ECE547/CSC547 Cloud

Computing Fall 2023

Project

Concert Ticket Booking Website

Prasanna Venkatesh Thambuanantharaman(pthambu), Girish Gopalakrishnan Nagarajan(ggopala4)

1 Introduction

1.1 Motivation

Describe your motivation in doing this project[1].

Our project is to design a cloud architecture for a Concert Ticket Booking website. We got the motivation for this project after seeing the Taylor Swift-Ticketmaster controversy [1] where Ticketmaster simply could not handle the traffic to their website leading to the website crashing numerous times and leaving fans frustrated. We think this will be a great problem statement for a cloud architect to solve.

1.2 Executive summary

Summarize, in one paragraph or two, the essence of this project. (Who are you writing this summary for?)

Our aim is to design a cloud architecture for a Concert Ticket Booking website that can service millions of users simultaneously with reasonably good response times. The website will handle every part of the ticket booking process, right from discovery (finding concerts to attend) to collecting feedback and customer support post concert.

2 Problem Description

2.1 The problem

What is the high-level problem you are designing for? Must be a realistic one, in order to derive good Business Requirements (BRs).

The primary challenge in this project is going to be in designing an architecture that can handle millions of service requests and workloads simultaneously, without crashing. In addition to that the architecture must also support other aspects of a ticketing website such as cross-platform support, analytics, secure payments etc.

2.2 Business Requirements

You must list here the business goals for the project.

BR1. 24/7 availability: The website should be available 24/7 with minimal or no downtimes. Architecture design should incorporate fault tolerance and scalability to ensure this requirement is met.

BR2. Disaster recovery: The Website should handle disaster and outage scenarios gracefully. Sufficient redundancies and backups should be included in the design solution to minimize the probability of such disaster scenarios.

BR3. Security: Design a secure architecture which provides safeguards against outside attacks, securely stores sensitive data and ensures authorized access to data.

BR4. Accommodate time varying workloads: We expect the website to experience time varying workloads with peak windows during opening of tickets for a popular concert. Therefore the architecture design should account for this and have sufficient measures to handle such peak traffic scenarios.

BR5. Ensure low latency: The website should suffer from minimal latency, especially when handling ticket booking and payment related workloads.

BR6. Optimize costs: The architecture should adhere to best practices and make reasonable tradeoffs in order to optimize the cost.

BR7. Reporting and analytics: The architecture should include a thorough reporting and analytics component in order to analyze trends as well as keep track of and improve website performance.

BR8. Legal and compliance: The architecture should be designed in such a way that it meets all the data protection standards as well as all the legal and compliance standards across regions.

BR9. Tenant identification: Design the architecture around the multi-tenancy principle.

BR10. Sustainability: The architecture should keep in mind sustainability principles and should be designed to minimize the carbon footprint.

2.3 Technical Requirements

You must list here the technical requirements (TRs) for the project.

The following requirements satisfy BR1:
TR1.1 The choice of cloud provider as well as the architecture design should implement sufficient redundancies (replicas in different availability zones) to ensure 24/7 availability. The cloud provider and the location/number of their infrastructure can play a crucial role in the availability of the application and should therefore be chosen carefully.
TR1.2 The architecture should avoid single points of failure since any failure along these points would seriously affect the availability of the application.
TR1.3 Implement robust testing to validate the architecture and the load managing capabilities of the infrastructure.
TR1.4 Distribute resources based on number of actual/expected users in each geographic zone.

Justification: Distributing resources across geographic zones ensures adequate availability, in case any region experiences an outage. Further, distributing the load across multiple replicas also reduces latency as well as avoids single point of failure. Such distribution must be based on the number of users, either actual (if that metric is available) or expected, in order to ensure each zone has sufficient resources as well as to ensure each resource is correctly configured to handle such a load.

The following requirements satisfy BR2:
TR2.1 The architecture should implement automatic failure detection and recovery mechanisms so that the application can gracefully handle disaster scenarios.
TR2.2 Implement a multi-cloud solution to minimize the risk of application failure if a cloud provider's infrastructure has failed.

Justification: The health of each resource in the infrastructure is continuously monitored and if any failure is detected, appropriate recovery mechanisms are triggered, which include redistributing the load, decommissioning/ fixing the affected resource and provisioning new resources. In order to further protect the application against disaster scenarios where the entire infrastructure of a cloud provider has failed, a multi-cloud solution is required where the application is deployed on the infrastructure of another cloud provider.

The following requirements satisfy BR3:
TR3.1 Adhere to Data best practices such as encryption of data, especially sensitive customer data both while in rest and while in transit.
TR3.2 Implement a firewall to protect cloud resources from unauthorized access and outside attacks.
TR3.3 Set up strict Identity access management to ensure only authorized users can access the data.

Justification: By encrypting data and adhering to the data best practices we ensure that customer data is protected. In addition, by implementing a secure firewall and strict IAM policies we protect the application from unauthorized/malicious actors.

The following requirements satisfy BR4:
TR4.1 Monitor workloads and automatically scale up or down to meet capacity requirements.
TR4.2 Implement a load balancer to distribute load across resources and replicas to handle bursty

workloads and maintain good response times.
TR4.3 Distribute resources across regions and availability zones to both ensure redundancy as well as to minimize latency.

Justification: Utilizing a load balancer and implementing automatic scalability helps the application to handle periods of both high and low traffic efficiently. During peak load the auto scaler provisions additional resources to handle the traffic while the load balancer efficiently distributes the load across these resources. Once the period of peak traffic has subsided the auto scaler can then decommission unwanted resources to prevent underutilization and optimize costs.

The following requirements satisfy BR5:
TR5.1 Constantly monitor network utilization, traffic patterns to reduce latency, distance and jitter.
TR5.2 Setup replicas of the application across geographic zones to ensure location agnostic performance and low latency.
TR5.3 Implement caching, in addition to distributing resources across availability zones, to further reduce latency.
TR5.4 Utilize a Content Delivery Network to serve website content closer to the user and thereby further reducing latency.

Justification: By setting up application replicas closer to users, by implementing caching and using a CDN to serve website content to the users, we minimize the physical distance between cloud infrastructure and the user which in turn reduces latency. In addition, through constant monitoring of the cloud network and optimization we further reduce the latency.

The following requirements satisfy BR6:
TR6.1 Implement auto scaling to make sure resources are provisioned only when required.
TR6.2 Reduce costs further by storing old but important data like old logs etc. in cold storage.
TR6.3 Automate and optimize workflows by using CI/CD pipelines, Infrastructure as Code services and serverless computing to reduce costs associated with development, manual creation and provisioning of resources etc.
TR6.4 Setup appropriate load/utilization thresholds for each resource and scale horizontally only when the threshold target is hit. If a particular resource is noticed to be constantly underutilized or it cannot handle the workload, scale vertically.

Justification: By utilizing autoscaling, load balancing, automated workflows, and continuous monitoring we ensure that the infrastructure is utilized efficiently, preventing both over-provisioning of resources as well as underutilization of provisioned resources which in turn reduces costs. In addition to resource utilization costs we can further minimize storage costs by storing old data like logs etc. in cold storage since such data would be retrieved/used very rarely. Storing data in cold storage increases the time required to retrieve such data but it significantly reduces the storage costs, thus it makes sense to store old data in cold storage.

The following requirements satisfy BR7:
TR7.1 Monitor website performance, usage trends and user behavior to discover insights which can be used to further improve performance, improve security or optimize costs. For example if we discover

that the load on the website increases during specific times of the day we can prepare for it by having additional resources on standby for those particular times of the day.
TR7.2 Monitor essential application metrics such as traffic, service utilization metrics, session length, time required to complete the purchase of a ticket etc to identify bottlenecks and resolve them.
TR7.3 Collect logs of compute instances and store them for analytics purposes.

Justification: By utilizing a managed monitoring service we can keep track of application metrics and take appropriate action. By incorporating a machine learning component along with the monitoring and reporting component, we can generate actionable insights to further improve application performance.

The following requirements satisfy BR8:
TR8.1 Implement industry standard encryption of data both while in transit and while in the database. Ensure proper usage and disposal of cookies in accordance with data protection laws.
TR8.2 Conduct regular audits of infrastructure, resource usage and security policies.

Justification: By following data protection standards, encrypting sensitive data, disposing of cookies in accordance with data protection laws and by conducting regular audits we can ensure that our application follows all the legal and compliance standards required to operate in a region/country.

The following requirements satisfy BR9:
TR9.1 Ensure appropriate data partitioning, isolation and monitoring of resource usage to make sure data is secure and there is no authorized access.
TR9.2 Ensure each tenant (user) of the application has a separate identity store and customize the website experience like concert recommendations etc. for each user.

Justification: We need to ensure that the tenant/user data is sufficiently isolated from those of other tenants and that each tenant of our application has a unique identity, to sufficiently implement appropriate tenant identification and multi-tenancy model.

The following requirements satisfy BR10:
TR10.1 Measure and report the carbon emissions of provisioned resources.
TR10.2 Ensure architecture maximizes resource utilization and energy efficiency. For example two hosts running at 40 percent utilization are less efficient than one host at 80 percent utilization due to baseline power consumption per host [2].

Justification: By ensuring maximum resource utilization and energy efficiency, we lower the carbon footprint of our provisioned resources. In addition, we monitor and report the carbon emissions of the infrastructure to both further optimize the usage of resources as well as to meet well architected framework standards.

2.4 Tradeoffs

Every design needs to make tradeoffs[2]. If possible, include here (some of the) tradeoffs you may have made, *without even having started the design process*; mention your justifications for these tradeoffs.

1. Cost optimization (BR6) vs High availability (BR1): Ensuring high availability, especially 24/7 availability would require us to configure and deploy redundancies across different regions and availability zones (and also across multiple clouds) which would naturally increase costs. However we opt to prioritize high availability in order to ensure a good experience for users.

2. Sustainability (BR10) vs High availability (BR1) and Low latency (BR5): Since we are opting to prioritize high availability and low latency, the deployment of these resources would naturally increase the carbon footprint of the overall application.

3. High availability (BR1) vs Security (BR3) and Legal (BR8): There might be cases where legally we cannot store some sensitive data in certain geographic locations or across multiple clouds. In such cases we would choose to satisfy the legal and compliance requirements over our availability requirements.

4. Cost optimization (BR6) vs Security(BR3): Secure storage, encryption introduces further complexities and costs but we choose to prioritize this requirement over cost optimization in order to ensure safety and security of user data.

5. Low latency (BR5) vs Security(BR3): In addition to higher costs, secure storage and secure transmission(encryption) introduces complexities that slows down the process. Balancing stringent security with an optimized, low latency network requires tradeoffs. However since security, especially of sensitive data, is our utmost priority, we decide to opt for secure storage and transmission over further reduction in latency.

3 Provider Selection

You will need a set of services, in order to fulfill the TRs you have listed in Section 2.3. In this section, you must search for the cloud provider that offers most, if not all of these services. You must include at least three providers. At this stage of the game, we do not expect you to know all the details of all the required services.

3.1 Criteria for choosing a provider

Supply the criteria you will use for choosing a provider.

To assess whether a service provider would be a good fit for our application or not we look at their service offerings and see if those services satisfy some of the key requirements for our application. The below list offers us a jumping off point to begin evaluating service providers but by no means is the list exhaustive. It contains some of the key aspects of a ticket booking website and thus would provide us with an effective criteria to choose cloud providers.

Relational Database: The data associated with a concert ticket booking website is relational in nature, with strong relationships between customers, the concerts they attend, their transactions etc. Therefore the chosen cloud provider must offer an excellent relational database along with services to effectively it.

Key-value Database: To ensure the application can handle peak traffic scenarios we need to implement a key-value database designed to offer single-digit millisecond response time at any scale. Therefore the chosen cloud provider must offer a key-value based database service along with support for integrations to other services such as analytics, caching, replication etc.

High availability and auto scaling: Since we expect the application to face time varying, bursty traffic, it would require a load balancer to distribute traffic as well as auto scaling to handle spikes in traffic. In addition the application must have near 24/7 availability. Therefore the chosen cloud provider must offer robust auto scaling services, load balancing services and near 24/7 availability.

Compute: The compute infrastructure is one of the most crucial aspects of our application. The application needs to process multiple thousands of transactions per second, especially during times of peak load. The compute infrastructure is also responsible for executing database queries. It would also be responsible for handling inter-process and inter-service communications with the help of a message queueing service. Therefore the chosen cloud provider must offer state of the art compute infrastructure.

Minimal latency: Minimal latency is of utmost importance to our ticket booking application. Ticket booking transactions, database queries, website content all need to be processed and delivered to the end-user in the shortest amount of time as possible and thus the chosen cloud provider must have infrastructure and services such as a Content Delivery Network capable of minimizing all types of latencies associated with the website and its operations.

Security: Security is another crucial pillar of the application. All operations, data and transactions of the website must be secured. This involves encrypting data, using secure inter-process communication, implementing stringent user access policies etc. Thus the chosen cloud provider must offer services that are capable of implementing the highest security standards.

Cost optimization: Utilization of such state of the art resources, databases and infrastructure would naturally increase the costs associated with running the website. Therefore the cloud provider must offer services to optimize costs so that we do not exceed the infrastructure budget.

Monitoring: Resource utilization, application performance metrics, traffic metrics etc need to be continuously monitored and reported so that cloud architects and cloud engineers have reliable data to inform their decision making, as well as to further optimize infrastructure usage.

3.2 Provider Comparison

Supply a table with the ranking of the providers you considered. Justify the rankings.

Criteria	Services offered by AWS	Services offered by Azure	Services offered by GCP	Our selection	Justification
Relational database	Amazon RDS and Amazon Aurora	Azure SQL	Google Cloud SQL	AWS	Amazon Aurora provides excellent performance and provides flexibility when it comes to choosing a database engine, with full compatibility for both PostgreSQL and MySQL.
Key-value Database	Amazon DynamoDB	Azure Cosmos DB	Cloud Spanner	AWS	DynamoDB is a serverless key-value database which is designed to handle high performance applications at any scale.
High availability and auto scaling	AWS Auto Scaling, AWS Elastic Load Balancing	Azure Front Door, Azure Traffic Manager, Azure application gateway and Azure Load Balancer	Google Cloud Load Balancing, Auto Scaling with Managed Instance Group	AWS	All 3 providers offer good availability with resources distributed across the globe. Therefore the choice of the provider

					depends on the auto scaling services provided and in this aspect AWS provides better, more comprehensive services
Compute	AWS Lambda, AWS Elastic Beanstalk, AWS App Runner	Azure Functions, App Service, Virtual Machine Scale Sets	Compute Engine, App Engine, Cloud Run	Tie between all 3	All 3 providers offer competitive services and therefore the choice of the provider depends the choice for the rest of the services
Minimal Latency	AWS Direct Connect, AWS Global Accelerator, AWS Local Zones, AWS CloudFront	Network as a Service, Proximity Placement Groups, Receive Side Scaling, Azure CDN	Network as a Service, Network Analyzer, Cloud Trace, Network Connectivity Center, Google CDN	AWS	AWS provides a comprehensive suite of services specifically aimed at reducing latency.
Monitoring	AWS CloudWatch, AWS Config, AWS CloudTrail	Azure Monitor, Azure Application Insights	Cloud Monitoring, Cloud Logging, Cloud Trace	Tie between AWS, Azure and GCP	All 3 providers offer comprehensive services for monitoring
Security	AWS offers over 20 different services to implement security best practices. A few of them include AWS IAM, AWS	Azure offers about 15 different services for security. A few of them include Azure Active Directory, Azure Firewall,	GCP offers around 12 services for security, which include Google IAM, Google IDS, Google Cloud Firewall	AWS	While all 3 providers offer dedicated security services

	Security Hub, AWS Firewall Manager etc.	Azure Information Protection etc.			
Cost Optimization	AWS Cost Explorer, AWS Budgets, AWS Cost and Usage report	Microsoft Cost Management	Google Cloud Cost Management, Active Assist	AWS	AWS offers a comprehensive suite of services dedicated specifically for cost optimization

Table 1: Database comparison

3.3 The final selection

If there is a clear winner, just announce your final selection. If not, mention and justify the tradeoff you made.

We have opted for Amazon Web Services as our final choice of cloud provider. AWS offers a diverse portfolio of services to satisfy all the core requirements of our concert ticket booking website. This diverse portfolio combined with AWS's position as the market leader and a dedicated community of users provide us with confidence in our final choice.

3.3.1 The list of services offered by the winner

AWS offers over 280 products and services. A few of these offerings, along with their short descriptions are listed below.

1. Compute:
   a. Amazon Lambda: Lambda is a serverless, event-driven compute service that lets us run code without having to provision servers.
   b. AWS Elastic Beanstalk: Helps deploy and scale web applications.
2. Database:
   a. Amazon Aurora: Aurora is a fully managed SQL database which offers high performance and supports both MySQL and PostgreSQL.
   b. Amazon DynamoDB: DynamoDB is a fully managed key-value based database which offers single digit millisecond level performance.
3. Management and Auto Scaling:
   a. Amazon Cloudwatch: Cloudwatch monitors services and applications and helps optimize performance as well as provide insights into the health of resources.
   b. AWS Auto Scaling: Auto Scaling monitors applications and automatically scales capacity to maintain reliable performance at lowest possible cost.
4. Latency:

    a. AWS CloudFront: CloudFront is a Content Delivery Network (CDN) designed to offer high performance and security.
    b. AWS Direct Connect: It is a service that allows us to directly connect our application to the AWS network and bypass the public internet in the process.
5. Monitoring:
    a. AWS Config: Config continuously monitors, audits and evaluates the configurations of AWS resources as well as the relationships between them.
    b. AWS CloudTrail: CloudTrail tracks user activity and API usage across the AWS infrastructure.
6. Security:
    a. AWS Identity and Access Management: AWS IAM manages identities and access to AWS resources to ensure only authorized users can access the infrastructure resources.
    b. AWS GuardDuty: GuardDuty is threat detection service that monitors AWS accounts as well as workloads to detect any malicious activity.
7. Cost optimization:
    a. AWS Cost Explorer: Cost Explorer is a service that lets the user visualize and manage costs associated with usage of AWS resources over time.
    b. AWS Budgets: Budgets is a service that lets the user set a custom budget and alerts the user when action is required.

4 The first design draft

In this section, you must provide the first draft of your proposed design. It will help if you give this section a try *without consulting chapter 4* (especially Sections 4.4 through 4.6) of the class notes.

4.1 The basic building blocks of the design

We provide the first draft of our proposed solution in this section. We have identified the core requirements of our concert ticket booking website in the previous section, as well as compared the offerings of the major 3 cloud providers to determine the best fit. We have chosen AWS as our cloud provider and we will now provide the list of AWS services we plan to use for our solution and provide a description of each of them.

1. AWS Lambda: Lambda is a serverless, event-driven compute service that operates as Function as a Service(FaaS). We have opted for Lambda over other serverless compute services to minimize cost as well as to design for scalability. Lambda offers a wide range of event triggers.
2. Amazon DynamoDB: DynamoDB is a fully managed, serverless key-value based NoSQL database. It is specifically designed to offer single digit millisecond performance at scale.
3. Amazon EC2: Elastic Compute Cloud, EC2 provides scalable compute capacity. EC2 would be utilized to run most of the compute workloads for our application, such as managing and querying the database, handling transactions as well as creating logs.

4. Amazon Aurora: Aurora is a high performance, relational database with support for MySQL and PostgreSQL database engines. Aurora would be used to store customer data, such as account information, transaction history, booked concerts, concert listings etc.
5. AWS ElastiCache: ElastiCache is a full-managed in-memory caching service that delivers real-time, cost optimized performance. We implement our caching requirement using ElastiCache.
6. Amazon VPC: Amazon Virtual Private Cloud is a service that provides a logically isolated section of the AWS Cloud for your resources. VPC ensures a secure and isolated network for our concert ticket booking website. It further facilitates fine-grained control over the virtual networking environment including resource placement, security and access policies.
7. AWS API Gateway: API Gateway is a fully managed service that allows us to create RESTful API endpoints and invoke Lambda functions when a trigger occurs. These APIs are how the end-users are served website content as well as the results of the database queries and thus we require efficient, real-time communication, which is provided by API Gateway. In addition API Gateway also provides traffic management, caching, authorization and access control etc.
8. Elastic Load Balancing: Elastic Load Balancing is a service that automatically distributes incoming traffic across multiple targets. A load balancer is essential to distribute load across compute and storage instances to ensure efficient usage of resources as well as to maintain reliable availability of the application.
9. Amazon CloudFront: CloudFront is a Content Delivery Network (CDN) service that delivers content to users with minimal latency, based on their geographical location.
10. Amazon Elastic Kubernetes Service: EKS is a managed Kubernetes service to run Kubernetes in the AWS Cloud. EKS manages the scalability and availability of the Kubernetes clusters, in addition to scheduling containers, storing cluster data etc.
11. Amazon Route 53: Route 53 is a scalable and highly available Domain Name System (DNS) service. Route 53 is used to connect end users to our application running on the AWS cloud.
12. Amazon SageMaker: SageMaker is a fully managed service that offers tools to build, train and deploy Machine Learning models at scale. We utilize SageMaker to train and deploy a Machine Learning model to discover insights from our data.
13. AWS Identity and Access Management: IAM is used to specify access policies, manage permissions and analyze access to refine such permissions.
14. AWS Firewall Manager: Firewall Manager is a security service that allows us to configure and manage firewall rules across our AWS infrastructure.
15. AWS Web Application Firewall: WAF is a security service that protects our web application from common web exploits and bots. These exploits and bots can compromise the security of our application or can consume excessive resources and thereby increase our costs.
16. Amazon Simple Storage Service: Simple Storage Service(S3) is an object storage service which offers excellent scalability, performance and, availability and security.
17. Amazon CloudWatch: CloudWatch is a monitoring service that monitors applications, optimizes resource usage and provides insights into operational health.
18. Amazon Macie: Macie is a security service that uses Machine Learning to discover and protect sensitive data.
19. Amazon Key Management Service: Amazon KMS allows us to create and manage cryptographic keys used to encrypt our data.

20. AWS Security Hub: Security Hub is used to automate security best practice checks and aggregate all security alerts related to our AWS resources in a single place.
21. Amazon Cognito: Cognito is used for tenant identification and authorization. In addition it offers support for social and enterprise identity federation and advanced security features to help protect customers.

4.2 Top-level, informal validation of the design

Provide arguments that your solution will work; that is, your design will achieve the TRs you came up with in Section 2.3. The instructors (or even yourself) may find the arguments “weak”; *that’s OK at this stage of the game.*

In this section we provide our plans on how to utilize the above mentioned services and some arguments to show that these services can be utilized to satisfy our Business and Technical requirements. Note that the following arguments will be further refined and solidified in the next section.

1. Relational Database: We have opted for Amazon Aurora as our relational database to store data such as customer profiles, concert event information etc. We have decided to store these data in a relational database because of the underlying relationships between them. We use Aurora as it provides built-in security including encryption, continuous backups, serverless compute, and automated multi-region replication.
2. Key value database: We have opted for DynamoDB as our key value database. We implement a key-value database to store time-sensitive data such as transaction data etc. DynamoDB provides consistent single-digit millisecond performance with limitless scalability, data encryption, and automated multi-region replication.
3. Compute: For our compute needs we have opted to use EC2 instances as our primary compute resources. We will containerize our application and deploy the containers in Kubernetes based clusters using Amazon EKS. EKS manages and auto scales our compute infrastructure.
4. Minimal latency: We have configured an AWS CloudFront instance for our Content Delivery Network requirements. In addition we implement AWS ElastiCache for caching and utilize DynamoDB for single-digit millisecond reads and writes.
5. Monitoring: We have opted to use Amazon CloudWatch for our monitoring needs. CloudWatch provides visibility to infrastructure-wide performance metrics and allows us to set alarms and react to changes.
6. Security: We utilize a range of AWS services to implement our security requirements. We configure Cloud IAM for access management, Macie to classify and protect sensitive data, Security Hub to automate security checks and aggregate alerts, and KMS to manage cryptographic keys.
7. Cost Optimization: To fulfill our cost optimization requirements we implement AWS Cost Explorer service to monitor and analyze our costs, as well as CloudWatch to further monitor and optimize usage. In addition with the implementation of auto-scaling we ensure provision of only required resources at the required time, protecting against underutilization and wastage.
8. High availability and scalability: To fulfill our high availability requirements we replicate our resources across regions and maintain database replicas as well. In addition we utilize Amazon EKS to scale our compute instances, and both of our databases are fully managed which means they can auto-scale based on load.

4.3 Action items and rough timeline

< skipped >

5 The second design

Hopefully, you came to this phase of your design after getting feedback from the instructors. Now you'll follow some structured process for creating the details of your design. Read Sections 4.3 - 4.6 in the class notes.

5.1 Use of the Well-Architected framework

Mention here the distinct steps suggested by the Well-Architected framework. See Section 4.3 in the class notes and the complete descriptions of the framework in the AWS pages for details.

Since we have chosen AWS as our cloud provider, following the Well-Architected Framework provides us with valuable guidance and best practices. By incorporating all the 6 principles of the Well-Architected Framework into the design of our cloud architecture we make sure that our architecture adheres to the highest of standards.

1. Operational Excellence: The primary objective of operational excellence is to get new features and bug fixes into customer's hands quickly and reliably. Some of the design principles associated with Operational Excellence are [3]:
    a. Perform operations as code: Utilize Infrastructure as Code tools and services to automate the process of provisioning and configuration of cloud infrastructure resources.
    b. Use managed services: By using managed services we reduce our operational burden and let AWS handle the provisioning, management and distribution of our resources.
    c. Implement observability for actionable insights: Implement continuous monitoring to gather essential metrics and perform analytics operations on them to generate actionable insights.

2. Security: The security pillar is one of the most important ones in the Well-Architected Framework. It outlines how to leverage cloud technologies and services to better secure data, resources and services. Some of the design principles used to increase security are [4]:
    a. Implement a strong identity foundation: It is recommended to centralize identity management, implement the principle of least privilege, and authorization for each interaction with AWS resources.
    b. Protect data in transit and at rest: Classify data based on sensitivity and use appropriate mechanisms such as encryption, access control etc. for each level.
    c. Apply security at all layers: Follow a defense in depth approach with multiple security controls applied to all layers such as VPC, load balancing, DNS, compute etc.

3. Reliability: The goal of the reliability pillar is to design and build architectures that are resilient, have strong foundations, and proven disaster recovery processes. This pillar is concerned with the ability of the workload to correctly and consistently perform its intended function, whenever it is expected to. Some of the design principles associated with this pillar are [5]:
    a. Automatically recover from failure: Establish constant monitoring of key metrics and automated recovery procedures whenever a threshold is breached. With sophisticated automation and monitoring it is possible to anticipate and fix failures before they even occur.
    b. Test recovery procedures: Constantly test and validate recovery procedures to make sure that our policies and procedures can resolve an actual disaster scenario. Failure scenarios can be simulated or recreated from past failures.
    c. Scale horizontally to increase aggregate workload availability: Distribute load across multiple smaller resources, instead of a single large one to ensure that there are no single points of failure.

4. Performance Efficiency: The performance efficiency pillar is concerned with the efficient use of resources, especially computing resources, to meet workload requirements and to efficiently scale performance as requirements evolve over time. The design principles associated with this pillar are [6]:
    a. Go global in minutes: Replicating resources across AWS Regions allows us to minimize latency and improve reliability.
    b. Use serverless architectures: The usage of serverless architectures eliminates the need for us to run and maintain physical servers for traditional compute workloads. This can reduce operational burden as well as reduce transaction costs since managed, serverless services operate at a cloud scale with its associated efficiency and performance.
    c. Experiment more often: With virtual, automated resources we can quickly carry out comparative testing and analysis to find and optimize the right resource for our needs.

5. Cost Optimization: Cost Optimization is another crucial pillar in the Well Architected Framework and it focuses on utilizing all available resources and achieving the outcome at the lowest possible price. It is a continuous process of refinement over the span of a workload/resource/project lifecycle. Some of the design principle associated with this pillar are [7]:
    a. Adopt a consumption model: Implement a pay as you go pricing model and only pay for the resources you consume, and scale usage accordingly.
    b. Analyze and attribute expenditure: Accurately identify the cost and usage of workloads through monitoring and cost management services. Cost management services provide a transparent look at the costs of each resource, its usage and other key metrics, thus giving us all the required information to assess the efficiency and effectiveness of resources.

6. Sustainability: Sustainability is the newest pillar in the Well Architected Framework and it focuses on addressing the long term environmental, economic and societal impact of our usage of cloud resources. Some of the design principles associated with this pillar are [8]:
    a. Maximize utilization: Provision appropriate resources, right-size workloads and implement efficient design to ensure maximum utilization and energy efficiency.
    b. Understand your impact: Monitor and measure the impact of provisioned cloud resources. Use this data to establish KPIs and evaluate approaches to improve these metrics.

5.2 Discussion of pillars

In this section we discuss 2 of the Well Architected Framework's pillars, Security and Cost Optimization.

1. Security: As previously mentioned Security is one of the most crucial pillars in the Well Architected Framework. It outlines design principles, best practices and strategies to protect data, systems and resources from all kinds of threats. In addition to the design principles mentioned above in the previous section, the Security pillar contains a few more design pillars such as Maintain Traceability, Automate Security best practices, Keep people away from data, and Prepare for Security events. These design principles fall under the 7 key areas of Security in the Cloud, which are:
    a. Security Foundations: It encompasses principles such as Shared Responsibility, Governance, and operating Workloads securely.
    b. Identity and Access Management: It focuses on the implementation of Identity Management and Permission Management. Some best practices for robust identity management are using strong sign-in mechanisms, using temporary credentials, using a centralized identity provider such as IAM, and auditing and rotating credentials periodically [9]. Best practices for Permission management include granting least privilege access, reducing permissions continuously, analyzing public and cross-account access as well as establishing emergency access processes [10].
    c. Detection: Detection consists of two main aspects, detection of unexpected or unwanted configuration changes, and the detection of unexpected behavior. Best practices involved with Detection are Configure service and application logging, analyze logs, metrics centrally, automate response to events, implement actionable security events [11].
    d. Infrastructure protection: Infrastructure protection focuses on protecting networks and protecting compute. Some best practices involved with the former are creating network layers, automating network protection and implementing inspection and protection [12]. Best practices related to protecting compute are performing vulnerability management, implementing managed services, automating compute protection, validating software integrity etc. [13]
    e. Data protection: Data protection consists of 3 main topics, Data Classification, Protecting data at rest and Protecting data in transit. Best practices associated with Data Classification are Identifying data within our workload, defining data protection controls, automating identification and classification and defining data lifecycle management [14]. Best practices associated with Protecting data at rest are implementing secure key management, enforcing encryption at rest, enforcing access control etc. [15]. Best

practices associated with Protecting data in transit are enforcing encryption in transit, automating detection of unintended data access, and authenticating network communications.

   f. Incident response: This topic focuses on response to any security incident. It includes best practices such as developing incident management plans, preparing forensic capabilities and establishing a framework for learning from incidents.
   g. Application security: Application security focuses design and security of our workloads. It involves best practices such as automating testing throughout the development and release lifecycle, performing penetration testing, and deploying software programmatically.

2. Cost optimization: Cost optimization is another core pillar of the Well Architected Framework and a core aspect of designing cloud architecture. This pillar outlines several best practices in following topic areas:
   a. Cloud Financial Management: Cloud financial management focuses on evolving existing financial processes to operate with cost transparency, control, and optimization. Some best practices associated with it are establishing ownership of cost optimization, establishing cloud budgets and forecasts, monitoring cost proactively, and creating a cost-aware culture.
   b. Monitor cost and usage: This topic focuses on a granular understanding of costs, breaking it down into core components and optimizing them. Some of its best practices are adding organization information to cost and usage, identifying cost attribution categories, establishing organizational metrics, and configuring billing and management tools.
   c. Decommissioning resources: Appropriate decommissioning of resources is a core aspect in the cost optimization process. Some of its best practices include tracking resources over their lifetime, implementing a decommissioning process, enforcing data retention policies.
   d. Right sizing: Selecting appropriate resource type, size and number helps us to ensure full utilization of provisioned resources and prevents wastage. Some best practices for this approach include performing cost modeling, choosing type, size and number based on data or metrics.
   e. Select the best pricing model: Cloud providers offer various pricing models such as on demand, commitment discounts etc. It is imperative to choose the right one based on our application requirements. Some best practices to follow for selecting the best pricing model are performing pricing model analysis, choosing regions based on cost, performing pricing model analysis at the management account level

5.3 Use of Cloudformation diagrams

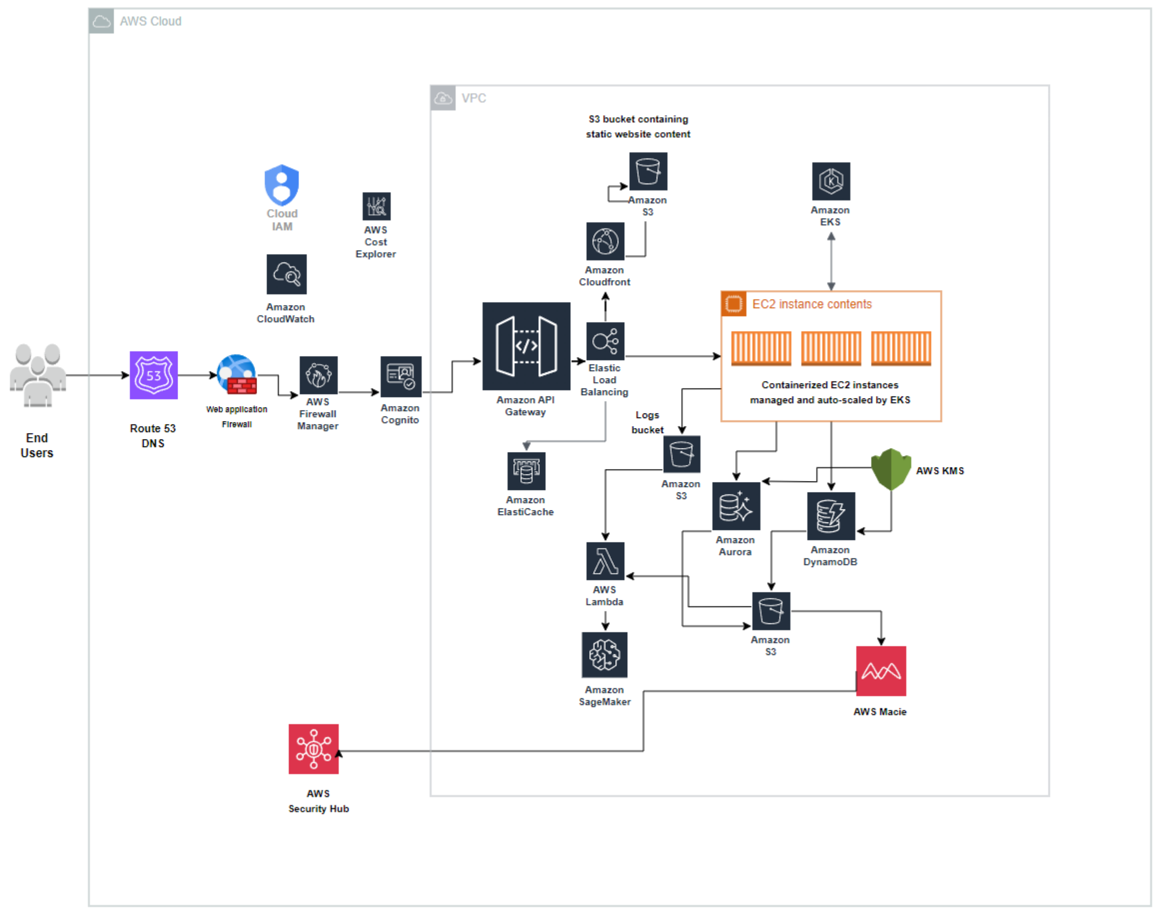


Figure 1: Overall Architecture Diagram

Our final architecture design incorporates all of the above mentioned requirements. We start with the end users trying to visit our web application. They are routed to our application through Amazon's Route 53 DNS service. This request then goes through our Web application firewall, specifically designed to help protect our application from common exploits and bots. This firewall is managed by AWS Firewall Manager. From there the request goes to Amazon Cognito for tenant identification and authorization. Once authorized the user's request is handled by the Amazon API Gateway which acts as the "front door" of our application, allowing the user to search for and book concert tickets, view their transactions etc. All of these compute tasks are handled by our primary compute resources, containerized EC2 instances managed by Amazon Elastic Kubernetes Service (EKS). We have configured an Elastic Load Balancer to help distribute the load and in addition to that EKS manages the availability and scalability of our containerized EC2 instances, thus providing us with highly available, highly scalable and efficient compute. Moreover to further reduce latency, we have implemented Amazon CloudFront for content delivery and static website assets which are stored in a S3 bucket. We have also implemented Amazon Elasticache to retrieve cached data quickly. For our database requirements we have utilized 2 different databases, Amazon Aurora, a relational database to store customer information, concert information and other relational data, as well as Amazon DynamoDB, a NoSQL key-value database used to store transaction data, state etc. We have opted to use 2 different databases because DynamoDB offers single digit millisecond performance at scale, which is something we require for transaction handling for hundreds of thousands/ millions of requests that we expect our application to receive during peak loads. However we do not always need such high performance, and especially not for all our data. Thus to reduce costs we have used Aurora as our relational database. Both databases have built in security, encryption as well as support auto-scaling and multi-Region replication. In addition we have utilized Amazon Key Management Service to manage our cryptographic keys. User data, transaction data etc. from these databases is then fed into an S3 bucket which is then provided to a Lambda function. The application logs are also provided to this Lambda function, through another S3 bucket and this Lambda function periodically utilizes Amazon SageMaker to run ML models on this data to gain actionable insights. We have also utilized Amazon Macie which monitors the S3 bucket with customer and transaction data to look for sensitive information that might be exposed. Alerts from Macie are then provided to AWS Security Hub. All of these applications/services are monitored by Amazon CloudWatch for security and optimization purposes. These resources are configured inside a VPC and we utilized IAM for centralized identity management and AWS Cost Explorer to get a better understanding of our resource usage metrics and cost metrics. All the resources inside of our VPC can be replicated across AWS Regions, with the ones outside being global across regions, to enhance availability and resiliency but due to size,clarity and readability constraints we have decided not to depict multi-region replicas in our architecture diagram. We have attached a rough diagram depicting the multi-region replicas below.

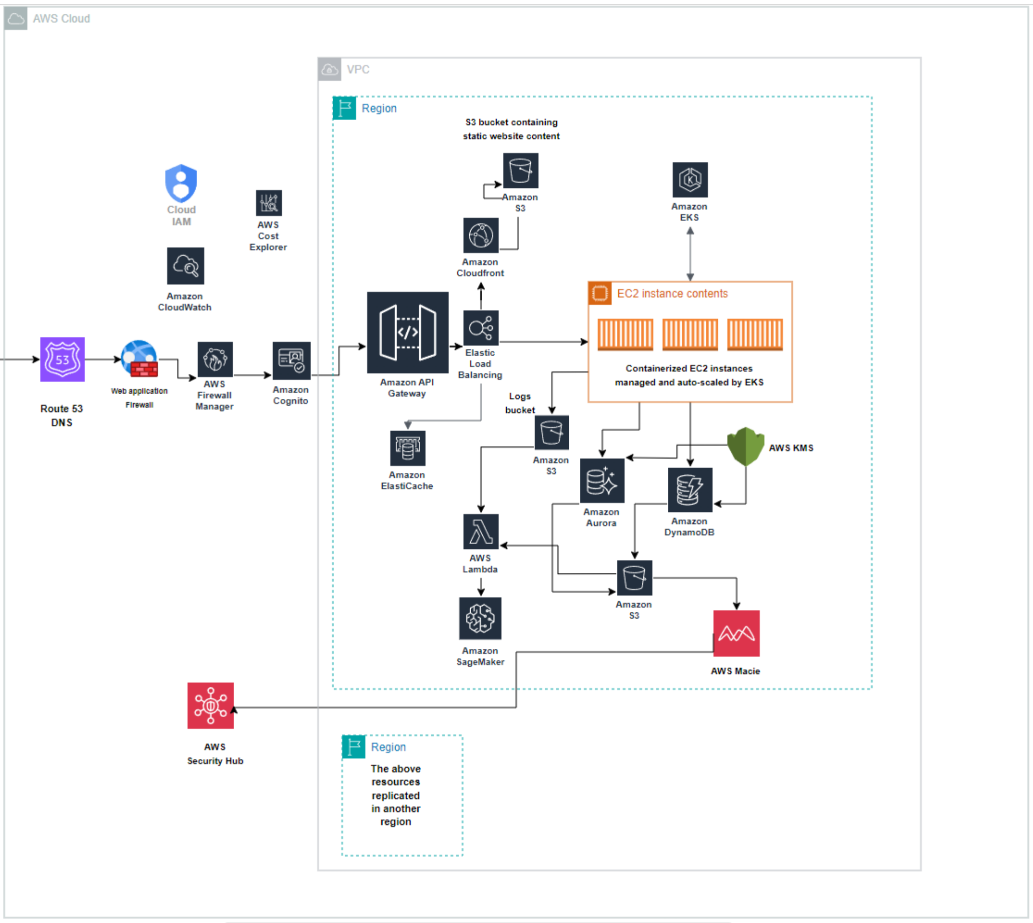


Figure 2: Architecture diagram depicting multiple regions

5.4 Validation of the design

Argue about how your design meets the BRs or TRs you listed in Section 2.

TR1.1 Our architecture implements replicas and redundancies across availability zones to ensure high availability.
TR1.2 The architecture design avoids single points of failure by maintaining multiple replicas, as well as distributing the load to multiple smaller resources than to a single larger one, as outlined by Reliability best practices in the Well Architected Framework.
TR1.3 The architecture is tested in the section below.
TR1.4 The design allows for flexibility in choosing resource types and sizes for different geographic regions.

TR2.1 The architecture implements managed services which are designed to handle disaster scenarios gracefully. In addition our compute infrastructure is managed by EKS which automatically monitors the health of our compute resources and provisions new ones if required.

TR2.2 Even though our architecture does not explicitly solve this technical requirement as it requires a multi-cloud solution, the design and principles used for this architecture can be replicated using the services of another cloud provider.

TR3.1 Our architecture implements data encryption for data while at rest and during transit, in addition to using Amazon KMS to manage cryptographic keys. Moreover we have implemented Amazon Macie to monitor our data constantly to look for exposed sensitive data, and the alerts generated from Macie are aggregated in AWS Security Hub.
TR3.2 We have implemented a Web Application Firewall to protect our application from web exploits and bot attacks. This firewall is managed by AWS Firewall Manager
TR3.3 We use AWS Identity Access Management to safeguard our data, services and applications against unauthorized access.

TR4.1 Amazon EKS manages our containerized compute infrastructure and it implements auto scaling to scale up/down the compute resources based on load. Additionally we use managed databases and Lambda functions (which are serverless), which can automatically scale up based on requirements.
TR4.2 We have utilized Elastic Load Balancing to distribute load across our compute infrastructure and across regions to maintain excellent performance and handle time varying workloads.
TR4.3 We maintain replicas of our infrastructure resources across regions to ensure high availability and lower latency.

TR5.1 We implement AWS CloudWatch for monitoring and optimizing our resource usage and performance. In addition our AWS API Gateway also feeds API calls data, throttling and monitoring data to CloudWatch as well, helping us reduce latency.
TR5.2 We implement replicas and redundancies of our resources across geographic regions to ensure high availability and location agnostic performance.
TR5.3 We have utilized AWS ElastiCache to meet our caching requirements, which further reduces latency when retrieving commonly accessed data.
TR5.4 We configure an AWS CloudFront instance to act as our Content Delivery Network, serving static web content to our users with minimal latency.

TR6.1 We have utilized Amazon EKS to handle the auto scaling of our compute infrastructure. Since we implement auto scaling, we only provision resources when the load requires it and once the period of high load has subsided, the provisioned resources are decommissioned automatically. Moreover we utilized managed databases for our storage requirements and therefore they implement auto scaling as well, scaling up and down as required.
TR6.2 We have set up a separate S3 bucket to store logs data, which is first fed into our Lambda function for analytics purposes. We can then define lifecycle rules for the data in our S3 bucket which would automatically move the data from S3 to Glacier storage.
TR6.3 We have implemented Lambda as our serverless compute and other managed services such as DynamoDB to reduce costs associated with managing and governing our services manually.
TR6.4 EKS and managed services handle the horizontal scaling of our infrastructure and through CloudWatch monitoring and Cost Explorer reports we can implement vertical scaling if the data supports it.

TR7.1 We have configured an AWS Lambda function which feeds data to AWS SageMaker to implement our analytics requirements. SageMaker is used to train and deploy Machine Learning models at scale and thus it can be used to discover actionable insights.
TR7.2 We configure AWS CloudWatch to monitor all aspects of our application and help the cloud architect resolve bottlenecks and optimize performance.
TR7.3 Logs collected from Kubernetes and EKS are stored in an S3 bucket which is then retrieved by a Lambda function to be fed into our machine learning model developed by SageMaker.

TR8.1 Implement industry standard encryption of data both while in transit and while in the database. Ensure proper usage and disposal of cookies in accordance with data protection laws. We utilize 2 different databases in our architecture, both of which support full encryption of data while at rest. In addition we make use of AWS Key Management Service to encrypt our data and manage our cryptographic keys. Further we configure Macie to constantly monitor our database to find any exposed sensitive data.
TR8.2 We can implement AWS Audit Manager whenever an audit of our infrastructure is required. At other times we opt to not use the service in order to optimize our costs, since Audit Manager continuously audits our infrastructure.

TR9.1 We configure AWS Cognito to handle our tenant authentication and authorization which.
TR9.2 In addition to tenant authentication and authorization Cognito provides an identity store that can scale to millions of users and offers advanced security features to help protect our tenants and our application. Further, insights from SageMaker can be utilized to deliver a personalized experience to the tenant/user.

TR10.1 We utilize CloudWatch and Cost Explorer as our primary monitoring and cost optimization tools which in turn help us to monitor and optimize our resource usage.
TR10.2 With the help of EKS and managed services we make sure our architecture is fully utilized to the thresholds we set while configuring these services, minimizing wastage.

5.5 Design principles and best practices used

List any specific principle or best practices you used in your design. See Sections 4.5 and 4.6 in the class notes for info.

5.5.1 Design Principles Followed

1. Use managed services: We have utilized managed services in our architecture wherever possible. We have opted for managed database services for both of our databases, Aurora and DynamoDB and we use EKS to manage our containerized compute infrastructure.
2. Implement observability for actionable insights: We have configured AWS CloudWatch to constantly monitor our infrastructure and to deliver insights that can be used to further improve

resource utilization, performance etc. In addition we employ SageMaker as well to deliver us further insights from data that can be used to further improve our application.

3. Implement a strong identity foundation: We implement a central identity management through the use of AWS Identity Access Management and implementing a principle of least privilege.
4. Protect data in transit and at rest: We encrypt our data both while in rest and in transit using AWS KMS and managed databases which support encryption.
5. Apply security at all layers: We follow a defense in depth approach by implementing security controls across layers, including but not limited to DNS, Web Application Firewall, API Gateway, VPC and encrypted databases.
6. Automatically recover from failure: We use managed services with built in automated recovery whenever possible. In addition we utilize EKS which automatically handles failures of compute instances, pods, clusters etc.
7. Scale horizontally to increase aggregate workload availability: EKS scales horizontally by increasing replicas based on load, which increases aggregate workload availability.
8. Go global in minutes: With the ability to replicate our infrastructure resources across regions we can reliably serve customers across the globe with minimal latency.
9. Adopt a consumption model: With the utilization of AWS resources, auto scaling and load balancing we only pay for resources that we utilize.
10. Analyze and attribute expenditure: With the help of CloudWatch and Cost Explorer we can analyze our spending, resource utilization and efficiency of resources with respect to its cost.
11. Maximize utilization: With the use of auto-scaling of our compute resources, databases and with the use of an Elastic Load Balancer we ensure maximum utilization of our provisioned resources.

5.5.2 Best Practices Followed

1. Identity Access Management: We implement stringent Identity management through the use of AWS IAM.
2. Granting least privilege access: While granting access to resources we make sure to follow the principles of least privilege access.
3. Analyze logs: We have implemented a Lambda function in conjunction with AWS SageMaker to analyze our application logs.
4. Implementing managed services: With our implementation of managed services throughout the architecture we ensure adequate security, scalability and performance.
5. Key Management: We utilize AWS Key Management Service to manage our cryptographic keys.
6. Enforcing encryption at rest: We enforce the encryption of our data at rest with the help of our databases, Aurora and DynamoDB, both of which support full encryption of the data.
7. Monitoring cost proactively: We utilize AWS Cost Explorer to routinely monitor our cost and take steps proactively to minimize it.
8. Decommissioning resources: EKS decommissions compute resources automatically after a period of time of under-utilization.

9. Use a data-driven approach for architectural choices: We have made our architecture design choices based on data from similar use cases, sample architectures and by following the principles and best practices outlined in the Well Architected Framework.

5.6 Tradeoffs revisited

Provide more details on the tradeoffs you made so far[3]. Be as exhaustive as you wish!

In this section we revisit the tradeoffs we made in section 2.4 with the insights and guidelines from reference [16] “Even Swaps: A Rational Method for making Trade-offs” in mind. Based on it our new tradeoffs and the rationale behind our decision making are listed below:

1. Single region vs Multi region deployment: Deploying our infrastructure across regions enhances availability and minimizes latency for users but it introduces additional complexity, costs and management and synchronization overhead. However guaranteeing high availability and reliability is one of the core requirements of our application and therefore we have opted for multi-region deployment.
2. Use of Web Application Firewall: Implementing the Web Application Firewall introduces some latency in the processing of the application request as opposed to not having a firewall at all. But in order to protect our application from outside threats and bot attacks we need to implement a firewall and therefore we have decided to implement the Web Application Firewall.
3. Use of ElastiCache: The implementation of ElastiCache to fulfill our caching requirements introduces complexity with regards to managing caching resources, consistency and costs. But the trade-off is made to ensure better performance of our application with the understanding that this decision could introduce cost and complexity overhead.
4. Use of containerized EC2 instances as compute vs Serverless compute: While serverless compute offers many advantages such as cheaper costs, built-in scalability, and reduced complexity, we have opted to use containerized EC2 instances due to the following reasons.
    a. By setting up containerized EC2 instances managed by EKS we let EKS handle the scaling, monitoring and provisioning of our EC2 instances, reducing our operational overhead.
    b. Further, EC2 instances are provisioned 24x7 and as a result they do not suffer from cold booting. Serverless however only provisions the resources at the time of execution, thus introducing some latency before the request can be served. Since minimal latency is one of the core requirements of our application, we have opted for EC2 instances.

    But we do utilize serverless compute, in the form of AWS Lambda, in our architecture. It is primarily used to handle event-driven/periodical tasks such as retrieving logs or running a SageMaker workload.
5. Data monitoring and classification using Macie: The utilization of Macie to monitor our databases and look for exposed sensitive data introduces additional costs and operational overhead. However the classification and protection of sensitive data is of utmost priority to our application, as well as a part of legal and compliance requirements. Therefore we have made the decision to use Macie.

6. A common source of conflicts and tradeoffs in the cloud come from added overhead and complexity of implementing robust security measures, and the performance penalties and costs that such measures bring with them. In our case as previously mentioned, we use a Web Application Firewall to protect our application. In addition we also implement encryption of data while in transit, configure Macie to monitor our databases and set up AWS Security Hub to aggregate all security alerts. All of these operations would naturally introduce some latency in performance and add to our costs but these measures are required to safeguard our application and data.

5.7 Discussion of an alternate design

For at least one of the objectives, present an alternate design; discuss why you did not pursue

it further. SKIPPED

6 Kubernetes experimentation

In this section, you'll validate some aspects of your design experimentally. Which ones is your choice, but you must discuss with the instructors first. The idea is to limit the scope of the experiment runs and required analysis[4].

6.1 Experiment Design

List the TR(s) you chose to work with. Describe in detail the experiment that will validate your design. More specifically, define the configuration of the environment, the inputs and the expected outputs.

We have chosen to validate TR4.1: Monitor workloads and automatically scale up or down to meet capacity requirements, and TR7.3: Collect logs of compute instances and store them for analytics purposes.. To validate TR4.1 we varied the workload to the Kubernetes cluster using a load generator and observed the HPA behavior. To validate TR7.3 we collect kubectl logs, which will be dumped into a S3 bucket, to be later used for analysis by SageMaker.

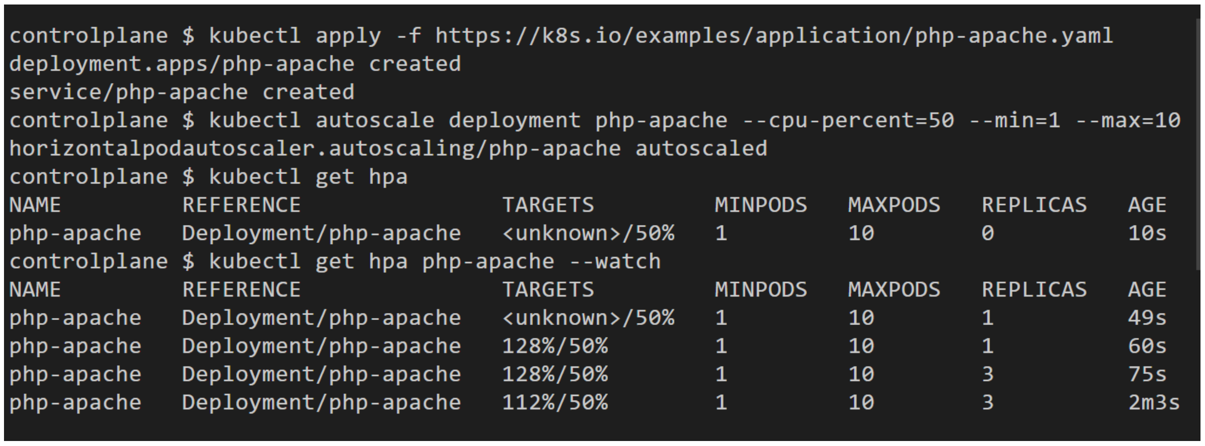
We have created our experiment environment by using the below steps:

```
kubectl apply -f https://k8s.io/examples/application/php-apache.yaml
```

This deploys the PHP-Apache server.

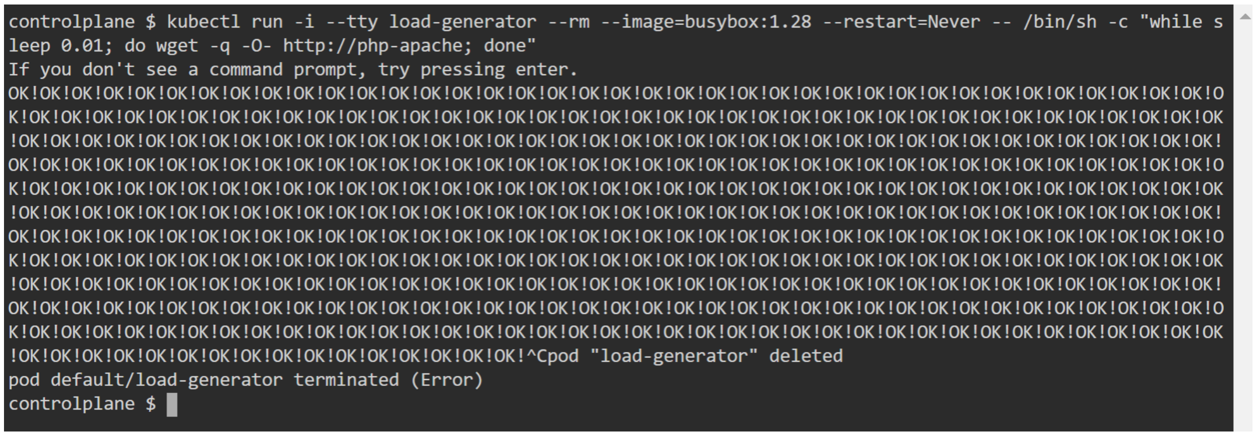
```
kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10
```

This creates the HPA that will maintain between 1 and 10 replicas with a target CPU utilization of 50%



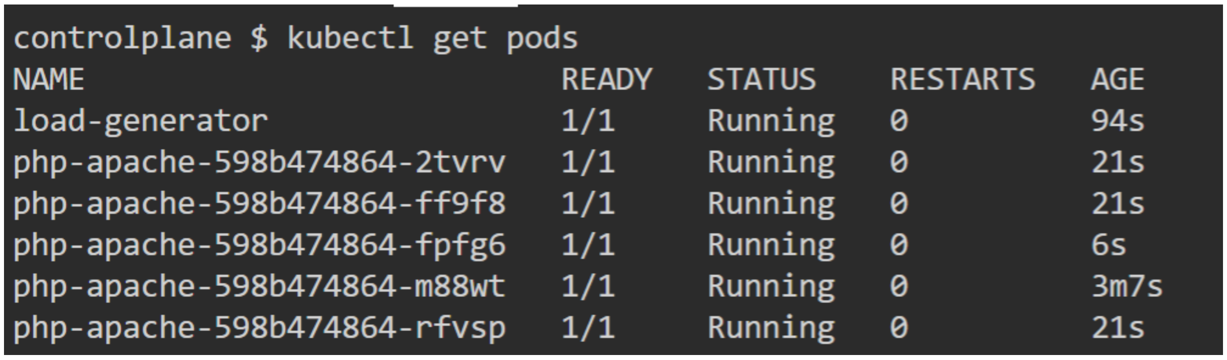
```
controlplane $ kubectl apply -f https://k8s.io/examples/application/php-apache.yaml
deployment.apps/php-apache created
service/php-apache created
controlplane $ kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10
horizontalpodautoscaler.autoscaling/php-apache autoscaled
controlplane $ kubectl get hpa
NAME         REFERENCE               TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   <unknown>/50%   1         10        0          10s
controlplane $ kubectl get hpa php-apache --watch
NAME         REFERENCE               TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   <unknown>/50%   1         10        1          49s
php-apache   Deployment/php-apache   128%/50%        1         10        1          60s
php-apache   Deployment/php-apache   128%/50%        1         10        3          75s
php-apache   Deployment/php-apache   112%/50%        1         10        3          2m3s
```

To generate load:
kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://php-apache; done"



```
controlplane $ kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never -- /bin/sh -c "while s
leep 0.01; do wget -q -O- http://php-apache; done"
If you don't see a command prompt, try pressing enter.
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!O
K!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK
!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!O
K!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK
!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!O
K!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK
!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!O
K!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK
!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!^Cpod "load-generator" deleted
pod default/load-generator terminated (Error)
controlplane $
```

To get the details of active pods we use the command:
kubectl get pods



```
controlplane $ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
load-generator                1/1     Running   0          94s
php-apache-598b474864-2tvrv   1/1     Running   0          21s
php-apache-598b474864-ff9f8   1/1     Running   0          21s
php-apache-598b474864-fpfg6   1/1     Running   0          6s
php-apache-598b474864-m88wt   1/1     Running   0          3m7s
php-apache-598b474864-rfvsp   1/1     Running   0          21s
```

6.2 Workload generation with Locust

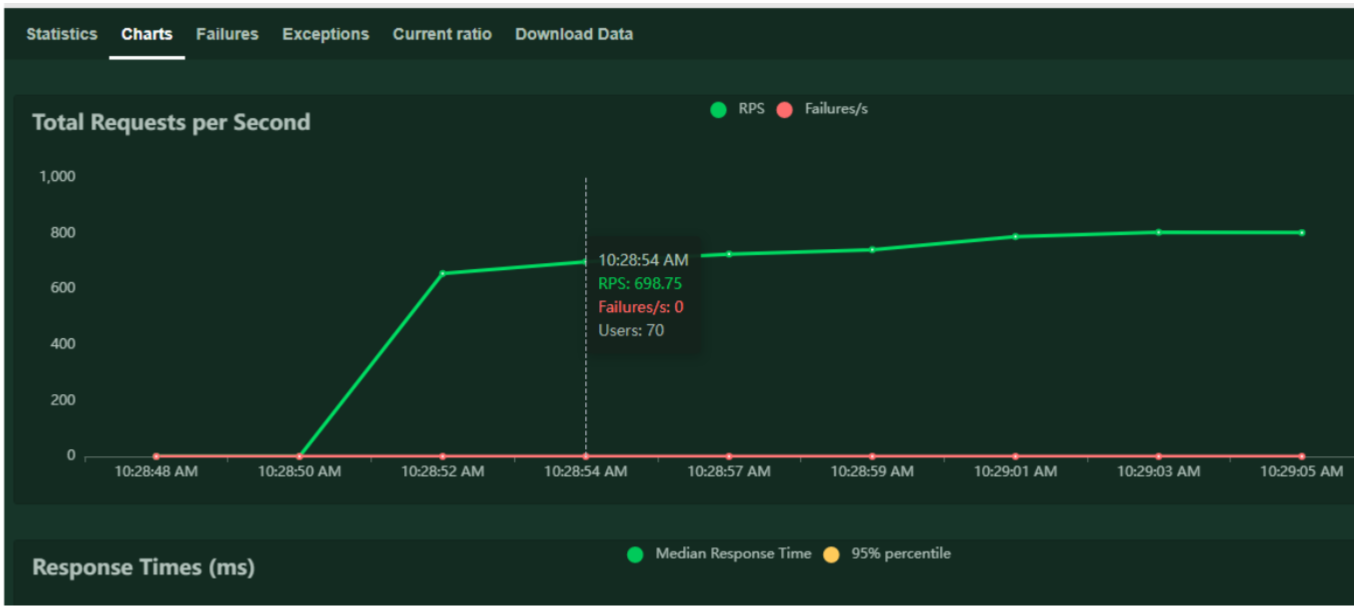
Describe how you intend to use Locust to create (all or some of) the inputs.

Workload generation with Locust is used to verify the behavior of the sample application under different workloads. A Locust user class called MyUser is defined in the workload.py file, and it configures a wait time of one to three seconds for each simulated user. The class's main task, compute_task(), makes HTTP GET calls to the sample application's /compute endpoint[17]. This Locust configuration makes it easier to simulate user interactions and makes it possible to systematically observe how the application performs with varying user loads. Locust generates different user loads, which are used as inputs for the experimental validation of the scalability of the application and the system's adaptability to workload fluctuations in accordance with TR4.1 (Source: ChatGPT)

We have used Locust for load testing our application and the experiment is as follows:

- Experiment: 750 users with a spawn rate of 100.

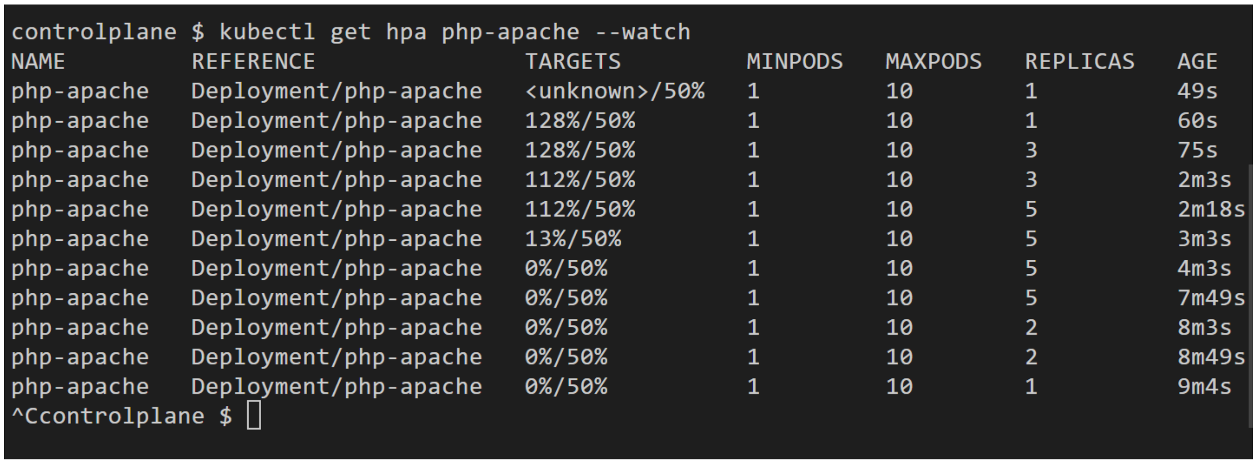
We have simulated our application with the above scenario and we see that there is a sharp rise initially in the number of requests because of the newly spawning requests and then the number of requests reaches a steady rate. This also indicates that the response time is in the order of milliseconds, which is expected from an application that is time-bound.



6.3 Analysis of the results

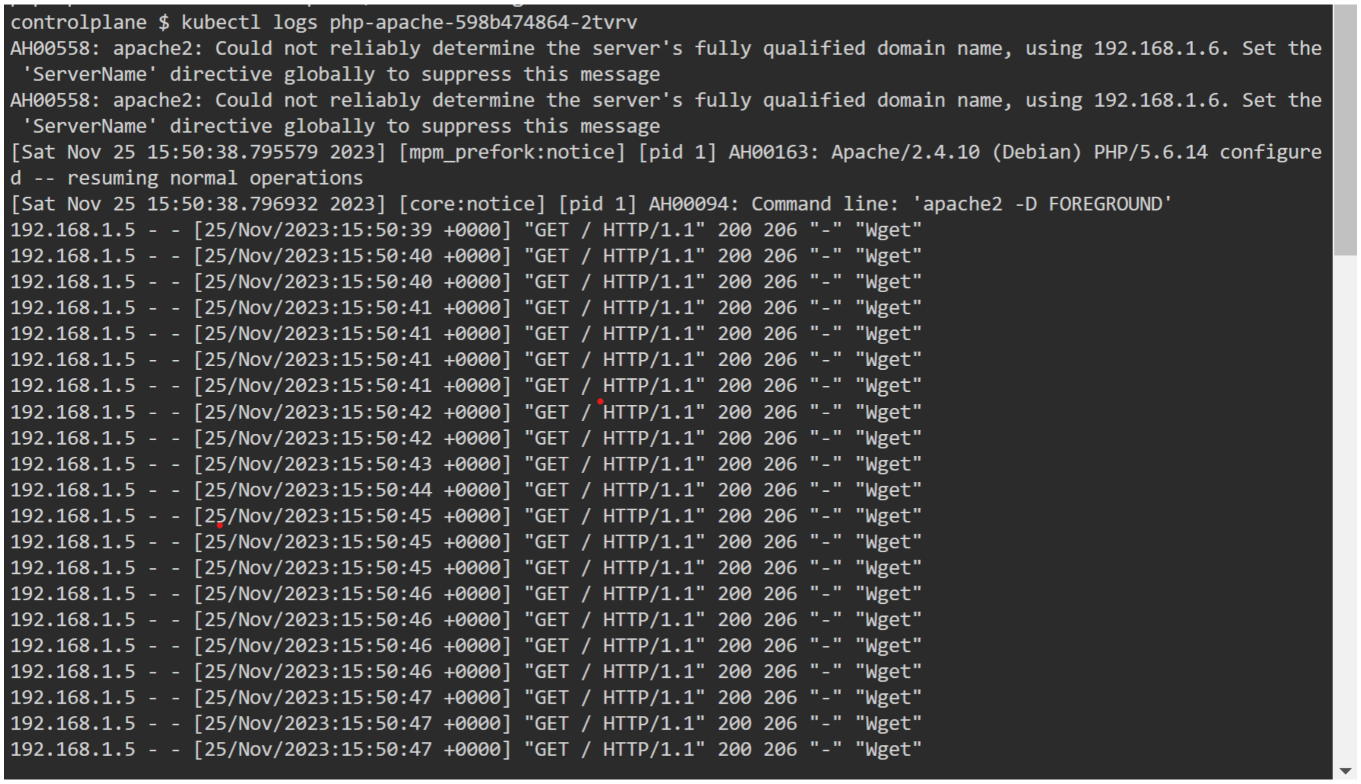
Analyze the outputs you observed. Why do they validate the design?

To validate TR4.1 we configured a Kubernetes cluster and implemented HPA to auto scale our cluster and create/delete replicas as per load and CPU utilization.
We begin monitoring the HPA with the command:
kubectl get hpa php-apache --watch



```
controlplane $ kubectl get hpa php-apache --watch
NAME         REFERENCE               TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   <unknown>/50%   1         10        1          49s
php-apache   Deployment/php-apache   128%/50%        1         10        1          60s
php-apache   Deployment/php-apache   128%/50%        1         10        3          75s
php-apache   Deployment/php-apache   112%/50%        1         10        3          2m3s
php-apache   Deployment/php-apache   112%/50%        1         10        5          2m18s
php-apache   Deployment/php-apache   13%/50%         1         10        5          3m3s
php-apache   Deployment/php-apache   0%/50%          1         10        5          4m3s
php-apache   Deployment/php-apache   0%/50%          1         10        5          7m49s
php-apache   Deployment/php-apache   0%/50%          1         10        2          8m3s
php-apache   Deployment/php-apache   0%/50%          1         10        2          8m49s
php-apache   Deployment/php-apache   0%/50%          1         10        1          9m4s
^Ccontrolplane $ 
```

We notice once load generation begins CPU utilization increases to well above target level, which is when HPA begins auto-scaling and deploying replicas to handle the extra load. We can notice that the number of replicas increases gradually until CPU utilization is within target levels and then it stabilizes. Once load generation is stopped we notice that the number of replicas does not immediately go down to 1 but instead gradually decreases and reaches 1 eventually. This gradual increase and decrease in replicas demonstrates graceful auto-scaling by HPA, which ensures we do not overprovision resources for a short term bursty load and also we don’t decommission our resources as soon as there is a temporary break in the load. Since we utilize containerized EC2 instances, managed by EKS as our compute infrastructure, this experiment validates that EKS can reliably scale our compute infrastructure based on load.



```
controlplane $ kubectl logs php-apache-598b474864-2tvrv
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 192.168.1.6. Set the
 'ServerName' directive globally to suppress this message
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 192.168.1.6. Set the
 'ServerName' directive globally to suppress this message
[Sat Nov 25 15:50:38.795579 2023] [mpm_prefork:notice] [pid 1] AH00163: Apache/2.4.10 (Debian) PHP/5.6.14 configure
d -- resuming normal operations
[Sat Nov 25 15:50:38.796932 2023] [core:notice] [pid 1] AH00094: Command line: 'apache2 -D FOREGROUND'
192.168.1.5 - - [25/Nov/2023:15:50:39 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:40 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:40 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:41 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:41 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:41 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:41 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:42 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:42 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:43 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:44 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:45 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:45 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:45 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:46 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:46 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:46 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:46 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:47 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:47 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
192.168.1.5 - - [25/Nov/2023:15:50:47 +0000] "GET / HTTP/1.1" 200 206 "-" "Wget"
```

From the above image we can see logs generated for each individual pod, thus providing us with higher granular data in addition to the CloudWatch logs. This data can be used by a machine learning model to discover insights and optimize our application, which we have implemented using Amazon SageMaker.

7 Ansible playbooks [0%]

In this section, you'll define specific management tasks and write Ansible playbooks to automate them.

7.1 Description of management tasks

List the specific tasks. Describe the difficulty involved in executing these tasks

manually. SKIPPED

7.2 Playbook Design

Write the playbooks for executing the tasks automatically. Comment on the ease/difficulty of producing the playbooks.

SKIPPED

7.3 Experiment runs

Run the playbooks. Supply verification that the tasks were executed properly.

SKIPPED

8 Demonstration [0%]

A demo is not required for this project. However, if you want to use any of the (AWS, Azure, IBM, or Google) cloud platforms, you are welcome to do so. Your demo should highlight aspects of your architectural design.

The demo should focus on a few specific TRs. Discuss with the instructors which TRs to use.
SKIPPED

9 Comparisons [0%]

SKIPPED

In this section, you will provide a comparison between two solutions, approaches, tools, current trends, etc. The comparison can be theoretical or experimental.

Here are some suggestions:

1. Facebook's Katran and another LB algorithm
2. AKS, GKS, EKS and Openshift
3. VxLAN, Geneve, NVGRE virtualization protocols
4. RFC8926: Amazon- and other cloud vendor-specific control channels
5. CloudFormation and another IaC implementation
6. Observability (trend)
7. Microsoft Azure's Architectural Framework vs AWS's vs Google's

Framework 8. Students' selected topic

9. Others TBD in due time

10 Conclusion

10.1 The lessons learned

Describe, in detail, your experience with this project. What was difficult? interesting? boring? unexpected? etc.

Through this project we gained valuable insights and experiences of working through an architecture design problem from the perspective of a cloud architect. We learnt to come up with solid Business requirements and their corresponding Technical requirements as well as justifying how our choice of technical requirements solve their specific business requirements.

Comparing cloud providers and their service offerings provided us with a small glimpse into the plethora of services/solutions offered for each problem. Sorting through them all, defining our criteria for choosing a provider, comparing the services and finally making a choice proved to be a bit overwhelming but we managed to overcome this challenge.

After this was the solidifying of the design and creating the architecture diagram, which were the most enjoyable parts of the project. It was nice to see all the hard work done in the earlier sections come in handy, with the requirements and building blocks being finalized, the creation of the architecture turned out to be quite easier.

After this however was perhaps the most challenging part of the project, running the Kubernetes experiment to validate our design choices. Having almost no prior experience with Kubernetes and Locust, this section proved to be quite tricky. However with the help of lab sessions and plenty of research we managed to run our experiments and validate our design choices.

Overall this project provided us with a great hands on experience of designing a cloud architecture, as well as reiterated the importance of teamwork and collaboration to solve a problem.

10.2 Possible continuation of the project

SKIPPED

If you plan to take the advanced course during the spring 2023 semester, provide here some ideas on how to continue the project. Ditto if you are interested in further research/publications.

11 References

Include here your references.

We acknowledge that we have not copied from any other sources than the ones mentioned below.

1. https://en.wikipedia.org/wiki/Taylor_Swift%E2%80%93Ticketmaster_controversy
2. https://docs.aws.amazon.com/wellarchitected/latest/sustainability-pillar/design-principles-for-sustainability-in-the-cloud.html
3. https://docs.aws.amazon.com/wellarchitected/latest/operational-excellence-pillar/operational-excellence.html
4. https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/security.html
5. https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/design-principles.html
6. https://docs.aws.amazon.com/wellarchitected/latest/performance-efficiency-pillar/design-principles.html
7. https://docs.aws.amazon.com/wellarchitected/latest/performance-efficiency-pillar/design-principles.html
8. https://docs.aws.amazon.com/wellarchitected/latest/sustainability-pillar/design-principles-for-sustainability-in-the-cloud.html
9. https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/identity-management.html
10. https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/permissions-management.html
11. https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/detection.html
12. https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/protecting-networks.html
13. https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/protecting-compute.html
14. https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/data-classification.html
15. https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/protecting-data-at-rest.html
16. https://hbr.org/1998/03/even-swaps-a-rational-method-for-making-trade-offs
17. ChatGPT