory and processing power to a single instance has limitations and initial cost can increase enormously to support large system. It also require more precise analysis of required infrastructure which is not always possible. Scale-out or running multiple servers is more common architecture because it is easier to buy hosted or cloud servers and it is also a fail-safe approach. Figure 2.1 explains this architecture. This architecture of multiple servers with multiple compute intensive services can support large number of concurrent requests. More servers can be added to system to support more concurrent users if needed. Load balancing solution can help
Figure 2.1: servers of web application and compute intensive service

to ensure that each request will eventually be handled but response latency will be very high if web application and CIS handle each request as unique request.

2.3 In-Memory Cache Usage and Benefits

Usage of in-memory cache can make a huge impact in enterprise applications. Application can be designed and developed in completely different way and in-memory cache can help to perform faster computation and analytic. [10]. For CIS, they can prepare response and keep it available using in-memory cache. This cache can be reused for all preceding similar requests and they can get response with low latency. This available cache can actually help CIS to handle upcoming similar requests with low response latency.

In-memory cache approach enables very fast response time because instead of hard disk the state is loaded and used from RAM. Trend of using in-memory cache is rapidly increasing because enterprises see big advantage in using in-memory cache. Many enterprise vendors that are using in-memory cache claims that performance and response latency has been improved 10 to 100 times than traditional on-disk methods [11]. Continuous reduction cost of RAM also makes it easy to use more in-memory cache with time, for example; the per megabyte cost of RAM has dropped from $10,000 in 1980 to $1 in 2000 and it has been further reduced to only 0.4 cents today [11].

A performance optimized compute intensive service executes request in three steps;

- First step is to load state data from disk.
- Second step is to prepare problem or data state in memory using data variables.
- In third step actual computation is performed using data variable values and solution is sent back to web application.
Figure 2.2 explains a scenario where Request2 gets lowest latency due to execution only step. Request1 and Request3 will get high latency due to disk read operations and cache build. Request1 and Request3 will also make application service busy and this will block next incoming request i.e. Request4.

In-memory cache can help to handle compute intensive request with low response latency. Cache reuse of in-memory cache can optimize a service to build solution or response with very low response latency and a service can handle more similar requests. However there are few limitations in usage of in-memory cache which can make it difficult for organisation to prefer in-memory cache based solution for enterprise applications. Large data-sets cannot be loaded in RAM. It is also difficult to increase RAM while server is running and RAM increase can introduce down-time. And third limitation is that RAM still costs more than disk space.[11]. While running multiple services it is difficult to guarantee if the target service already have in-memory cache or not. In these cases a specific service instance can become busy in building cache and it cannot take any more request. If multiple
service instances are running then request will be routed to another service instance but it can also become busy in building cache if it does not already have in-memory cache for state of given data-set.

These limitations can create big performance impact for enterprise web applications. This architecture of in-memory cache and load balancing with multiple servers can help to ensure that each request will eventually be handled but response latency can vary for requests and it can become difficult to get predictable performance. In next sections performance impact of in-memory cache and increasing concurrent requests has been explained for this architecture.

2.4 Performance Impact of In-Memory Cache

Performance impact due to in-memory can become important if variance of incoming requests is bigger. Too many requests requiring cache building can make compute intensive services blocking and increase waiting time. Similarly too many requests reusing in-memory cache need less number of compute intensive services and can improve performance. There are three main cases when a service has to build cache for incoming request;

- No Cache; no cache hit found for state of given data-set in service cache table. This can happen if it is the first service that has received request for given data-set or if service that previously has received the request is busy and request has been routed to a new service.

- Invalid Cache; in-memory cache is invalid and new state of data-set has to be loaded from disk. A new state identifier of data-set can indicate that data is invalid.

- Cache Refresh; cache has been removed due to a cache refresh. This is mostly required to remove unused cache. A service restart can remove all existing cache entries to allow new cache entries to be loaded.

Request$_3$ in figure 2.2 shows the case of no-cache due to busy service. Common web applications do not have cache information about each service. If a service is found busy then request will be routed to next available service. This can create an increasing performance impact if many requests for same data-set are received because each request may will get its own service and each service may perform cache build. This process has been explained in figure 2.3. Each time the service will build cache it will become busy and this will effect overall waiting time. Too many occurrences of cache building can make system performance unpredictable at that specific time. It is a good approach to use cache whenever possible but for large scale web applications it demands to handle cache building impact on performance.
Figure 2.3: in-memory cache latency
2.5 Performance Impact of Simultaneous Requests

Large scale enterprise and web applications have hundred or thousands of registered users. These users can belong to different regions, sub organizations and partners. Registered users do not reflect actual system load. There can be thousands of registered users but simultaneous user requests which actually use system depends on their working hours. If a web application has registered users from Asia, EU and US then due to difference in their working hours load on compute intensive services will increase during working hours and it will decrease during off time. Figure 2.4 shows increasing and decreasing concurrent requests where each region has its own infrastructure.

Running servers for each region can reduce utilization and servers can become under used because these servers will have very low usage in off-work hours. This factor of working hours makes it difficult for organizations to decide how many servers they should keep for each region. If web application allows API access and requests then this problem becomes even bigger. For example, service providers of CIS who provide API access or web services for integration, to offer their services for organisations running other enterprise applications [12]. These integration calls can vary in size and time therefore it is difficult to predict and find required capacity of servers. This performance factor of increasing requests is even more important for multi-tenant service providers. In multi tenant environment each organisation can increase and decrease simultaneous requests load and this can effect overall system performance.

This often require a study of expected simultaneous requests, request and data-set types, wait-time tolerance. Performance evaluation for increasing and decreasing simultaneous requests is also required to find system capacity to handle demand growth and potential performance degrade [13]. Additional application specific cache optimizations are often considered as a solution to achieve performance but we have seen that in-memory cache have limitations and it has performance impact for increasing simultane-
ous requests. Development of infrastructure services is becoming a common solution to handle scalability and resource requirements. Infrastructure services are required to address special characteristics of CIS and provide a flexible system with low cost but high performance.

2.6 Infrastructure Choices for Compute Intensive

Compute intensive services need high processing and memory resources if they need to handle hundred or thousands of simultaneous requests coming through web application. Today, there are multiple infrastructure options such as on premises dedicated servers, cloud or hosted dedicated servers and cloud computing with IaaS.

On premise dedicated servers are considered as first option in many organisations because this approach have one-time cost and low maintenance is expected. In this architecture resources are bought and placed on-premises and IT administrators take care of this infrastructure to maintain upgrades, backups and replacements. These servers are maintained by using on-premises tools. These tools are also installed and maintained by IT administrators. This architecture is most suitable if all users are in same building or use VPN. It is also considered as a secure and more long-term approach. For web applications that are built for large number of users or multiple organisations these advantages may lose their actual significance. This approach has a big on-front cost. Maintenance cost can increase and it can become difficult to maintain this architecture for a web application where we have hundreds and thousands of users. Maintenance cost of on-premise can also change if an upgrade is planned due to software upgrades or hardware upgrades to support more users. It is also difficult to reduce downtime during system upgrades.

A relatively better and more common approach is hosted or cloud servers. These servers can be added and removed at any time and they usually have fixed cost per month which include maintenance cost of actual hardware. In this approach initial cost can be reduced and servers can be purchased on monthly basis or for a specific time. Some maintenance cost will still be there but it is much reduced because IT administrator can use on-line tools to maintain servers, perform backups, add or remove servers. However these actions are manually performed and they are mostly based on predictions. Due to manual nature of these actions it is also difficult to perform them often and before required time.

Cloud computing has been emerged as services oriented way of using IT infrastructure resources [4]. Enterprises are shifting towards cloud computing and it is reshaping IT industry [15]. Users can access cloud resources via services offered by cloud computing providers and they can achieve benefits like cost saving, high availability and easy scalability [15]. Cloud computing
providers offer virtual resources as a service by hiding infrastructure details [16]. Elasticity has became one of the most important advantage in cloud computing. Cloud computing providers link elasticity to different types of resources which includes server instances, storage, RAM, CPU resources and network capacity [1]. In next sub-sections different cloud computing services have been discussed with their advantages and challenges for CIS.

2.6.1 Cloud Computing and IaaS Providers

Cloud computing providers offer a wide variety of cloud services to make it easy for different types of enterprises to use cloud resources. In public cloud all cloud resources are managed by cloud provider. It depends on cloud provider how they manage different customers but usually cloud resources are shared among customers with privileges to reserve and release sources according to usage requirements [15]. Private clouds are built completely on private networks and a single customer is the owner. Private clouds can be managed by a cloud provider or IT department of customer but is becoming more cost effective to use services of cloud provider [15]. Hybrid cloud mostly have on premises resources and cloud resources are acquired to fulfil on demand elasticity requirements. Hybrid clouds are sometimes more complex to manage but cloud computing providers are also focusing more to provide services for hybrid cloud [15].

Cloud computing is a collection of services and mostly these services are divided in three different types IaaS, Platform as a Service (PaaS) and SaaS. Cloud services that allow to manage cloud resources are mostly considered as IaaS. PaaS creates an extra layer of IaaS with some specific software or application [15]. For providers of CIS, SaaS is an interesting area because in SaaS whole application runs on cloud. End-user access is performed by a thin client like browser and API access is allowed for integration application.

There are several challenges in adopting cloud computing, such as security, performance, availability, integration with in-house infrastructure where security and performance being most significant [17]. Security challenges are mostly part of organizational processes but performance challenges need to be studied. Due to special characteristics of web applications that use compute intensive services there is a need of flexible yet inexpensive infrastructure services. A scalable infrastructure can help to handle performance impacts of in-memory cache and increase number of concurrent user requests.

Today, there are many cloud computing vendors that provides flexibility and out of the box infrastructure services such as scalability. AWS is one of leading cloud computing vendor and AWS services like auto scaling can be used to add and remove servers when needed. In AWS standard features, an auto scaling group can be created and on-demand servers can be added and removed by scaling policies which can be triggered by using
CloudWatch alarms. These CloudWatch alarms can be configured to trigger scaling policies according to system load and resource demand. Many standard metrics are available for CloudWatch alarms and scaling policies e.g. CPU utilization [6]. One of the main advantage of cloud computing is scalable architecture. Therefore, it is very important that application architecture have components for scalability. If components are not easily scalable then usage of cloud computing services like AWS auto scaling will not help to their full potential. A scalable application supports resource increments and it has heterogeneous components [5].

Private cloud, hybrid or public cloud infrastructure services can be selected according to requirements but web applications need to provide SaaS based architecture and solution to become enterprise ready. Enterprises and organisations need standard approach of infrastructure features and services to get predictable performance, fixed cost and smooth upgrades. Cloud IaaS providers such as AWS has many useful products to build a SaaS solution.

2.6.2 SaaS Solution Challenges for Compute Intensive

Performance evaluation and scalability is one of the most important challenge for applications that run on cloud computing [18]. Cloud computing providers claim elasticity and scalability as one of major advantage but it is difficult to validate this claim for different type of applications including web applications using CIS [18]. SaaS applications running on cloud computing are becoming popular and providers of CIS have an opportunity to solve their limitations with a SaaS based offering. CIS providers can sell their services with SaaS model and Service Level Agreement (SLA) to make a more standard solution offering [18]. Performance indicators like large number of simultaneous user requests and in-memory cache have big impact on performance and due to these factors it can become hard to define a common SLA for SaaS users. A thorough study of cloud IaaS features for SaaS and designing of multi tenant SaaS solution is required for compute intensive services.

Predictable performance of compute intensive services require scalability in order to easily add and remove servers when needed. Scale-out approach is considered a better choice because it does not rely on high performance computational equipment. It can save big on-front investments which are required in scale-up approach. Adding more servers horizontally helps to scale in distributed way and therefore it does not need huge up-front cost. This approach still require demand predictions on regular intervals. Cloud IaaS features help to automate this instead of manual monitoring. Scale-out approach can have several challenges for web application that use CIS which can handle more requests for cache hits but they can become blocked due to cache building requests. From performance aspect such requests can increase waiting time on overall system and response latency can increase...
The graph in figure 2.5 explains both approaches. Scale-up approach has big performance risks because after detection of load there is a gap when resources actually scale-up. Scale-out approach helps to achieve performance much earlier and faster than scale-up. New resources can be added to the system with an increase in simultaneous requests and resources can be removed when not necessary, to save cost.

Scale-out approach helps to achieve performance by adding resources to the system earlier. Due to separation of web application and CIS it is a better approach because new servers with CIS can be added on-demand. Scale-out approach require scalability data and it is difficult to find scalability data due to in-memory optimizations of CIS. The increase and decrease in number of requests cannot represent scalability data correctly. CIS do not have uniform resource demand with increase and decrease of simultaneous requests. The graph in figure 2.6 represents an example of resource demand for CIS. When system has cache building requests, for example at start of day then more resources are required but when CIS have cache ready and there are more cache hits then less resources are enough. For CIS instead of number of requests the type of requests and cache state is more important.

Cloud computing providers are still developing their products therefore these challenges need to be studied and solved [1]. Due to these challenges many enterprise web applications prefer scale-up approach to achieve performance by paying high infrastructure cost. There is a need to study and design a SaaS architecture for CIS to manage challenges of scale-out approach. A study is required to identify cloud products and design additional components.
2.6.3 Multi Tenant SaaS Solution Benefits

In multi tenant architecture a single infrastructure is shared by multiple organisations. Multi tenant architecture is becoming popular for SaaS based products because it removes installation and infrastructure complexities and it helps to save up-front costs. Organisations can start using application quickly and they can evaluate system in more effective way. Rapid evaluation helps decision making and due to multi tenant solution they get easy upgrades and immediate access to standard features. Figure 2.7 explains multi tenant architecture where multiple organisations can connect to a single web server to share resources and features.

Multi tenancy has benefits for both customer enterprise and service provider. Service providers can manage customers with predictable maintenance cost because single point of service is provided and infrastructure resources are shared between organisations. Multi tenant architecture for CIS have similar challenges like single-tenant but there are also few advantageous scenarios. Because organisations can share resources so they can
borrow and release shared resources on demand. This can make scaling faster but it is difficult to identify when organisations should borrow resources and when they should scale-out. For CIS there is a need to design a methodology for multi-tenancy to make sure that it do not effect performance of organisations and it help to achieve performance with faster scalability.

Cloud computing vendors like AWS provide logical modules to distribute resources and servers among organisations. For example AWS: Auto Scaling Group (ASG), an ASG contains its own servers and it can add and remove them according to need. ASG can be created for each tier of organization that require auto scaling. Organisations can also share ASG but for CIS a study and proposal is required to avoid performance impact.
Chapter 3

Experiments and Analysis of Dedicated Servers

Dedicated server approach are one of main infrastructure choice for enterprises to run web applications for enterprise users. Dedicated servers running on-premises are consider one of best solution to achieve performance and keep enterprise data in secure architecture. These servers can be bought and run on premises but this require big on-front investment. Not all organisations can prefer this architecture. Hosted or cloud servers are better alternative where user pay per month or for a specific time-period. This approach includes maintenance cost and manual actions to perform upgrades and scalability.

In this chapter motivational scenario 1.3 has been used with experimental load patterns to find different values of required services and servers. These experiments have been performed to identify required number of CIS and servers for a specific load and time. These results help to identify cost and performance impact of dedicated servers. Below assumption have been used in these experiments;

- We can run 5 services on each server
- Server execution capacity per second (SC) = milliseconds per second * services = 1000 * 5 = 5000
- Application service can serve 10 cached dataset requests per second
- Application service can serve 1 non-cached dataset requests/second

Organisations that have global users from different regions or partners can run single infrastructure of dedicated servers for all regions or they can allocate separate servers to each region. In next sections experiment details has been described for both approaches to find performance and cost impact. These experiments can help to identify limitations of dedicated server architecture for CIS.
3.1 Study of Region Specific Servers

In region specific architecture, each region has its own set of servers. Required number of servers are maintained according to load patterns of simultaneous requests. In order to calculate number of services required for simultaneous requests of a specific region like European Union (EU) load patterns have to be identified. In this research load patterns have been assumed as below. Graph in figure 3.1 shows these load pattern for a twenty four hours cycle.

- peak load or maximum simultaneous requests = 585
- average number of simultaneous requests = 160
- median of simultaneous requests = 23

**Experimental Calculations:** to find infrastructure requirements of a specific region using above load pattern and motivating scenario defined in 1.3;

- Total computational time of cached requests \( \text{Cached}_R \)
  \[ \text{Total compute-time} = (\text{compute-time per request in ms}) \times (\text{requests per second}) \]
  - Max = 100 * 585 = 58500ms
  - Avg = 100 * 160 = 16000ms
  - Median = 100 * 23 = 2300ms

- Servers needed normal cached request \( \text{Cached}_S \)
  - Max = CT-Normal/SC = 58500/5000 = 12server
  - Avg = CT-Normal/SC = 16000/5000 = 4server
  - Median = CT-Normal/SC = 2300/5000 = 1server
Total computational time of non-cached requests $Peak_R$

- Peak
  \[
  \text{Max} = 1000 \times 585 = 585000 \text{ms}
  \]
  \[
  \text{Avg} = 1000 \times 160 = 160000 \text{ms}
  \]
  \[
  \text{Median} = 1000 \times 23 = 23000 \text{ms}
  \]

- Servers needed for non-cached requests $Peaks$
  \[
  \text{Max} = \text{CT-Peak/SC} = \frac{585000}{5000} = 117 \text{server}
  \]
  \[
  \text{Average} = \text{CT-Peak/SC} = \frac{160000}{5000} = 32 \text{server}
  \]
  \[
  \text{Median} = \text{CT-Peak/SC} = \frac{23000}{5000} = 5 \text{server}
  \]

Above calculations show big variance in server requirements. Best case for average and cached requests is 32 servers where as worst case is 117 servers. But worst case may never happen because cache will be reused and not all requests will trigger cache building. The difficult part is finding actual requirement of more then 32 servers.

Graph in figure 3.2 shows load of a specific region i.e. EU. This load can go higher each day but required number of services is much lower then number of users i.e. max 184 services for 585 user requests. Once, in-memory cache becomes ready then only 58 services are required for same number of user requests. The average is 32 and due to non-working hours the median is only 5 services. Most of resources are idle for $1/3rd$ of the time i.e. non-working hours 17 : 00 till 08 : 00. In case of hosted servers this is the cost paid for no use. It can be observed from graph that server requirement is mostly under 100 services if there is a cache ready system. Graph in figure 3.2 also shows that cache building demand and increase of simultaneous request has big impact on number of required services. The graph shows faster increase in resource demand when cache-build increases. The architecture has big variance for required servers due to cache reuse.
When each server has cache ready then low number of servers are enough. If we get a data update and all cache has to be rebuilt then we again need higher number of services.

3.2 Study of Single Infrastructure Servers

A single infrastructure can be used for all regions to increase server utilizations. Work timings of each region differ from other if an enterprise have users from regions like Asia, EU and America then servers will mostly remain busy and idle time will be reduced. In this research load patterns have been assumed as below. Graph in figure 3.4 shows load pattern for a twenty four hours cycle.

- peak load or maximum simultaneous requests = 1040
- average number of simultaneous requests = 538
- median of simultaneous requests = 524

**Experimental Calculations:** to identify required services for single infrastructure using above load pattern and motivating scenario defined in

Total computational time for non-cached requests $Peak_R$

\[ Peak_R = (\text{compute-time per request in ms}) \times (\text{requests per second}) \]

Max = 1000 \times 1040 = 1040000ms

Avg = 1000 \times 538 = 538000ms

Median = 1000 \times 524 = 524000ms

Servers needed non-cached request $Peak_S$

Max = CT-Peak/SC = 1040000/5000 = 208 server
Above calculations show that variance is still huge between $Cached_S$ and $Peak_S$. In single infrastructure because idle time reduces so variance between average and median also reduces. Best case is 11 servers but worst case is 208 servers for $S - Peak$ which is even higher than region based servers. If it is considered that worst-case will never happen because of cache reuse then still it is difficult to find required number of servers between 11 and 208.

Graph in figure 3.4 shows that in single infrastructure there can be several peaks higher than region specific servers during overlapped working hours. For example, maximum or highest load peak is 1040 user requests and it require 281 services, but due to single infrastructure for all regions there are multiple average peaks with average peak load of 600 requests. These peaks require 90 to 115 services. Average peak load gives a better view of required services but during highest peak load system performance will be effected and response latency can increase. It can be observed from graph that a cache-ready system require much smaller number of services and at start of working-hours each region demands more services due to cache building requests. The difference between max and average is still high but the difference between average and median has been reduced due to reduction of idle time. It is still difficult to decide between 100 and 200 services and achieve performance in single infrastructure for CIS.
3.3 Evaluation and Limitations

It is hard to identify required number of servers for cache optimized CIS. User feedback or performance log analysis can be used to find required number of servers. This can help to identify if more servers are required or not. However, this approach cannot ensure runtime performance and servers can only be added after performance impact. This approach also does not give full information about server requirement. If unique or non-cached requests are received then more services are required but once cache gets built then less number of services are enough. On premise infrastructure can give better performance but it can have high cost due to hardware, infrastructure services and maintenance. During peak load each non-cached request demand one service per user request.

Experiments have shown that region based and single infrastructure, both have big variance for required servers between peak load and average load. Dedicated servers can require two times more servers to handle peak load then average servers. Region based servers have idle time in non-working hours and load peaks can appear during working hours. However, as compare to single infrastructure for all regions, these load peaks are smaller and therefore they will create less performance impact. In single infrastructure, very high load peaks can occur and they can degrade performance for all users of all regions. Single infrastructure can remove idle time and hence it can give low cost but performance can become worse during collective load peaks.

If number of users is very low and state of data-sets doesn’t change often then dedicated servers can give desired performance with predictable cost. But, an enterprise business can lose competitive edge if web application allows only limited access to CIS services or if data-set states remain static for longer time periods.

3.3.1 Limitations for DataSet Update Policies

Data-set state contains latest information about solutions. In enterprise business processes it has been considered that these states will remain static for longer periods of time. A static state can improve cache reuse but this may not represent correct business data. Allowing more frequent state update also open new possibilities for business development and sales. Data-set state update policies can be spontaneous or time based like every morning, weekends, or once a month.

Each state update of data-set can introduce new cache building requests. If dedicated servers are not enough in number then performance will degrade after every state update. If state of data-set does not have frequent update and there are fixed small number of simultaneous user requests then dedicated servers can provide predictable performance with low cost. However
this stops organisations to scale and therefore partners and enterprise users cannot get freedom to use system with latest data-set states which can include inventory, pricing and sales information.

### 3.3.2 Limitations for Additional Users and API Requests

Regional and partner offices have their own set of users that can use web application. In dedicated server approach, it is difficult to allow additional users without proper planning. Introduction of new users can increase simultaneous requests and existing system capacity will not be enough to support. Usually organisations prohibit bulk user registration to avoid performance degradation but delay in new users can reduce potential use of CIS.

Additional users and variance in simultaneous requests is an even bigger problem for service providers of CIS. They commonly have to find expected resource requirements for their customers. If providers of CIS recommend too many dedicated servers then not all enterprises will accept it due to high cost. If they recommend less dedicated servers then enterprise users may get unpredictable performance and they may complain about it. Enterprise specific solutions can be created for each complain case but providers of CIS still need a more general approach to add and remove servers when required to achieve performance with low cost.
Chapter 4

SaaS Architecture Design
And Performance Analysis

Solution model of web application with CIS requires a standard infrastructure offering for enterprise customers. Large enterprises are more interested in cloud based architecture that is flexible and elastic to save upfront cost and pay according to usage [14]. In this chapter a prototype SaaS architecture has been presented for single tenant and multi-tenant organisation. This architecture provides a general solution and standard offering. SaaS offering can help organisations to get standard architecture, immediate access and they can pay according to usage [14].

In next sections of this chapter cloud IaaS products has been identified to build this prototype SaaS architecture. Due to special characteristics of CIS these cloud products cannot be used directly. Later in this chapter, supplementary components has been identified and developed to fill shortcomings of these cloud products.

4.1 Applicable Cloud IaaS Products

Cloud providers offer infrastructure services over cloud resources to manage the architecture. These services help to deploy web based solution with more flexibility and minimal administration cost [5]. Today, many of these services are free to use and they can become building blocks for a SaaS solution. Below list shows few of AWS products that has been used in this research to build the SaaS architecture.

- AWS Elastic Compute Cloud (EC2), virtual servers in the cloud
- AWS ASG, container for servers with dynamic elasticity
- AWS Elastic Load Balancing (ELB), high scale load balancing
- AWS CloudWatch, monitor resources and applications
Figure 4.1: ASG Scaling

- AWS Launch Configuration (LC), used to add new server to ASG
- AWS Lambda, run code in response to events
- AWS Cloud Formation (CF), create and manage resource templates
- AWS S3, storage service with objects and buckets

One of the most fundamental cloud computing product for prototype SaaS architecture of this research is AWS ASG. An ASG provides elasticity and it organises a group of servers or instance. Each ASG has properties that represent capacity of auto-scaling group in different states. For example figure 4.1 explains three distinct attributes of ASG that can be used to provide elasticity [20]. ASG can scale up till maximum size and it will maintain minimum one instance. Interesting attribute is desired capacity. Updating desired capacity can trigger auto-scaling action to add or remove instances [6]. An ASG can have manual, scheduled or dynamic scaling. Manual scaling means changing desired attribute of an ASG directly whereas in dynamic scaling desired attribute is controlled by scaling policies [6]. These auto-scaling polices can be triggered from CloudWatch alarms [23].

When multiple ASG servers run together then there is also a need to ensure that traffic is distributed evenly across all instances. AWS ELB provides a layer above auto-scaling group to monitor traffic and perform load balancing by distributing load across all instances [24]. Auto-scaling groups can be attached to AWS ELB. AWS ELB can send data about load balancers and EC2 instances to AWS CloudWatch for monitoring purposes [24]. ELB saves data in different metrics of CloudWatch.

In AWS CloudWatch different alarm can be created by using metrics data. CloudWatch alarms take input data metric and they can trigger different types of action [22]. A CloudWatch alarm can have below actions [23]:
• Notification

• AutoScaling Action: trigger auto scaling policy of an ASG

• EC2 Action: stop or reboot specific instance

ASG scaling policies can be triggered by CloudWatch alarms with specific alarm condition [23]. Each alarm can trigger actions, for example whenever a specific metric value goes above a specific threshold then trigger scale-up to change ASG desired attribute. AWS provides standard metrics to support dynamic scaling based on metrics data uploaded by ELB [23]. This metrics data includes many interesting metrics like ELB request count, EC2 CPU usage and many more. In dynamic scaling of ASG the CloudWatch metrics data can also be loaded from customized or supplementary sources [22]. It is recommended to create at least two scaling policies to facilitate scale-up and scale-down events [20].

4.2 Basic SaaS Architecture

A basic SaaS architecture in this research has been built by using AWS cloud IaaS products and web application. The web application runs on its own server and it can connect to multiple compute intensive services [19]. Figure 4.2 represents a SaaS architecture built using AWS infrastructure products or services mentioned in 4.1.

SaaS architecture presented in figure 4.2 is a basic architecture and it is difficult to perform scalability of compute intensive services due to performance factors of compute intensive services mentioned in 1.1. CPU time and request count metrics in AWS CloudWatch can help for web application scalability but they do not have sufficient information for scaling of compute intensive services. A compute intensive service can handle higher number of requests if it has cache ready and it can also go in blocked state if it needs to build cache for a request. AWS provide dynamic scaling features which can be used in such cases. Supplementary components are required to collect informations about CIS and load it in AWS CloudWatch custom metric. Next section discusses details about these components and methods to collect performance related data of CIS.

4.3 Supplementary Features and Components

Dynamic scaling in AWS ASG require scalability data [20]. If continuous scalability data input is available then pre-built cloud products can be used to scale-out and scale-down. In this research scalability data of CIS consists of simultaneous requests and cache building requests. Each service does not know about other services so there are two possible approaches to collect
Figure 4.2: saas basic architecture
scalability data. In first approach, specific data about cache hit can be loaded from each CIS whereas in second approach web application can track overall cache hit details and simultaneous requests. In this research we have used second approach because we had a possibility to identify impact of cache-hits and simultaneous requests as waiting-time in web application.

Web application receives user request and it forwards it to available compute services. If a service is not available due to previous cache building request then web application will try to use another service. If all services become blocked for longer period then request waiting time will increase for incoming requests. Similarly, this waiting time will also increase if too many simultaneous request are received. This waiting time can be aggregated by web application as queue time. This queue-time will increases and decreases according to cache hits and simultaneous requests. This queue-time can represent performance impact and it can be used to perform scalability. Next sub sections explain supplementary methods and components to collect and use queue-time for scalability. These supplementary components are required for SaaS architecture.

4.3.1 Queue-Time Input For Metric Data

ASG of CIS need scaling if web application cannot find available compute service and queue is building up. Web application can calculate queue-time of requests that are waiting for a compute service and this queue-time can be used as a CloudWatch metric data. If this queue-time remains for longer time then scalability requirement can be identified and more resources can be added to the system.

CIS have multiple servers queuing model such as M/M/C where arrivals come from single queue [25]. Each request has a response time and waiting time. Response time is the total time a request has spent in queue and in service [25]. It is important to consider that Service rate can have big variance in CI-services due to variance of non-cached requests. A simple queue-time calculation can be performed by web application by counting number of requests and total time spent by active requests in system.

Example of queue-time calculation; let’s consider that at a specific time web application has 15% non-cached requests, system has 10 concurrent requests and three CIS are running. If three requests are active with total time as 2911 then we can divide this total time by not started requests count to get queue-time as 415.

- Waiting requests count \( R_q = 7 \)
- Active Request \( R_1 \) total time = 889
- Active Request \( R_2 \) total time = 767
- Active Request \( R_3 \) total time = 1255
• Queue-Time $W_q = (R_1 + R_2 + R_3)/7 = 415$

This is just one way of calculating queue-time and there can be several other ways. Calculation of queue-time is part of web application and this research assumes that web application calculates queue-time that can be retrieved on regular intervals. If queue-time $W_q$ and waiting requests $R_q$ remain like this for multiple periods then queue-time will start raising rapidly as shown in 4.3. A continuous increase in queue-time reduce performance and response latency can become too high.

If queue-time $W_q$ is greater then threshold response latency and waiting requests $R_q$ remain like this for multiple periods then scaling requirement can be identified by using multiple occurrences of $W_q$. This can help to make a scale-up policy. For example: if cumulative queue-time increase more then 3sec for 3mins then add more compute services. Queue-time can represent load of simultaneous requests and it gives more precise performance data for CloudWatch metric. If queue-time is zero for longer time then scale-down can be performed.

4.3.2 Scaling Service For Web Application

Web application performs queue-time calculations which helps to get more precise performance data. This queue-time has to be updated in AWS CloudWatch metrics on regular intervals. Dynamic scaling can provide scalability based on AWS CloudWatch metric data [20]. A scaling service or application is required to run beside web application to read queue-time from web application and update AWS CloudWatch metric. Scaling application need to provide inputs for AWS CloudWatch metric as well as for web application. For AWS CloudWatch metric input, it will collect queue-time from web application on regular intervals and it will update metric data in AWS. This can be done by using API services of AWS. For web application
it will collect active and healthy server from AWS and it will update web application on regular intervals. A regular interval can be every minute. Diagram in figure 4.4 represents both actions.

In step 1.1 and 1.2 queue-time is collected from web application on regular intervals and queue-time metrics is updated. This queue-time metric data further triggers alarms and scaling actions. In steps 2.1 and 2.2 latest list of available servers is obtained from auto scaling group and it is updated in web application for future requests.

4.3.3 Scaling of Web Application Clusters

In common web applications each request gets its own execution thread upon arrival and this thread remains active during execution life cycle. Web application allocates required memory to each thread and each thread also require some computation. Although this computation can be very small in nature but if web application need to handle hundred and thousands of simultaneous requests then memory and computational requirements for thousands of concurrent requests can exceed from capacity of a single server. This scalability of web application can be performed by using standard products or services of AWS. ELB, CloudWatch and ASG products of AWS has standard features to support scaling based on increasing number of user requests. Web application need to support clustering in order to use these products.

Stateless web application instances sharing same database are required for high available cluster to provide reliability and continuous service even if one server fail [26]. AWS provides region based conceptual support to manage clusters of web application. A single ELB running under a specific regions can have multiple availability zones. Each availability zone can have one or more web application servers connected to same database to enable clustering. Usually one instance of web application in each availability zone are enough but multiple instances can be started by scaling-up if number of users increase beyond memory and computational capacity of available servers. Figure 4.5 explains clustering of web application over AWS products.
4.4 Implementation

Basic SaaS architecture shown in figure 4.2 required additional components to provide scalability and predictable performance for CIS. As described in previous sections of this chapter, queue-time input and scaling service has been used as supplementary components in this research. Together they can be used to design a SaaS architecture for CIS. Queue-time and service-rate calculation is done inside web application. Web application calculates service rate by collecting request execution time of requests handled in previous time span. Web application provides an end-point for external services or application to read queue-time $W_q$ at any time.

As a first step, AWS components are required. These components can be created by using AWS management console or by using AWS CF templates [27]. Required components for scalability are; ASG with scaling policies, CloudWatch metrics and alarms [23]. New servers are created by using LC with Amazon Machine Images (AMI) [6]. Each instance of web application have an auto scaling group attached to it. In standard AWS auto scaling groups can be attached to ELB but for compute intensive services we need auto scaling group for web application instances. For this purpose EC2 instance tags have been used. Multiple auto scaling groups are created in each environment and selected auto scaling group-id is saved in tag of web application instance. This adds an additional job for scaling service. On start of a web application instance, the scaling service should fetch list of available auto scaling groups and it should add group-id of first available auto scaling group to tag of web application instance. At least two AutoScalingGroup policies with CloudWatch alarms are required. Appendix A has example of cloud formation for these AutoScalingGroup policies and CloudWatch alarms. Below list has names of these policies and alarms:

- scalingCIServiceQueueTimeToHigha1
- scalingCIServiceQueueTimeToLowa1
CloudWatch alarm `alarmCIServiceQueueTimetoHigha1` is triggered if queue-time metrics value goes greater then threshold latency for evaluation periods. For example, if we have threshold latency $T_t$ as 3000ms, period length is 60sec and evaluation periods $E_p$ are 3 periods then `alarmCIServiceQueueTimetoHigha1` will trigger alarm when queue-time will go higher then $T_t$ for $E_p$ times. The scaling policy will perform action defined in `scalingCIServiceQueueTimeToHigha1` which is attached to `alarmCIServiceQueueTimetoHigha1`. Different actions can be defined in `scalingCIServiceQueueTimeToHigha1` based on linked `alarmCIServiceQueueTimetoHigha1`. In this case of high queue-time desired-capacity of auto scaling group changes to +1 which will add more resources to auto scaling group. These AWS components need continuous updates in queue-time metrics and they need to update web application after each alarm trigger resulting in scale-up or scale-down.

In this research the scaling service for supplementary features has been developed using Java. This scaling service runs beside web application. Java application runs as a service when AWS EC2 instance of web application starts. AWS provides SDK (software development kit) for Java applications [28]. This SKD provides standard Java objects that can be used to access and manipulate AWS resources including S3, EC2 services of ASG, CloudWatch and more [28]. Main AWS SDK objects required and used in this scaling service are;

- `AmazonAutoScalingClient`, Client for accessing Auto Scaling
- `AWSCloudWatchClient`, Client for accessing CloudWatch
- `AmazonEC2`, Client for accessing Amazon EC2

Scaling service need to get authenticated session with AWS API. For this purpose AWS profile details have been used and credentials are verified using AWS InstanceProfileCredentialsProvider object in Java [28]. After authentication, unused ASG is obtained from environment and its group-id is saved in tag of current web application instance. Algorithm 1 explains this step. ASG and web application EC2 instance should belong to same region and availability zone. ASG should not be in use of other web application EC2 instances. This can be validated by collecting ASG tags of all web application EC2 instances in same region. After start-up, scaling service runs an infinite loop to continuously update queue-time metric and server list. Algorithm 2 explains this step. Queue-time is retrieved from web application end point and it is updated in queue-time metrics of AWS CloudWatch by using Java CloudWatchClient. Web
application also calculates service rate and queue-time continuously. Calculation and update interval depends on each web application and it can vary from few seconds till one minute. Intervals more then one minute can increase delay in scale-up and scale-down actions.

Algorithm 1 GetASG Algorithm

```
for all EnvironmentASGForServices do
    if ASG ∉ Used and ASG.Instances ≥ currentASG.Instances then
        currentASG = ASG
    end if
end for
```

Ensure: `webAppInstance.Tag.ASG = currentASG.Id`
Ensure: `currentASG.min ≥ 1`

Algorithm 2 Update QueueTime and ServerList Algorithm

```
Require: currentASG ≠ null
loop
    for all Instances I ∈ currentASG.Instances do
        if I.State = running and I.healthchecks = OK then
            instanceList = instanceList : I
        end if
    end for
    metric.name = currentASG.Id – QueueTime"
    metric.value = getQueueTimeFromWebApp()
    metric.dateTime = DateTime.Now"
    Ensure: `updateWebApp(instanceList)`
    Ensure: `updateCloudWatchMetric(metric)`
end loop
```

4.5 Architecture and Runtime View

The SaaS implementation proposed in previous section helps to build SaaS architecture to provide scalability for compute intensive services. Figure 4.6 explains runtime architecture with scaling service and ASG for CIS. This architecture can scale in multiple zones and in multiple clusters of web application where each region has its own ELB [19]. Each web application instance manages its own ASG for servers of CIS.

User request arrives at ELB which uniformly distributes load among different availability zones and web servers running under each availability zone. ELB does not consider request type and therefore it can only distribute load by using number of simultaneous user requests. When a user request
arrives at web application then web application selects a compute service using one of below choice:

- If request has existing user session then reuse previous compute service
- If request has new session then allocate first available compute service

In first case, there is a high probability that request will be routed to an instance which already have cache. In second case there is less probability that request will be routed to an instance which already have cache but it is still possible. In both cases compute service will be allocated to handle user request and if no compute service is available then queue-time will start increasing.

Queue-time can also increase if too many user requests are sent to a web application and more importantly if more requests result as non-cached requests. These factors will reduce service-rate and therefore queue-time will increase. Scaling service loops on given period e.g. every 5th second or every minute to fetch queue-time from web application endpoint and upload queue-time to AWS CloudWatch metrics. AWS CloudWatch metrics triggers alarm on ASG by using alarm settings. These Alarms operate by using scaling policies to trigger scale-up or scale-down actions to perform scalability by adding or removing instances. Scaling service also loads avail-
able EC2 instances from ASG and it sends instance-list to web application in each cycle of same loop.

4.6 Experiment and Solution Evaluation

Experiments on SaaS architecture has been performed by using motivating scenario defined in 1.3. These experiments has been conducted by simulating scalability and simultaneous requests. Queue-time has been calculated for a period of one hour. Simultaneous requests starts from 15 requests per second and it raises till 450 requests in 30mins. In next 30mins simultaneous requests reduce back till 15 requests. Scaling service reads queue-time every 5th second and it updates CloudWatch metric. It is hard to predict when CIS will get more requests as cached requests. For experimental purposes it has been assumed that in last five seconds of every minute there are 15 non-cached requests. Experiments have below two assumptions for non-cached request. Three different experimental simulations has been performed.

In first experiment basic scalability has been used with single scale-out policy. In this case non cached request take 1000ms and initially architecture have two servers with 10 CIS. Below scaling polices have been used;

- **Scale-Up policy:** if $ST_q$ system total queue-time is $> 1500\text{ms}$ for two periods in $1\text{min}$ then update desired attribute of ASG as desired = desired + 2, to add two servers to architecture. No action for next four cycles/mins

- **Scale-Down policy:** if $ST_q$ is 0 for 96 periods/8mins update desired attribute of ASG as desired = desired - 2, to remove two servers from architecture.

Graph in figure 4.7 shows changing queue-time with increasing and decreasing simultaneous requests. It also shows server add and remove with services count on right or secondary vertical axis. It can be observed from graph that increase in simultaneous requests creates several load peaks represented as queue-time. Queue-time increase triggers scale-out and when more servers are added to the system then these load peaks decrease in size and eventually removed. In basic scalability queue-time has increased too high and it had gone above threshold latency several times. Queue-time has raised till 1200ms and at multiple occasions it has gone above threshold latency of 3000ms.

For second experiment, scaling polices can be optimized to further reduce these queue-time peaks. They can be updated as below;

- **Scale-Up policy-1:** if $ST_q$ system total queue-time is $> 1500\text{ms}$ for two periods in $1\text{min}$ then update desired attribute of ASG as desired
Figure 4.7: saas experiment single scaling, fast scale-out

Figure 4.8: saas experiment multiple scaling, lazy scale-out

= desired + 2, to add two servers to architecture. No action for next four cycles/mins

• Scale-Up policy-2: if $ST_q$ system total queue-time is > 1500ms for four periods in 5mins then update desired attribute of ASG as desired = desired + 2, to add two servers to architecture. No action for next four cycles/mins

• Scale-Down policy: if $ST_q$ is 0 for 96 periods/8mins update desired attribute of ASG as desired = desired - 2, to remove two servers from architecture.

• Scheduled policy: add two servers to architecture at 08:01:30 and at 08:20:30 for predicted load.

Graphs in figure 4.8 shows changing queue-time with multiple scaling polices including scheduled scale-out for predicted load at start of hour and
in middle of peak load time. It can be observed that queue-time peaks are much reduced and queue-times have occurred for very few times. Scale-up has been triggered multiple times and it helped to stop queue-time rising leaving few trivial queue-time load peaks. Queue-time remained under threshold latency of 3000ms most of the time and five queue-time peaks went above threshold latency of 3000ms. However queue-time still went above threshold latency and load peaks have not been completely removed. The second experiment has shown that optimized scaling policies helped to further reduce queue-time peaks. It is important to note that scaling polices vary for each scenario and in a SaaS architecture each organisation should optimize its scaling polices according to its requirements. Scaling policies can also be changed at any time using AWS management console.

Figure 4.9: saas experiment heavy cache building

Third experiment has been performed for CIS with higher cache building time. Graph in figure 4.9 shows queue-time and scalability for requests where cache build-time is 3000ms. Due to increase in cache-build time the threshold latency is considered as 6000ms. Graph in figure 4.9 shows experiment results for heavy cache building scenario. It can be observed from graph that queue-time peaks have been controlled by scaling events but queue-time has crossed threshold latency of 6000ms seven times. Queue-time has raised too high in three occasions with values 20000ms, 21500ms and 26500ms.

These experiments and graphs show that queue-time provides useful performance data for scalability. Queue-time peaks can be reduced by using SaaS architecture but few queue-time peaks still remain with standard scaling policies. These peaks can be reduced by optimizing scaling policies and adding scheduled scaling for predictable load. Changing scaling policies of
an ASG and changing initial minimum attribute of ASG are trivial and easy to perform tasks in cloud IaaS although it is still a manual operation. Scaling policy can be changed easily at any time by using AWS console and going into EC2 and ASG module. Required ASG can be selected and then scaling policies can be added or edited. Similarly minimum server requirements can be updated at any time by selected details of ASG and changing Min attribute.

Auto-scaling reduces chances of exponential growth of queue-time. Scheduled scaling can be performed with ASG scheduled actions in AWS. ASG can be configured to scale based on defined schedule [20]. Scheduled actions can be defined by using schedule frequency, start-time and end-time. For example; it can be scheduled; set ASG.Min and ASG.Desired equals to three from 08:00am till 06:00pm where default ASG.Min and ASG.Desired equals to two.

4.6.1 Limitations

A SaaS architecture for web application and CIS helps to automate server add or remove based on queue-time. The architecture detects queue-time load peaks which represents load peaks of simultaneous requests and cache building requests. Queue-time load peak growth can be stopped by adding more servers through CloudWatch alarms and ASG scaling policies. Servers are also successfully removed when not needed to save cost. However few high queue-time load peaks have been observed during experiments. Optimization of scaling polices and scheduled scaling helps to further reduce these peaks but it is still hard to completely automate and remove these queue-time load peaks.

First and most important limitation is AWS delay of server start-up. It can take three to five minutes in AWS to attach and start an AWS windows server. CloudWatch also triggers to add new server after couple of queue-time alerts and it is required to make sure that we have an actual peak and not a sudden behaviour due to cache building requests. Both these factors contribute to delay scale-out and during this time queue-time will keep on increasing. This creates a small gap between detection and actual scale-out. In this research we have used alarm value as half of threshold latency i.e. 1500ms of 3000ms. This helps to initiate early scale-out but there is still a possibility that queue-time peak can go above 3000ms threshold latency in this gap of four to five minutes. In experiments of this research this increase was trivial but it can be faster and then it can make more impact on overall system performance for a specific period of time.

It is also hard to automate optimization of scaling polices and scheduled scaling according to requirements of each organisation. It is to update scaling policies and add scheduled scaling but these are still manual operations. If minimum servers of an ASG are very low in number then a high queue-
time peak can occur on very next load. Scheduled scaling can help but it is hard to find or predict precise number of servers for minimum and this requirement can change every day. Both scale-out and scale-down actions can be performed by using $SQ_t$ (system queue-time) but $SQ_t$ does not provide complete information for scale-down. If scale-down is performed only on basis of $SQ_t$ then there is a possibility that removing a server will introduce queue-time and next increase of simultaneous requests can create high peak of queue-time. As a part of future work, scale-down can probably be done in two steps. In Step-1 an instance can be temporarily detached from an ASG and if no new queue-time increase is detected in couple of periods then this instance can be terminated else it can be attached back to ASG quickly.

Another limitation is related to data-sets with heavy cache-building time. Architecture cannot guarantee low latency and there can be high queue-time load peaks if scaling is not fully optimized. In-memory cache reduces latency if cache hit occur however it is not guaranteed. Whenever user is routed to a compute service which do not have cache ready for given data-set then cache build can happen. If cache build takes many seconds e.g. more then 3000ms then it is hard to keep latency under 3000ms because queue-time will be added over cache build time and it will always be greater then 3000ms. Scalability can reduce impact of cache building requests but low latency cannot be achieved by using only scalability solution. In this research main scalability solution has been designed and tested for performance factors of compute intensive services. Complete details about possible methodologies of state heavy scenarios is not main part of this research and an overview will be discussed in 4.8.

Solution proposed in this research is not appropriate for batch requests. Queue-time does not represent type of incoming requests and a batch request can contain many requests combined in one batch and it can occupy all resources. This can make huge increase in queue-time. Therefore it is important to keep batch requests out of regular request. If system has many batch requests then a scalability solution can be designed for batch requests queue. Web application should provide a user interface to separate regular requests from batch requests. Handling batch request is not part of this research but SaaS architecture proposed in this research can be enhanced with multiple ASG and multiple queue-time metrics to support batch requests. In this way, regular requests can be separated and protected from impact of batch requests. In-memory cache can help to perform batch requests rapidly. For example; if a batch request contain 2000 requests where each request execution can take 1000ms as non-cached requests and 100ms for cached requests then a batch job running on fiveCIS can take 40sec. Below are calculations for this example:

- Time to build cache for first request = $(1000 \times 5)/5 = 1ms$
- Time taken for 2000 requests = $(100 \times 1995)/5 = 39900ms = 39sec$
In a multi-tenant architecture resources are shared among organizations in a controlled way so no organisation effect performance of other organisation. For CIS a multi-tenant architecture can decrease or increase overall system performance. A multi-tenant architecture has been proposed in this research based on SaaS architecture of this research with below considerations:

- Each organisations should always have basic resources available
- Organisations should be able to get extra resources when required

Architecture defined in figure 4.10 explains proposed multi-tenant architecture. In this architecture each organisation has their own web application clusters for each region and availability zone. Each web application has scaling service and it is connected to an ASG of CIS running desired servers. Organisations and web applications do not share ASG and scalability is performed inside each ASG by using queue-time of its specific qeb application cluster. A multi-tenant ASG can be created in each region and availability zone. EC2 instances running in this multi-tenant ASG can be used to perform immediate scale-up. This can also help to address the AWS EC2 start-up delay limitation.

For example; if scaling-service of web application cluster $WC_a$ receive queue-time alerts to trigger a scale-out then scaling service can pick a running server from multi-tenant ASG and add it to its list of available servers for web application cluster $WC_a$. This will reduce scale-up time of few minutes to few seconds. Multi-tenant ASG will automatically scale-up to keep instances pool [6]. AWS cloud watch alarm will also trigger a scale-out for $WC_a$ and new server will be initialized for ASG of CIS. Multi-tenant ASG server can be removed after a specific time-period when new server is available in ASG of $WC_a$. Algorithm 3 presents an updated version of algorithm 2 for multi-tenant ASG.

This approach can help to improve performance but due to compute intensive services it also has some disadvantages. If high increase have been
**Algorithm 3 Update QueueTime and ServerList Multi-Tenant Algorithm**

**Require:** \( \text{maxQueueTime} \neq \text{null} \)

**Require:** \( \text{currentASG} \neq \text{null} \)

```
loop
    for all \( \text{Instances Inst} \in \text{currentASG.Instances} \) do
        if \( \text{I.State = running and I.healthchecks = OK} \) then
            \( \text{instanceList = instanceList + Inst} \)
        end if
    end for

\( \text{queueTime} = \text{getQueueTimeFromWebApp()} \)"

if \( \text{queueTime} \geq \text{maxQueueTime} \) then
    \( \text{multiTenantInstance = borrowMultiTenantASG()} \)
    \( \text{instanceList = instanceList + multiTenantInstance} \)
    \( \text{borrowInstanceId = multiTenantInstance.Id} \)
    \( \text{borrowInstanceStarttime = getTimeNow()} \)
end if

\( \text{metric.name} = \text{currentASG.Id – QueueTime} \)"
\( \text{metric.value} = \text{queueTime} \)"
\( \text{metric.dateTime} = \text{DateTime.Now} \)

**Ensure:** \( \text{removeBorrowInstanceFromListIfTimeOut(instanceList)} \)
**Ensure:** \( \text{updateWebApp(instanceList)} \)
**Ensure:** \( \text{updateCloudWatchMetric(metric)} \)

end loop
```
observed for queue time during scale-up delay then it can be handled by adding a running instance from multi-tenant ASG. This multi-tenant ASG will be removed after a specific period. This means cache will be lost and more non-cached requests will appear in system. A multi-tenant ASG solution is useful if rapid increase in queue-time is detected during regular scale-up delay of four to five minutes. If queue-time does not cross threshold latency or if increase is trivial then multi-tenant ASG may not help in increasing performance. Few examples when multi-tenant ASG pool can help are:

- queue-time crossing threshold latency
- queue-time is double then alarm value
- queue-time can increase rapidly due to state heavy scenarios

Cloud computing providers also offer low price resources which can be used for shared or spontaneous usage [30]. In AWS cheaper servers called spot instances are offered. These spot instances can be used in multi-tenant pool [29]. This can reduce cost of multi-tenancy and it can help both provider of CIS and tenant organisation to save cost and achieve performance. Usage of spot instance need reliability planning and reserve instances should be used as fall back option [30].

4.8 Evaluation for State Heavy Scenarios

Scalability solution proposed in this research tries to reduce queue-time load peaks from system but it does not improve in-memory cache usage and low latency is not guaranteed. End user can experience low latency on cache hit but latency will be higher if no cache hit occurs. This means threshold latency should be higher then latency of non-cached request handling time. There are scenarios when cache-building takes much longer time due to heavy state of data-set. In state heavy scenarios the usage of in-memory cache has much bigger impact on cost and performance.

Cache availability is also not guaranteed in scalable architecture. Every time a new server is added then it has to rebuild its in-memory cache. Similarly every time a server is removed in-memory cache is lost. State heavy scenarios can also take more memory and each service has a specific size of in-memory cache. When memory limit is meet then either service should get more memory or it has to clear its memory for new cache. These scalability and cache availability factors increase cache building and it is not possible to use in-memory cache for all requests.

More requests getting cache hits means smaller queue-time which leads to less number of servers. This can help in achieving better performance and cost reduction. Dedicated server instances can help to keep in-memory
cache available once built but if there is a high frequency in change of core data, say every day or every hour, then this cache of dedicated servers can also becomes invalid.

In order to achieve low latency for compute intensive services, updates are required in both web application routing algorithm and scaling architecture. Scalability solution proposed in this research can be further developed to achieve low latency. In next section a solution proposal have been presented to achieve low latency by maintaining cache availability. Solution proposal is based on SaaS architecture and scalability solution presented in this report. Future work can be performed to experiment and evaluate this solution proposal.

4.9 Achieving Low Latency with SaaS Scalability (Future Work)

Achieving low latency requires web application to route request to a cache ready service. Cache ready services means there is a compute intensive service with cache-set ready for a specific data-set. A cache ready instance can handle more requests every second then a service instance which gets cache building request. If infrastructure has more cache ready instances then less number of instances will be required and these cache ready instances can replace multiple service instances [31].

Required routing technique have two steps; in first step the compute intensive services prepare cache-set on service start and on data-set state update. In second step web application maintain a cache-table and route request to a specific service which already have cache. Both these steps can introduce new limitations and requirements as below;

- How to write snapshot and build cache of a compute service
- Cache of all data-sets is not required all the time
- Few data-sets can have more frequent usage then other
- Specific services can become bottle-neck
- During cache building a service is not available.
- If memory limit has meet then cache service need to loose old cache.

Building cache in compute service is a web application and service specific task. A Compute intensive service can write its state as a snapshot on a shared storage. AWS Simple Storage Service (S3) can be used for this purpose. S3 can synchronize a specific disk folder from a specific EC2 instance to a S3 folder. Any other EC2 instance can synchronize this S3 folder to its local disk folder. On start, a compute intensive service can build its
in-memory cache using snapshot state. Memory requirements for data-set state and cache hit rate should be saved and used for cache construction [31].

Web application should maintain a Cache Routing Table (CRT) to save cache-hit or queue-time details of each service. Scaling service can read these cache-hits or queue-time detail on regular intervals and upload it to AWS S3 storage and AWS cloud watch metrics. AWS cloud watch can use these metrics and it can trigger scale-up based on alarm condition. An alarm example; if a service has cache of 10 data-set states and it gets more then 100 hits per second then trigger alarm to add a new cache-ready instance. This alarm is based on assumption that all data-set states require same amount of cache memory and a service can keep maximum of 10 data-set states. If data-set states can have variance in required cache memory then expected cache size calculation is required and alarm should be updated.

Dynamic scalability can add and remove on-demand servers and we can perform server initializing actions using cloud IaaS features. Limitations like ‘service bottleneck’ and ‘add more cache memory’ can be solved by using scalability. New server $S_n$ can load routing-table statistics from S3 storage and it can select the service $S_h$ with highest queue-time or cache hits. $S_n$ service can perform cache building by load snapshot details of $S_h$ service from AWS S3 storage. $S_n$ service can update AWS EC2 instance tags with its cache-list. Scaling service can read cache-list details of each AWS EC2 instance and it can send it back to web application on regular intervals. Web application is responsible to perform cache-aware routing and send cache-hit or queue-time details to scaling service. Figure 4.11 explains this proposed architecture. Example tables for routing and cache-hit details can be found in Appendix B.

Achieving low latency by using in-memory cache can help to update core cache with zero downtime and it can help to reduce infrastructure cost. For
example, if cache building time is 3000\textit{ms} and 15\% requests can trigger cache building then at a specific time for 500 requests there is a need of 24 servers each running five CIS. Below are calculation details;

- Arrival Execution Load $A_e = (75*1000)+(425*100) = 1175000$
- Servers Required $S_r = \frac{A_e}{1000/5} = 24$ Servers

With low latency solution most requests can be handled by using in-memory cache and latency will be 100\textit{ms}. For popular requests threshold latency can be reduced to 1500 with queue-time alarm triggering on 500\textit{ms}. For 500 requests there will be a need of 10 servers each running five CIS. This can reduce infrastructure cost with predictable low latency. Below are calculation details;

- Arrival Execution Load $A_e = (0*1000)+(500*100) = 50000$
- Servers Required $S_r = \frac{A_e}{1000/5} = 10$ Servers

4.9.1 Limitations and Considerations

This solution can increase scale-up delay due to cache building at start. This delay depends on cache building time but it should not be big. Calculations in below example shows that cache building time can be less then two minutes.

- time required to build a cache $C_b = 3000\textit{ms}$
- memory required for data-set build and execute $M_r = 30mb$
- total memory size of a service $M_t = 1000mb$
- possible number of data-sets in each service $D_p$
  $= \frac{M_t}{M_r} = 1000/30 = \text{aprox. 30}$
- time to build total cache $C_t$
  $= D_p * C_b = 3000 * 30 = 90000\textit{ms} = 90\text{sec}$

This solution proposal requires at least one cache of each data-set in CIS. Creating at least one cache of each data-set is sometimes not possible due to usage of old data-sets and only latest or popular data-sets can be cached. It is also a business decision if there is a need to create at least one cache of each data-set or only popular data-sets should be cached. In case of ’popular data-sets’ there are two more considerations. Identification of ’popular data-sets’ should be performed on regular basis, for example every morning. A new ASG should be prepared using latest popular data-sets and it should replace active ASG. At least one server should be available to receive new unique requests. If number of data-set states sent to a specific server exceed allocated memory then another server should be added for unique requests using scalability.
4.9.2 Alternative Approaches

An alternative approach is shared memory or Inter-Process Communication solution [32]. In this approach CIS may share in-memory cache if they are on same server. Shared memory solutions can require extra development at application level and shared memory solutions also have limitations which can reduce system performance. Shared memory solutions cannot hold cache for too many or large data-sets and cache can also have concurrency issues. In shared memory solutions cache size have to be increased after sometime and an increase in cache size may raise garbage collector issues in few computer language implementations.

AWS ElastiCache is another approach with scalable cache solution. It is a cloud service that can be used to deploy, operate, and scale in-memory cache of a compute service [33]. ElastiCache can improve performance by usage of scalable and fast in-memory caches [33]. However, there is a need to run multiple CIS to support simultaneous user requests. AWS ElastiCache supports cache scalability for a single compute service which does not provide a performance gain for increasing simultaneous requests.
Conclusion and Discussion

In this research initial experiments have been conducted on dedicated servers and then a SaaS architecture has been presented for compute intensive services. Study of dedicated servers has shown that it is difficult to find required number of dedicated servers for cache optimized CIS. If very few servers are used considering cached requests then performance will degrade during load peaks but at the same time if many servers are used considering peak load then cost becomes too high and servers are also under used in off load time. Dedicated servers can give cost-effective performance if number of simultaneous requests is very low and cache remains static for longer periods. However, support for both these factors is essential for enterprise business.

SaaS architecture has been built using cloud computing products of AWS, queue-time metric and scaling service. AWS products like ASG, CloudWatch, LC and ELB has been used to build infrastructure. Experiments on SaaS architecture has shown that queue-time can represent performance data for scalability. Scale-out and scale-down event can be triggered by using queue-time inputs which helps to reduce load peaks while keeping cost low. It has been observed that proposed approach does not remove all load peaks. Optimizations of scaling polices and scheduled scaling for each organisation can help to remove these load peaks further. Server start-up time has been observed as three to five minutes which does not allow robust and immediate scaling. Single tenant SaaS architecture can help to achieve cost-effective performance but there are still possibilities of small load peaks due to server start-up delay. These load peaks are eventually handled by the SaaS architecture with a scale-out event but these load peaks can reduce performance for a small period of time.

The single-tenant SaaS architecture of this research has been used to propose a multi-tenant SaaS architecture. This architecture uses a shared ASG which can help to achieve immediate scalability and it can remove small peaks created due to server start-up delay. Although in this research we have discussed an approach to achieve cost-effective performance for CIS, we believe that this approach can be further developed to achieve low latency and usage of in-memory cache can be further optimized. Proposal to achieve low latency is based on CRT and it tries to remove cache building requests. This proposal to achieve low latency for CIS is also based on SaaS
Future Work

Experiments have shown that queue-time load peaks cannot be completely removed in single tenant SaaS architecture. Optimizations of scaling polices and scheduled scaling can help to reduce load peaks but it is a manual task today. Input for scheduled scaling can be calculated and automated by using scalability data of last day with machine learning and prediction algorithms. Dynamic scaling will still provide scalability for unpredictable load of simultaneous requests and cache building requests.

Another reason of these remaining load peaks is early scale-down. If scale-down is performed on zero queue-time then it is doubtful if server is still partially required or not. Scale down can be delayed and performed only if server is completely free. Sale-down can be developed further to perform delayed scale-down in two steps. In Step-1 an instance can be temporarily detached from an ASG and if no new queue-time increase is detected in couple of periods then this instance can be terminated else it can be attached back to ASG quickly.

Experiments in this research have been performed by using generated queue-time and it is difficult to perform variety of scalability experiments for different expected scenarios. There is a need to design a simulation system or framework to test scaling-policies and alarms of SaaS architecture. It should take required number of simultaneous and cache building requests as input and test run given scaling policies and scheduled scaling. It can help to understand system performance, optimize architecture and scaling policies.

SaaS architecture can be further developed for proposed multi-tenant architecture and for batch requests handling. This research has been conducted to reduce load peaks but this architecture can be further developed to achieve low latency for CIS. It can help to get predictable and low response latency and data-sets with large cache building time can also be optimized.
Appendices
Appendix A

AWS CloudFormation for Scaling Policy and Alarms

```
{
    "scalingCIServiceQueueTimeToLowA1": {
        "Type": "AWS::AutoScaling::ScalingPolicy",
        "Properties": {
            "AdjustmentType": "ChangeInCapacity",
            "Cooldown": "300",
            "ScalingAdjustment": "-1",
            "AutoScalingGroupName": {"Ref": "CIServiceA1"}
        }
    },
    "scalingCIServiceQueueTimeNoActivityA1": {
        "Type": "AWS::AutoScaling::ScalingPolicy",
        "Properties": {
            "AdjustmentType": "ExactCapacity",
            "Cooldown": "10",
            "ScalingAdjustment": "0",
            "AutoScalingGroupName": {"Ref": "CIServiceA1"}
        }
    },
    "scalingCIServiceQueueTimeToHighA1": {
        "Type": "AWS::AutoScaling::ScalingPolicy",
        "Properties": {
            "AdjustmentType": "ChangeInCapacity",
            "Cooldown": "300",
            "ScalingAdjustment": "1",
            "AutoScalingGroupName": {"Ref": "CIServiceA1"}
        }
    },
    "alarmCIServiceQueueTimeNoActionA1": {
        "Type": "AWS::CloudWatch::Alarm",
        "Properties": {
```
"ActionsEnabled": "true",
"ComparisonOperator": "GreaterThanOrEqualToThreshold",
"EvaluationPeriods": "11",
"MetricName": {
    "Fn::Join": ["", ["Ref": "CIServicea1"], ",QueueTime"]}],
"Namespace": {
    "Fn::GetAtt": ["ToLowerCase", "Output"]},
"Period": "300",
"Statistic": "Average",
"Threshold": "0.0",
"AlarmActions": [{"Ref": "scalingCIServiceQueueTimeNoActivitya1"}]
},
"alarmCIServiceQueueTimetoHigha1": {
    "Type": "AWS::CloudWatch::Alarm",
    "Properties": {
        "ActionsEnabled": "true",
        "ComparisonOperator": "GreaterThanOrEqualToThreshold",
        "EvaluationPeriods": "3",
        "MetricName": {
            "Fn::Join": ["", ["Ref": "CIServicea1"], ",QueueTime"]],
            "Namespace": {
                "Fn::GetAtt": ["ToLowerCase", "Output"]},
        "Period": "60",
        "Statistic": "Average",
        "Threshold": "1000.0",
        "AlarmActions": [{"Ref": "scalingCIServiceQueueTimeToHigha1"}]
    }
},
"alarmCIServiceQueueTimetoLowa1": {
    "Type": "AWS::CloudWatch::Alarm",
    "Properties": {
        "ActionsEnabled": "true",
        "ComparisonOperator": "GreaterThanOrEqualToThreshold",
        "EvaluationPeriods": "15",
        "MetricName": {
            "Fn::Join": ["", ["Ref": "CIServicea1"], ",QueueTime"]],
            "Namespace": {
                "Fn::GetAtt": ["ToLowerCase", "Output"]},
        "Period": "60",
        "Statistic": "Average",
        "Threshold": "0.0",
        "AlarmActions": [{"Ref": "scalingCIServiceQueueTimeToLowa1"}]
    }
}
Appendix B

CRT For Achieving Low Latency Proposal

<table>
<thead>
<tr>
<th>1st</th>
<th>Incoming requests [Bold = Cache Hits]</th>
<th>Routing Table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R1, R3, R4, R5, R1, R3, R4, R5</td>
<td>Instance-001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instance-002</td>
</tr>
<tr>
<td>2nd</td>
<td>R6, R4, R9, R1, R10, R3, R7, R5</td>
<td>Instance-001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instance-002</td>
</tr>
<tr>
<td></td>
<td>6th unique request (R9) received we expect system</td>
<td>Instance-003</td>
</tr>
<tr>
<td></td>
<td>to receive more than 8 requests triggering</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scale-Up i.e. add Instance – 003</td>
<td></td>
</tr>
<tr>
<td>3rd</td>
<td>R7, R3, R2, R5, R10, R9, R1, R8</td>
<td>Instance-001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instance-002</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instance-003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R1, R3, R4, R50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R6, R7, R9, R10</td>
</tr>
</tbody>
</table>

Table B.1: routing table to handle cache overflow and runtime limitations
<table>
<thead>
<tr>
<th>Incoming requests</th>
<th>Queue Time</th>
<th>Routing Table</th>
<th>Queue Count/Additional Request Count Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st R1, R3, R4, R5, R4, R3, R1, R5</td>
<td>1</td>
<td>Instance-001 R1, R3, R4, R5</td>
<td>R1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instance-002 R1, R3, R4, R5</td>
<td>R2 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instance-003 R1, R3, R4, R5</td>
<td>R3 1</td>
</tr>
<tr>
<td>2nd R6, R4, R9, R1, R10, R3, R7, R5</td>
<td>1</td>
<td>Instance-001 R1, R3, R4, R5</td>
<td>R1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instance-002 R1, R3, R4, R5</td>
<td>R2 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instance-003 R6, R7, R9, R10</td>
<td>R3 0</td>
</tr>
<tr>
<td>3rd R7, R11, R2, R5, R10, R12, R1, R8</td>
<td>1</td>
<td>Instance-001 R1, R3, R4, R5</td>
<td>R1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instance-002 R6, R7, R9, R10</td>
<td>R2 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instance-003 R2, R8, R11, R12</td>
<td>R3 0</td>
</tr>
<tr>
<td>4th R5, R3, R2, R5, R2, R5, R3, R5</td>
<td>4</td>
<td>Scale Instance-001 R1, R3, R4, R5</td>
<td>R1 1</td>
</tr>
<tr>
<td></td>
<td>Up</td>
<td>Instance-002 R6, R7, R9, R10</td>
<td>R2 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instance-003 R2, R8, R11, R12</td>
<td>R3 0</td>
</tr>
<tr>
<td>5th R7, R3, R2, R5, R1, R9, R3, R8</td>
<td>1</td>
<td>Instance-001 R1, R3, R4, R5</td>
<td>R1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instance-002 R6, R7, R9, R10</td>
<td>R2 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instance-003 R2, R8, R11, R12</td>
<td>R3 0</td>
</tr>
</tbody>
</table>

Figure B.1: CRT Table 2
Bibliography


[8] Defining Standards for Web Page Performance in Business Applications Rempel, Garret John, Lizy, K. (General Chair) ; Smith, Connie, U. (General Chair) ; Sachs, Kai (Program Chair) ; Lladó, Catalina, M. (Program Chair)

[9] Sumit Sharma, Dan Diephouse, Katie Burton. MuleSoft, SaaS Provider Handbook (How to Make Your App Enterprise Ready ), Ac-


[32] Inter-process communication https://en.wikipedia.org/wiki/Inter-process_communication
