Cloud Architecture Project

FALL ‘23
CSC 547

Presented by:
Jay Vishaal Janarthanan
Harini Ravi

Student	ID	Unity ID
Jay Vishaal Janarthanan	200537782	JJANART
Harini Ravi	200537960	HRAVI2

TABLE OF CONTENTS

1 Introduction

1.1 Motivation

The motivation behind designing the cloud architecture for a video streaming and live streaming software project is to address the growing demand for high-quality, reliable, and engaging content delivery. We believe that cloud computing is the ideal platform for this task, as it offers several advantages over traditional on-premises solutions. Here are the key motivations for undertaking this project:

1. **Meeting User Demand:** The project aims to meet the ever-increasing user demand for video streaming services, particularly during live events, offering users a seamless and enjoyable viewing experience.
2. **Global Reach:** By designing a scalable and geographically distributed architecture, the project can reach a global audience, delivering content to viewers worldwide.
3. **Enhancing User Experience:** Improving the user experience by providing adaptive streaming, low latency, and high-quality video playback is a fundamental motivation. Users should be able to enjoy content without interruptions or buffering.
4. **Security and Content Protection:** Safeguarding the rights of content creators and preventing piracy is a critical motivation. The architecture must include robust security measures to protect content from unauthorized access and distribution.
5. **Revenue Generation:** Enabling monetization options, such as pay-per-view and advertising, can lead to revenue generation, which is essential for the sustainability of the streaming service.
6. **Data-Driven Insights:** Collecting user data and analytics is another motivation. This data helps in making informed decisions, enhancing content recommendations, and understanding user behavior.
7. **Compliance and Legal Requirements:** Ensuring compliance with content licensing agreements, copyright laws, and regional regulations is crucial to avoid legal complications.
8. **Competitive Edge:** Building a cutting-edge streaming service that can compete with industry leaders motivates the project. The aim is to provide features and capabilities that set the service apart.
9. **Engagement and Interactivity:** The motivation includes engaging viewers during live events with features like live chat, real-time interactions, and audience engagement tools.
10. **Scalability and Reliability:** Creating a highly scalable and reliable architecture ensures that the service can handle traffic spikes during live events without interruptions or downtime.
11. **Innovation:** Innovating in the streaming industry by adopting the latest technologies and trends motivates the project. Staying at the forefront of technology is essential.

12. **Adaptation to New Norms:** Especially relevant in the context of the global situation, the project aims to adapt to changing norms, where more content and events are being delivered via live streaming.

In summary, the project's motivation is to deliver a world-class video streaming and live streaming service that caters to the needs and expectations of users, content creators, and the industry as a whole. The architecture design is a critical component in achieving these goals.

1.2 Executive summary

This project entails the creation of a cloud architecture for a video streaming and live streaming service, inspired by platforms like Netflix and Disney Plus Hotstar. The core objective is to provide a highly responsive and reliable content delivery system, optimizing user experience through adaptive streaming, low latency, and data-driven personalization. The project aspires to meet the global demand, maintain the security and integrity of digital content, comply with legal requirements, and generate revenue through various monetization models. Innovating with new features, ensuring scalability and reliability, fostering audience engagement during live events, and staying technologically up-to-date are central tenets of the project.

2 Problem Description

2.1 The problem

As the principal cloud architect of an online video streaming service , we must design the complete cloud architecture to deploy our application. Our application is similar to Disney+ Hotstar.

2.2 Business Requirements

1. Ensure high availability to minimize downtime and disruptions.
2. Implement error-free tenant identification.
3. Monitor individual tenant usage and prepare a plan for resource allocation appropriately.
4. Implement tenant isolation to protect their data and privacy.
5. Notifications must be sent in the event of any changes or failures.
6. Enable easy switch of domain names and also automatic reroute on domain changes.
7. Implement incident reporting mechanisms to record breaches or privacy incidents.
8. Optimize infrastructure costs.
9. Choose providers with environmental certifications.
10. Design for rapid scalability to accommodate fluctuating user demand.
11. Implement a disaster recovery plan to ensure data resilience.
12. Manage and store large volumes of multimedia content efficiently.
13. User data should be backed up reliably, securely and efficiently.
14. Interoperability must be accommodated keeping in mind changing needs.
15. Billing information should be manageable.
16. Consider well-reputed vendors.
17. Avoid vendor lock-in.
18. Optimize data redundancy.
19. Optimize content delivery for global scalability and low latency.
20. Ensure Simple Network Management

Conflicting BRs:

1. **BR8** and **BR16** might cause conflict while choosing the vendor as well reputed vendors (like the big-three for example) might be more expensive than less reputed ones. However, choosing a less reputed vendor to cut back on costs might come at the expense of quality of service.
2. **BR13** and **BR18** might be conflicting while designing a data backup plan because a high level of redundancy is required to ensure reliability of backed up data. For example, to achieve “reliable” backup, we may need to save multiple copies of the same data in various data centers in geographically dispersed locations, which might not necessarily be the optimal solution to storing data.
3. **BR14** and **BR17** might conflict with each other as some vendors (who provide exactly what is required) might not provide services which can be used along with those

provided by other vendors, which might force us to select the services provided by the same vendor.

2.3 Technical Requirements

BR1: Ensure high availability to minimize downtime and disruptions.
TR1.1: The architecture must be a redundant one with load balancing to ensure 99% availability.
TR1.2: Auto-scaling must be implemented to handle increased loads automatically.
TR1.3: Utilize at least 3 data centers in geographically distributed regions for failover capabilities.
TR1.4: Redundant network connections must be implemented to prevent connectivity disruptions.

BR2: Implement error-free tenant identification.
TR2.1: A secure authentication mechanism must identify tenants with 100% accuracy.
TR2.2: Identifiers and session keys may be used to monitor individual tenants' usage.
TR2.3: Role-based access control (RBAC) must be used to manage tenant privileges accurately.

BR3: Monitor individual tenant usage and prepare a plan for resource allocation appropriately.
TR3.1: Monitoring tools must be used to track tenant resource utilization.
TR3.2: Predictive analytics must be employed to forecast resource needs based on historical usage data.

BR4: Implement tenant isolation to protect their data and privacy.
TR4.1: Utilize Virtual Private Cloud (VPC) for tenant isolation.
TR4.2: Employ network segmentation to ensure tenant data privacy.
TR4.3: Employ tenant-specific authentication to protect tenant data and privacy.

BR5: Notifications must be sent in the event of any changes or failures.
TR5.1: Implement a notification system for real-time alerts on system changes and failures.
TR5.2: Configure email and SMS notifications for timely communication.
TR5.3: Configure alerting thresholds for specific changes and failures.
TR5.4: Notification system must be automated to alert on pricing changes.

BR6: Enable easy switch of domain names and also automatic reroute on domain changes.
TR6.1: Implement a domain name management system for seamless switching.
TR6.2: Set up automatic DNS rerouting for domain changes to minimize downtime.
TR6.3: Implement URL redirection policies to reroute traffic during domain changes.

BR7: Implement incident reporting mechanisms to record breaches or privacy incidents.
TR7.1: Establish a centralized logging system to support traceability in the event of data breach.

TR7.2: Define procedures for incident documentation, reporting, and resolution.
TR7.3: Ensure incident response procedures are compliant with legal regulations.
TR7.4: Automate Security policy updates in order to prevent any manual errors.

BR8: Optimize infrastructure costs.
TR8.1: Employ resource tagging for tracking resource costs and allocation.
TR8.2: Use cloud cost optimization tools to identify and manage resource waste.
TR8.3: Select vendors charging lesser prices for higher rated/more popular services.
TR8.4: Constantly keep checking and updating the services for cost optimization possibilities.

BR9: Choose providers with environmental certifications.
TR9.1: Select cloud providers with recognized environmental certifications (like ISO 14001).
TR9.2: Ensure that data centers adhere to green computing practices.
TR9.3: Choose data centers with optimal climatic conditions to minimize cooling efforts.
TR9.4: Choose data centers with access to green energy resources to minimize electricity usage.

BR10: Design for rapid scalability to accommodate fluctuating user demand.
TR10.1: Implement auto-scaling based on user traffic patterns.
TR10.2: Utilize serverless architecture to dynamically scale resources as needed.

BR11: Implement a disaster recovery plan to ensure data resilience.
TR11.1: Create secure data backups and store them in geographically diverse locations.
TR11.2: Develop and test a disaster recovery plan with a defined RTO (Recovery Time Objective).
TR11.3: Ensure that data retention policies are in accordance with legal requirements, business needs and usage patterns.
TR11.4: Disaster recovery drills must be conducted at least once a year.
TR11.5: Data recovery must be automated in the event of a failure.

BR12: Manage and store large volumes of multimedia content efficiently.
TR12.1: Utilize content delivery networks (CDNs) for efficient content distribution.
TR12.2: Implement deduplication techniques for storage optimization.
TR12.3: Implement content compression techniques to optimize storage capacity.

BR13: User data should be stored and backed up reliably, securely and efficiently.
TR13.1: (TR11.1) Create secure data backups and store them in geographically diverse locations.
TR13.2: Implement data lifecycle management to optimize storage costs.
TR13.3: Implement encryption for data at rest and in transit to ensure data security.
TR13.4: Implement authentication systems to prevent unauthorized access to backup data.

BR14: Interoperability must be accommodated keeping in mind changing needs.
TR14.1: Implement open standards and APIs for better interoperability.

TR14.2: Establish integration frameworks for connecting with external systems.

BR15: Billing information should be manageable.
TR15.1: There must be a billing and invoicing system with detailed reporting of resource and service utilization.
TR15.2: Ensure set procedures to settle billing disputes.

BR16: Consider well-reputed vendors.
TR16.1: Conduct vendor assessments based on industry reputation and reviews.
TR16.2: Verify vendors' compliance with industry standards and certifications.
TR16.3: Consider vendors who are compliant with the security guidelines provided by NIST.

BR17: Avoid vendor lock-in.
TR17.1: Develop a multi-cloud strategy to prevent dependency on a single vendor.
TR17.2: Utilize containerization and orchestration to enable workload portability.
TR17.3: Develop a cloud-agnostic strategy for porting applications between different cloud providers.
TR17.4: Leverage container orchestration platforms for vendor-agnostic workload management.

BR18: Optimize data redundancy.
TR18.1: Create only the minimum required replicas of the data to achieve optimal redundancy.
TR18.2: Employ data deduplication and compression techniques to reduce storage overhead.

BR19: Optimize content delivery for global scalability and low latency.
TR19.1: Employ content distribution networks (CDNs) to cache content closer to end-users.
TR19.2: Implement edge computing for reduced latency in remote regions.
TR19.3: Implement global load balancing for directing user requests to the nearest available servers.

BR20: Ensure Simple Network Management
TR 20.1: Implement automated simple network provisioning and configuration tools that simplify the process of adding new network devices or making changes to existing configurations.
TR 20.2 : Implement Network Management Tools with User-Friendly Interface
TR 20.3 : Enable remote network management capabilities, allowing administrators to oversee and make changes to the network from any location.

TR - BR mapping with Justification

BR	TR	Justification
BR 1 **Ensure high availability to**	TR 1.1 The architecture must be a redundant	Load balancing helps distribute incoming network traffic across multiple servers to ensure no single server becomes overwhelmed, thereby enhancing system stability and availability. Achieving 99%

minimize downtime and disruptions.	one with load balancing to ensure 99% availability.	availability suggests that the system would experience minimal downtime.
	TR 1.2 Auto-scaling must be implemented to handle increased loads automatically.	Auto-scaling allows the system to automatically adjust its capacity based on demand. During periods of increased load, thereby maintaining performance levels and preventing disruptions or slowdowns.
	TR 1.3 Utilize at least 3 data centers in geographically distributed regions for failover capabilities.	Utilizing at least three geographically distributed data centers ensures geographical redundancy. If one data center experiences an outage due to natural disasters, technical issues, or any other unforeseen circumstances, the system can failover to another data center, ensuring continuity of service.
	TR 1.4 Redundant network connections must be implemented to prevent connectivity disruptions.	Implementing redundant network connections helps ensure continuous connectivity. Should one connection fail, redundant connections act as backups, maintaining uninterrupted communication between servers, data centers, and users.
BR 2 **Implement error-free tenant identification.**	TR 2.1 A secure authentication mechanism must identify tenants with 100% accuracy.	By achieving 100% accuracy, the system fulfills the business requirement of error-free tenant identification, ensuring that only authenticated and verified tenants can access the resources or services.
	TR 2.2 Identifiers and session keys may be used to monitor individual tenants' usage.	Using identifiers and session keys enables the monitoring of individual tenants' usage within the system. These identifiers and keys allow tracking and recording the activities of each tenant separately, aiding in the accurate identification and monitoring of their usage patterns. This facilitates error detection and helps ensure that each tenant's activities are correctly attributed, contributing to

		error-free tenant identification as specified in the business requirement.
	TR 2.3 Role-based access control (RBAC) must be used to manage tenant privileges accurately.	RBAC is a system that restricts system access based on the roles of individual users. Implementing RBAC ensures that tenant privileges are accurately managed based on their roles or permissions within the system. By assigning specific roles and access rights to different tenants, the system ensures that each tenant has precisely the privileges they require and are authorized to access.
BR 3 **Monitor individual tenant usage and prepare a plan for resource allocation appropriately.**	TR 3.1 Monitoring tools must be used to track tenant resource utilization.	This requirement mandates the use of monitoring tools to track and analyze individual tenant resource utilization. Implementing such tools allows continuous monitoring of various resources consumed by each tenant within the system.
	TR 3.2 Predictive analytics must be employed to forecast resource needs based on historical usage data.	By leveraging predictive analytics techniques, the system can analyze historical usage data collected by the monitoring tools (as specified in TR3.1). Predictive analytics algorithms can extrapolate trends and patterns from this data to forecast future resource needs accurately. By understanding past usage patterns and behavior, the system can predict and plan resource requirements more effectively. This proactive approach helps in preparing an appropriate resource allocation plan, ensuring that resources are available when needed without over-provisioning or under-provisioning.
BR 4 **Implement tenant isolation to protect their data and privacy.**	TR 4.1 Utilize Virtual Private Cloud (VPC) for tenant isolation.	VPC provides a logically isolated section of the cloud dedicated to specific tenants or groups of tenants. By utilizing VPCs, each tenant's resources and data are isolated from others, ensuring that they operate within their own private network environment. This isolation helps prevent unauthorized access and data breaches between tenants, meeting the requirement of protecting tenant data and privacy.

	TR 4.2 Employ network segmentation to ensure tenant data privacy.	Network segmentation involves dividing a network into smaller segments to enhance security and control access. Implementing network segmentation ensures that tenant data remains within designated segments, allowing for controlled access and preventing unauthorized communication between different segments. This strategy adds an extra layer of security, reinforcing the protection of tenant data and privacy as mandated by the business requirement.
	TR 4.3 Employ tenant-specific authentication to protect tenant data and privacy.	Implementing tenant-specific authentication mechanisms ensures that only authorized users from each tenant can access their respective data. By requiring separate authentication credentials for each tenant, the system effectively protects tenant data and privacy. This approach prevents unauthorized access from other tenants and strengthens data security by enforcing individualized authentication measures for each tenant.
BR 5 **Notifications must be sent in the event of any changes or failures.**	TR 5.1 Implement a notification system for real-time alerts on system changes and failures.	This requirement ensures the establishment of a notification system capable of generating real-time alerts in response to system changes and failures. By implementing such a system, the stakeholders or responsible parties can be promptly informed whenever there are critical alterations or issues within the system, enabling timely responses and actions to mitigate problems.
	TR 5.2 Configure email and SMS notifications for timely communication.	Configuring email and SMS notifications ensures timely communication of alerts to designated individuals or teams. Email and SMS are widely used communication channels that can reach stakeholders efficiently. By configuring these channels for notifications, the system can promptly inform relevant personnel, allowing for immediate attention to changes or failures, even if they are not actively monitoring the system dashboard.
	TR 5.3	Configuring alerting thresholds involves defining specific criteria or thresholds for changes or failures

	Configure alerting thresholds for specific changes and failures.	that trigger notifications. This ensures that alerts are only sent when predefined thresholds or conditions are met, preventing unnecessary notifications for minor changes while highlighting significant alterations or failures that require attention.
	TR 5.4 Notification system must be automated to alert on pricing changes.	Automating the notification system to alert on pricing changes aligns with specific business needs related to pricing alterations. By configuring the system to automatically generate alerts for pricing changes, relevant stakeholders can be immediately informed of these alterations, enabling them to respond accordingly or take necessary actions.
BR 6 **Enable easy switch of domain names and also automatic reroute on domain changes.**	TR 6.1 Implement a domain name management system for seamless switching.	This requirement involves implementing a domain name management system that facilitates seamless switching between different domain names. Such a system allows administrators to efficiently manage domain names, making it easier to switch between them as required. Having a centralized management system streamlines the process of domain name changes, ensuring smooth transitions between domains.
	TR 6.2 Set up automatic DNS rerouting for domain changes to minimize downtime.	Automatic DNS rerouting ensures that when a domain name change occurs, the Domain Name System (DNS) is automatically updated to reflect this change. This helps minimize downtime by swiftly rerouting incoming traffic to the new domain, ensuring that users are directed to the correct destination without interruptions or delays.
	TR 6.3 Implement URL redirection policies to reroute traffic during domain changes.	URL redirection policies play a crucial role during domain changes by redirecting incoming traffic from the old domain to the new one. Implementing these policies ensures that even if users try to access the services or resources using the old domain, they are automatically redirected to the new domain. This redirection helps maintain continuity of service and prevents disruptions for users unaware of the domain change.
BR 7	TR 7.1	The requirement to establish a centralized logging

Implement incident reporting mechanisms to record breaches or privacy incidents.	Establish a centralized logging system to support traceability in the event of data breach.	system aims to support traceability in the event of a data breach or privacy incident. Such a system collects and stores logs from various sources, providing a centralized repository to track events, actions, and potential security breaches. It enables organizations to trace back and investigate incidents for auditing, analysis, and response purposes.
	TR 7.2 Define procedures for incident documentation, reporting, and resolution.	Defining procedures for incident documentation, reporting, and resolution is crucial. It ensures that there's a structured and standardized approach to record incidents, report them to the relevant stakeholders, and resolve them effectively. Having well-defined procedures facilitates a consistent and organized response to incidents, ensuring that incidents are documented accurately and resolved promptly.
	TR 7.3 Ensure incident response procedures are compliant with legal regulations.	Ensuring that incident response procedures comply with legal regulations is essential for handling breaches and privacy incidents appropriately. Compliance with legal requirements ensures that the incident response processes meet legal standards, which is crucial for handling sensitive data and privacy incidents while adhering to regulatory obligations.
	TR 7.4 Automate Security policy updates in order to prevent any manual errors.	Automating security policy updates helps in preventing manual errors that might occur during manual updates. Automated systems can consistently and accurately update security policies, ensuring that the system remains secure and compliant. This helps prevent human errors that could potentially lead to security vulnerabilities or non-compliance.
BR 8 **Optimize infrastructure costs.**	TR 8.1 Employ resource tagging for tracking resource costs and allocation	Resource tagging allows for the categorization and labeling of resources within the infrastructure. By tagging resources, it becomes easier to track and allocate costs associated with each resource. This helps in identifying which resources are consuming more resources and costing more, enabling better cost management and optimization.

	TR 8.2 Use cloud cost optimization tools to identify and manage resource waste.	Leveraging specialized tools for cloud cost optimization helps in identifying resource wastage or inefficiencies within the infrastructure. These tools analyze usage patterns, identify unused or underutilized resources, and suggest optimizations. By employing such tools, organizations can proactively manage and optimize their cloud expenses, ensuring efficient resource utilization and cost savings.
	TR 8.3 Select vendors charging lesser prices for higher rated/more popular services.	Opting for vendors offering competitive prices for high-rated or popular services can contribute significantly to cost optimization. Comparing and selecting vendors based on the balance between service quality and cost efficiency helps in reducing infrastructure expenses while maintaining service standards.
	TR 8.4 Constantly keep checking and updating the services for cost optimization possibilities.	Constantly reviewing and updating services for cost optimization possibilities is crucial. Technologies and pricing structures evolve, and periodically reassessing the infrastructure's components helps in identifying new cost-saving opportunities. This proactive approach ensures that the infrastructure remains optimized and aligned with cost-saving strategies.
BR 9 **Choose providers with environmental certifications.**	TR 9.1 Select cloud providers with recognized environmental certifications (like ISO 14001).	This requirement involves selecting cloud service providers that possess recognized environmental certifications, such as ISO 14001. These certifications ensure that the chosen cloud providers adhere to established environmental management standards, demonstrating their commitment to reducing their environmental impact through structured practices and policies.
	TR 9.2 Ensure that data centers adhere to green computing practices.	Ensuring that data centers adhere to green computing practices involves implementing energy-efficient technologies, optimizing server use, and employing eco-friendly hardware. These practices aim to reduce energy consumption and minimize the environmental footprint of data centers while maintaining operational efficiency.
	TR 9.3	Choosing data centers located in regions with

	Choose data centers with optimal climatic conditions to minimize cooling efforts.	optimal climatic conditions can help minimize the need for excessive cooling efforts. Favorable climatic conditions, such as cooler temperatures or natural airflow, can contribute to reducing the energy required for cooling systems within the data centers, thereby promoting energy efficiency and reducing environmental impact.
	TR 9.4 Choose data centers with access to green energy resources to minimize electricity usage.	Selecting data centers that have access to or utilize green energy resources, such as renewable energy sources (solar, wind, hydroelectric, etc.), helps minimize reliance on traditional fossil fuels. Utilizing green energy sources reduces the carbon footprint associated with electricity usage in data centers, aligning with environmental sustainability goals.
BR 10 **Design for rapid scalability to accommodate fluctuating user demand.**	TR 10.1 Implement auto-scaling based on user traffic patterns.	Auto-scaling allows the system to automatically adjust its capacity based on user traffic patterns. By analyzing incoming user traffic, the system can dynamically allocate resources (such as servers, CPU, memory) to match the demand. When there's a surge in user activity, auto-scaling ensures that the system scales up to handle the increased load, maintaining performance and responsiveness without manual intervention.
	TR 10.2 Utilize serverless architecture to dynamically scale resources as needed.	Serverless architecture abstracts infrastructure management, allowing resources to be allocated dynamically based on demand. With serverless computing, resources are provisioned automatically as needed, allowing for rapid scaling without the need for pre-allocated servers. This enables the system to respond instantly to fluctuating user demand by dynamically provisioning resources, ensuring optimal scalability and efficient resource utilization.
BR 11 **Implement a disaster recovery plan to ensure data resilience.**	TR 11.1 Create secure data backups and store them in geographically diverse locations.	Creating data backups and storing them in geographically diverse locations minimizes the risk of data loss in case of disasters affecting a specific region. This strategy ensures redundancy and resilience by having copies of critical data stored in multiple locations, reducing the impact of localized incidents on data availability.

	TR 11.2 Develop and test a disaster recovery plan with a defined RTO (Recovery Time Objective).	Developing and testing a disaster recovery plan with a defined Recovery Time Objective (RTO) sets clear expectations for how quickly systems and data can be recovered after a disaster. Testing this plan ensures that it's effective and can be executed within the desired timeframe, minimizing downtime and ensuring prompt recovery in case of an incident.
	TR 11.3 Ensure that data retention policies are in accordance with legal requirements, business needs and usage patterns.	Ensuring that data retention policies align with legal requirements, business needs, and usage patterns is crucial for effective disaster recovery. Properly managed data retention policies ensure that necessary data is available for recovery purposes while adhering to regulatory compliance and meeting business requirements.
	TR 11.4 Disaster recovery drills must be conducted at least once a year.	Regular disaster recovery drills at least once a year test the effectiveness of the plan, familiarize the team with procedures, and identify areas for improvement. These drills simulate disaster scenarios, allowing teams to practice their responses, validate the recovery plan, and refine it based on the lessons learned.
	TR 11.5 Data recovery must be automated in the event of a failure.	Implementing automation for data recovery in the event of a failure reduces response time and human error. Automated recovery processes can swiftly restore data or systems, ensuring a faster recovery and minimizing the impact of incidents on business operations.
BR 12 **Manage and Store Large Volumes of Multimedia Content Efficiently.**	TR 12.1 Utilize Content Delivery Networks (CDNs) for Efficient Content Distribution.	CDNs play a crucial role in efficiently managing and distributing large volumes of multimedia content. By strategically caching and delivering content closer to end-users, CDNs reduce latency, improve content delivery speed, and enhance the overall user experience. This technical requirement aligns with the business goal of efficient content management.
	TR 12.2 Implement Deduplication Techniques for	Deduplication is a key technique for optimizing storage efficiency. By identifying and eliminating duplicate copies of data, this technical requirement reduces storage space requirements, leading to more efficient storage utilization. This is particularly

	Storage Optimization.	valuable when dealing with large volumes of multimedia content that may have redundant elements.
	TR 12.3 Implement Content Compression Techniques to Optimize Storage Capacity.	Content compression is essential for maximizing storage capacity and minimizing storage costs. By reducing the size of multimedia files through compression techniques, this technical requirement contributes to efficient storage utilization without compromising the quality of the content. It aligns with the goal of managing large volumes of multimedia content effectively.
BR 13 **User Data Should Be Stored and Backed Up Reliably, Securely, and Efficiently.**	TR 13.1 (TR11.1) Create secure data backups and store them in geographically diverse locations.	Secure data backup procedures ensure that user data is consistently and regularly backed up, minimizing the risk of data loss. By incorporating encryption, the technical requirement enhances data security during the backup process, safeguarding sensitive information from unauthorized access.
	TR 13.2 Implement data lifecycle management to optimize storage costs.	Data lifecycle management is crucial for optimizing storage costs associated with user data. By defining policies for data retention, archiving, and deletion based on its lifecycle, the cloud architecture can efficiently manage storage resources and reduce unnecessary costs while ensuring compliance with data management regulations.
	TR 13.3 Implement encryption for data at rest and in transit to ensure data security.	Security is paramount when handling user data. Implementing encryption for data at rest and in transit adds an additional layer of protection, preventing unauthorized access and potential data breaches. This technical measure aligns with the business requirement for secure user data backup.
	TR 13.4 Implement authentication systems to prevent unauthorized access to data.	To further enhance the security of backup data, implementing robust authentication systems is necessary. This technical requirement ensures that only authorized personnel can access and manage the backup data, reducing the risk of unauthorized tampering or data breaches.
BR 14 **Interoperability Must Be**	TR 14.1 Implement open standards and APIs	To address the business requirement of accommodating changing interoperability needs, implementing open standards and APIs is crucial. Open standards ensure compatibility and facilitate

Accommodated Keeping in Mind Changing Needs.	for better interoperability.	communication between different systems, applications, and services. APIs provide a standardized way for diverse systems to interact, promoting flexibility and adaptability to evolving interoperability requirements.
	TR 14.2 Establish integration frameworks for connecting with external systems.	To further support interoperability, the establishment of integration frameworks becomes necessary. These frameworks serve as a structured approach to connect and integrate with external systems seamlessly. By adopting integration frameworks, the cloud architecture becomes more agile and capable of adapting to changing needs, ensuring that interoperability remains a key consideration in the face of evolving technological landscapes.
BR 15 **Billing Information Should Be Manageable**	TR 15.1 There must be a billing and invoicing system with detailed reporting of resource and service utilization.	To ensure billing information is manageable, the implementation of a robust billing and invoicing system is crucial. This system should provide detailed reports on resource consumption, service usage, and associated costs. Having a granular breakdown of the billing information allows for better transparency, cost tracking, and management of financial resources within the cloud environment.
	TR 15.2 Ensure set procedures to settle billing disputes.	It is essential to establish clear and well-defined procedures for settling billing disputes. This technical requirement ensures that in case of discrepancies or concerns regarding the billed amount, there are established processes to investigate, resolve, and settle such disputes. Having a systematic approach to handle billing discrepancies contributes to maintaining a transparent and trustworthy relationship between the cloud service provider and the consumer.
BR 16 **Consider Well-Reputed Vendors**	TR 16.1 Conduct vendor assessments based on industry reputation and reviews.	Before engaging with a vendor, it is crucial to evaluate their reputation within the industry. This involves researching customer reviews, testimonials, and industry reports to gauge the vendor's performance, reliability, and customer satisfaction. Choosing vendors with positive reviews contributes to the overall success and reliability of the services provided.
	TR 16.2 Verify vendors' compliance with	Ensuring that vendors adhere to industry standards and possess relevant certifications is essential for evaluating their commitment to quality and compliance. Verification against standards such as

	industry standards and certifications.	ISO, SOC, or other industry-specific certifications provides confidence in the vendor's ability to meet regulatory requirements and maintain a high level of service.
	TR 16.3 Consider vendors who are compliant with the security guidelines provided by NIST.	The National Institute of Standards and Technology (NIST) provides widely recognized security guidelines. Choosing vendors compliant with NIST guidelines ensures a strong focus on security best practices. This includes robust measures for data protection, encryption, access controls, and overall cybersecurity. Alignment with NIST standards enhances the security posture of the organization's cloud environment.
BR 17 **Avoid Vendor Lock-In**	TR 17.1 Develop a multi-cloud strategy to prevent dependency on a single vendor.	Implementing a multi-cloud strategy ensures that the organization is not tied to a single cloud vendor. By diversifying cloud service providers, the business reduces the risk of vendor-specific limitations and dependencies. This approach allows the flexibility to choose the most suitable cloud services from different vendors based on performance, cost, and features, preventing vendor lock-in.
	TR 17.2 Utilize containerization and orchestration to enable workload portability.	Containerization, coupled with orchestration tools like Kubernetes, provides a standardized and portable environment for applications. This ensures that workloads can run consistently across various cloud providers, facilitating easy migration and preventing vendor lock-in. Containers encapsulate dependencies, making applications independent of the underlying infrastructure.
	TR 17.3 Develop a cloud-agnostic strategy for porting applications between different cloud providers.	A cloud-agnostic strategy involves designing applications and services to be independent of specific cloud provider features. This allows for seamless migration between different cloud environments, reducing the impact of vendor lock-in. The strategy involves using standard APIs, avoiding proprietary features, and ensuring compatibility with multiple cloud platforms.
	TR 17.4 Leverage container orchestration platforms for vendor-agnostic workload management.	Container orchestration platforms, such as Kubernetes, provide a consistent way to deploy and manage applications across diverse environments. By adopting such platforms, organizations can achieve vendor-agnostic workload management. This ensures that applications can be easily moved and managed across different cloud providers without being tightly coupled to any specific

		vendor's ecosystem.
BR 18 **Optimize Data Redundancy**	TR 18.1 Create only the minimum required replicas of the data to achieve optimal redundancy.	Creating the minimum required replicas ensures that data redundancy is achieved without unnecessary duplication. This approach optimizes storage resources by avoiding excessive replication of data, reducing storage costs, and improving overall efficiency. It ensures that redundancy meets business requirements without introducing unnecessary complexity or overhead.
	TR 18.2 Employ data deduplication and compression techniques to reduce storage overhead.	Implementing data deduplication and compression techniques further optimizes data redundancy by reducing storage overhead. Deduplication eliminates redundant copies of data, and compression reduces the space required to store data. Together, these techniques contribute to efficient resource utilization, minimize storage costs, and enhance the performance of storage systems. This ensures that redundant data is managed in a space-efficient manner while meeting redundancy requirements.
BR 19 **Optimize content delivery for global scalability and low latency.**	TR 19.1 Employ content distribution networks (CDNs) to cache content closer to end-users.	CDNs distribute cached content across a network of servers strategically placed at various geographic locations. By doing so, the proximity of content to end-users is reduced, minimizing latency and improving scalability. This ensures that users worldwide experience faster content delivery, enhancing the overall user experience.
	TR 19.2 Implement edge computing for reduced latency in remote regions.	Edge computing involves processing data closer to the source of data generation, reducing the round-trip time for data to travel between users and centralized data centers. Implementing edge computing nodes in remote regions helps optimize content delivery by processing and delivering content locally, significantly reducing latency for users in those areas.
	TR 19.3 Implement global	Global load balancing distributes user requests across multiple servers located in different regions,

	load balancing for directing user requests to the nearest available servers.	ensuring that each request is directed to the nearest available server. This reduces the distance data needs to travel, minimizing latency and optimizing content delivery. It also enhances scalability by efficiently distributing the load among multiple servers, preventing overload on specific resources.
BR 20 **Ensure Simple Network Management**	TR 20.1 Implement automated simple network provisioning and configuration tools that simplify the process of adding new network devices or making changes to existing configurations.	Automated network provisioning tools streamline the addition of new devices and configuration changes, reducing the complexity and potential for errors associated with manual processes. By automating these tasks, the network management becomes simpler, more efficient, and less prone to human errors, ensuring reliable network operation
	TR 20.2 Implement Network Management Tools with User-Friendly Interface	Network Management Tools with a user-friendly interface contribute to simplicity by providing administrators with an intuitive platform to monitor and manage network resources. A user-friendly interface reduces the learning curve, allowing administrators to navigate and configure network settings easily. This ensures that network management tasks are accessible to a broader range of administrators, promoting efficiency and simplicity.
	TR 20.3 Enable remote network management capabilities, allowing administrators to oversee and make changes to the network from any location.	Enabling remote network management capabilities adds flexibility and simplicity to network management. Administrators can oversee and make necessary changes to the network from any location, reducing the need for physical presence. This remote accessibility ensures quick responses to issues, efficient troubleshooting, and overall simplified network management, especially in scenarios where on-site presence may not be practical or feasible.

2.4 TradeOffs

1. **TR 1.1, TR 18.1:** Ensuring redundancy in the architecture to ensure high availability of the application requires creating **multiple redundant replicas** of the data. However, to optimize redundancy, just **2 replicas** of the data **would suffice**. But this optimization **might not ensure the 99% availability** required by TR1.1, and hence, a trade-off needs to be made.
2. **TR 1.2, TR 1.4:** Implementing redundant network connections **can increase infrastructure costs and complexity**. It requires additional resources and maintenance to ensure the network's redundancy. However, the trade-off is necessary to **maintain high availability and reduce downtime**, which aligns with the primary business requirement.
3. **TR 4.2, TR 20.1:** Trade-off: **Increased complexity of network management.** Network segmentation **enhances security but adds complexity** to network management and maintenance. It's a trade-off between security and operational simplicity.
4. **TR 8.3, TR 9.2:** Data centers which follow **green computing practices** would **not** provide services at a **cheaper cost**, although the **quality of service might** be the same as any other data center. Therefore, a trade-off must be made between cost and green data centers.

3 Provider Selection

The finalized list of TRs we are working with is as follows:
TR3.2: Predictive analytics must be employed to forecast resource needs based on historical usage data.

TR4.1: Utilize Virtual Private Cloud (VPC) for tenant isolation.

TR5.1: Implement a notification system for real-time alerts on system changes and failures.
TR5.2: Configure email and SMS notifications for timely communication.

TR6.1: Implement a domain name management system for seamless switching.
TR6.2: Set up automatic DNS rerouting for domain changes to minimize downtime.

TR7.1: Establish a centralized logging system to support traceability in the event of data breach.

TR8.3: Select vendors charging lesser prices for higher rated/more popular services.
TR8.4: Constantly keep checking and updating the services for cost optimization possibilities.

TR11.1: Create data backups and store them in geographically diverse locations.

TR15.1: There must be a billing and invoicing system with detailed reporting of resource and service utilization.

TR19.1: Employ content distribution networks (CDNs) to cache content closer to end-users.
TR19.3: Implement global load balancing for directing user requests to the nearest available servers.

3.1 Criteria for choosing a provider

Services Offered: After comparing the services offered by various providers, we will select the provider providing the services which cater to our exact needs, as highlighted in the Technical requirements above.

In addition to those requirements, we have listed below some more criteria which we will use for choosing a provider.

- **Cost and Pricing Model:** By comparing the pricing structure (including hidden fees) between different providers, we will evaluate the cheapest provider who offers all the services required by our infrastructure. Providers with transparent pricing models will be favored.

- **Geographical Reach and Data Center Locations:** Having data centers in various regions can enhance performance and compliance with data sovereignty laws. Providers with sporadic data center locations, which align with our requirements, will be favored.
- **Service and Support:** We will evaluate the level of customer support extended by each provider with metrics like response times, availability, and support channels. The provider with the best service, with good reviews and testimonials will be favored.
- **Experience and Reputation:** Well reputed providers with the lowest pricing structure will be chosen.

3.2 Provider Comparison

In designing a cloud architecture, the choice of a cloud provider significantly impacts the overall performance, scalability, and cost-effectiveness of the system. Given that we are considering the three major cloud providers - AWS, GCP, and Azure - the selection process ultimately comes down to several key factors, with pricing being a crucial determinant.

Pricing Consideration:

- **AWS:** Known for its comprehensive set of services, AWS offers a pay-as-you-go pricing model. While it provides a vast array of resources, we should carefully optimize their usage to avoid unnecessary costs.
- **GCP:** Google Cloud Platform, often competitive in pricing, has a transparent pricing structure. It provides a sustained user discount and per-minute billing, potentially offering cost savings for specific workloads.
- **Azure:** With a focus on hybrid cloud solutions, Azure provides flexible pricing options. Its reserved instances and hybrid use benefits can be advantageous for organizations with existing Microsoft investments.

Comparison Metrics:

- **Service Offerings:** Assess the specific services required for your architecture. AWS has an extensive list, GCP is renowned for data analytics, and Azure seamlessly integrates with Microsoft technologies.
- **Global Reach:** Considering the geographical distribution of data centers, AWS has a broad global presence, GCP emphasizes network infrastructure and Azure's strength lies in integration with Microsoft's global network.
- **Ease of Use:** Evaluating the ease of use and management interfaces, AWS has a mature and feature-rich console, GCP focuses on simplicity, and Azure integrates seamlessly with Windows environments.
- **Community and Ecosystem:** Examining the community support and ecosystem around each provider, AWS boasts a large and active community, GCP emphasizes innovation, and Azure aligns well with Microsoft's enterprise ecosystem.

TR	AWS SERVICE	GCP Service	Azure Service	Ranking Justification
TR3.2: Predictive analytics must be employed to forecast resource needs based on historical usage data.	AWS Kinesis Video Streaming $0.08 per GB/ 24 hours Data retention	DataFlow $0.011 Per GB No data retention	Azure Stream Analytics (Slab Payments) 0 - 730 SU Hours $0.406 730 - 5,840 SU Hours $0.178 5,840+ SU Hours $0.148	1. AWS 2. Azure 3. GCP Cause of the price and data retention policy
TR4.1: Utilize Virtual Private Cloud (VPC) for tenant isolation.	AWS VPC $0.045 Per Nat Gateway / Hr + $0.015 Per ENI/Hr	Private Google Access $0.044 per Nat gateway / hr + $0.025 / Hr Network telemetry	Azure Virtual Network $0.045 per Nat gateway / per hour + $0.035 per hour inbound VNET peering	1. AWS 2. GCP 3. Azure AWS is selected for its competitive pricing and reliable VPC services.
TR5.1: Implement a notification system for real-time alerts on system changes and failures. TR5.2: Configure email and SMS notifications for timely communication.	AWS SNS, SES $2.00 per 100,000 notifications	Cloud Pub/Sub Volume: 1 GiB USD 0.80 + Subscriptions: 1 USD 1.1.49	Azure Service bus $0.05 per million operations + $0.0135/hour	1. Azure 2. AWS 3. GCP AWS is chosen for its cost-effectiveness and efficient notification services.
TR6.1: Implement a domain name management	Route 53 $0.40 per million	Google Cloud DNS $0.40 per million queries per	Azure DNS , Traffic Manager $0.54 per million queries	1. AWS 2. GCP 3. Azure AWS is

system for seamless switching. TR6.2: Set up automatic DNS rerouting for domain changes to minimize downtime.	queries / month	month		preferred for its competitive pricing and reliability.
TR7.1: Establish a centralized logging system to support traceability in the event of data breach.	AWS Cloudwatch $0.25 per GB per month	Stackdriver $0.01 per GB per month	Azure Monitor $2.30 per GB per month for analytics logs included	1. GCP 2. AWS 3. Azure Based on its cost efficiency and reliable logging solutions.
TR11.1: Create data backups and store them in geographically diverse locations.	RDS It runs on EC2 instances, for comparison we are gonna consider t4g.small $0.0320/ Hr	Cloud SQL, Bare Metal Solution $0.0537 per vCPU + $0.17 per GB/month for SSD storage capacity	Azure SQL database $0.5218/vCore-hour + $0.115 Per GB/ Per Month	1. AWS 2. GCP 3. Azure AWS is favored for its competitive pricing and comprehensive backup solutions.
TR15.1: There must be a billing and invoicing system with detailed reporting of resource and service utilization.	AWS Billing Conductor $8.25 per account for the first 500 accounts $6.75 per account for the next 1500 $5.25 per account for more than 2000	Cloud Billing Cloud Billing API is free for use	Azure Cost Management and Billing 1% of the total managed spend	1. GCP 2. AWS 3. Azure Though it's not free, AWS is selected based on its flexible pricing structure and features.

TR19.1: Employ content distribution networks (CDNs) to cache content closer to end-users.	CloudFront * footnote 1 varies based on region	Cloud CDN, Media CDN * footnote 2 varies based on region	Front Door * footnote 3 varies based on region	1. AWS 2. GCP 3. Azure AWS is chosen for its reliable CDN and load balancing services.
TR19.3: Implement global load balancing for directing user requests to the nearest available servers.	ELB $0.0225 per Application Load Balancer-hour (or partial hour)	Cloud Load Balancing $0.025/hour	Azure Load Balancer $0.025 / Hr	It's all of the same price so based on ease of usability we can rank them. 1. AWS 2. Azure 3. GCP

3.3 The Final Selection

Considering the importance of pricing in the decision-making process and the fact that each provider offers a comparable set of services, the justification for selecting a particular provider comes down to **optimizing costs without compromising on performance and reliability**.

After a meticulous evaluation of AWS, GCP, and Azure, **AWS** has been selected as the preferred cloud provider for its several advantages. AWS stands out due to its **fair and transparent pricing model, user-friendly interface, and a variety of features that align well with the specified Technical Requirements** (TRs). The decision is driven by AWS's commitment to accessibility, cost-effectiveness, and service reliability. The comprehensive suite of services offered by AWS, coupled with its user-friendly approach, positions it as the optimal choice for meeting the outlined business and technical needs.

3.3.1 The list of services offered by the winner

The below are some of the services provided by the winner that we might be using in our cloud architectural design.

1. AWS Kinesis : https://aws.amazon.com/kinesis/
2. AWS VPC : https://aws.amazon.com/vpc/

1 check https://aws.amazon.com/cloudfront/pricing/ for details

2 check https://cloud.google.com/cdn/pricing for details

3 check https://azure.microsoft.com/en-us/pricing/details/frontdoor/ for details

3. AWS SNS : https://aws.amazon.com/sns/
4. AWS Route 53 : https://aws.amazon.com/route53/
5. AWS Cloudwatch : https://aws.amazon.com/cloudwatch/
6. RDS : https://aws.amazon.com/rds/
7. Billing Conductor : https://aws.amazon.com/aws-cost-management/aws-billing-conductor/
8. CloudFront : https://aws.amazon.com/cloudfront/
9. ELB : https://aws.amazon.com/elasticloadbalancing/
10. AWS AppStream : https://aws.amazon.com/elasticloadbalancing/
11. AWS EKS : https://aws.amazon.com/eks/
12. AWS Lambda : https://aws.amazon.com/lambda/
13. AWS API Gateway : https://aws.amazon.com/api-gateway
14. AWS AppSync : https://aws.amazon.com/appsync

The comprehensive list of all services provided by AWS can be found in the link below: AWS Services .

AWS Lambda

AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS), and it plays a pivotal role in modern cloud-native projects. This service allows developers to run code without the need to provision or manage servers, offering a truly serverless architecture.

Key Features:

1. **Event-Driven Architecture:** AWS Lambda operates on an event-driven model, meaning it responds to events triggered by various AWS services or custom events. This architecture promotes flexibility and scalability.
2. **Multiple Language Support:** Lambda supports multiple programming languages, including Node.js, Python, Java, Go, and more. This flexibility enables developers to choose the language that best fits their application logic.
3. **Cost-Efficiency:** AWS Lambda follows a pay-as-you-go pricing model. You are charged based on the number of requests for your functions and the time your code executes. This cost-effective approach ensures you only pay for the compute time you consume.
4. **Scalability:** Lambda automatically scales based on the incoming request volume. It can handle a single request or scale to thousands of requests per second, making it suitable for a wide range of applications.
5. **Integrations:** Lambda seamlessly integrates with other AWS services, facilitating the creation of powerful serverless applications. Common integrations include S3 for object storage, API Gateway for building RESTful APIs, and DynamoDB for NoSQL database needs.
6. **Stateless Execution:** Lambda functions are designed to be stateless. They execute independently, with no affinity to the underlying infrastructure. This statelessness ensures easy scalability and fault tolerance.

Use Cases in Projects:

1. **Microservices Architecture:** AWS Lambda is instrumental in building microservices-based architectures. Each function can represent a microservice, allowing for a modular and scalable application structure.
2. **Real-time File Processing:** Lambda can be triggered by events such as file uploads to S3. This makes it ideal for real-time processing of files, like generating thumbnails or extracting metadata.
3. **Backend for Mobile and Web Applications:** Lambda serves as an excellent backend for mobile and web applications. It allows developers to focus on writing code without managing servers, promoting faster development cycles.
4. **Automation and Orchestration:** Lambda can be used for automating routine tasks or orchestrating workflows. For instance, it can respond to CloudWatch Events, triggering automated actions in response to system events.
5. **Chatbots and Voice Assistants:** Building chatbots or voice assistants becomes more manageable with Lambda. It can process and respond to user input in real time.

In summary, AWS Lambda is a powerful serverless computing service that aligns well with the paradigm shift towards serverless and event-driven architectures. Its flexibility, cost-effectiveness, and seamless integration with other AWS services make it a valuable asset for modern project development.

AWS CloudFront

AWS CloudFront is a content delivery service provided by Amazon Web Services (AWS), designed to enhance the performance, security, and scalability of web applications. It operates as a content delivery network (CDN), strategically distributing content across a global network of servers to reduce latency and improve end-user experience.

Key Features:

1. **Content Distribution:** AWS CloudFront accelerates the delivery of static and dynamic content, including web pages, images, videos, and APIs. It caches content at edge locations globally, reducing the distance between users and resources.
2. **Global Edge Network:** CloudFront leverages a vast network of edge locations strategically positioned around the world. These edge locations serve as caching points, ensuring faster content delivery by minimizing the physical distance between end-users and the servers.
3. **Security Measures:** CloudFront integrates with AWS Web Application Firewall (WAF) to protect against common web exploits and attacks. It also supports HTTPS (SSL/TLS) to encrypt data during transit, enhancing overall security.
4. **Customization and Personalization:** CloudFront allows for the customization of content delivery through features like Lambda@Edge. Developers can add business logic to modify or personalize the content based on user requirements.
5. **Origin Fetch Options:** CloudFront supports various origin fetch options, enabling content retrieval from diverse sources. This includes directly from an Amazon S3 bucket, an Elastic Load Balancer, or even a custom HTTP server.

6. **Cost Optimization:** With a pay-as-you-go pricing model, CloudFront is cost-effective. It minimizes data transfer costs and reduces the load on origin servers, as content is cached at edge locations.
7. **Streaming Media Support:** CloudFront supports the delivery of streaming media content, making it suitable for distributing video on demand (VOD) or live streaming content.

Use Cases in Projects:

1. **Website Acceleration:** CloudFront accelerates the delivery of entire websites, including both static and dynamic content. This results in faster page loads, improved user experience, and reduced bounce rates.
2. **Global API Delivery:** For applications with APIs, CloudFront helps in delivering API responses quickly by caching them at edge locations. This is crucial for low-latency access to APIs globally.
3. **Secure Content Delivery:** CloudFront, when combined with AWS WAF, provides security against common web exploits. It allows developers to implement security policies to protect web applications from various threats.
4. **Scalable Video Delivery:** For projects involving video content, CloudFront facilitates scalable and low-latency video delivery. This is particularly beneficial for streaming services and online video platforms.
5. **E-commerce Optimization:** CloudFront enhances the performance of e-commerce websites by caching product images, CSS files, and other assets, leading to a faster and more responsive online shopping experience.

In summary, AWS CloudFront is a powerful CDN service that optimizes the delivery of web content, enhances security, and improves the overall performance of web applications. Its global network, security features, and seamless integration with other AWS services make it a valuable component for projects aiming to deliver content efficiently and securely worldwide.

AWS API Gateway

AWS API Gateway is a fully managed service that simplifies the creation, deployment, and management of APIs (Application Programming Interfaces) at scale. It serves as a crucial component in building and maintaining serverless architectures and microservices by providing a secure, scalable, and customizable interface for applications to interact with backend services.

Key Features:

1. **API Creation and Design:** AWS API Gateway enables the creation of RESTful APIs or WebSocket APIs, allowing developers to define endpoints, methods, request/response transformations, and other aspects of API design.
2. **Endpoint Configuration:** API Gateway supports a variety of endpoint types, including HTTP/HTTPS, Lambda function, AWS service integrations, and more. This flexibility allows for seamless integration with various backend resources.

3. **Integration with AWS Services:** API Gateway easily integrates with other AWS services, such as AWS Lambda, AWS Step Functions, or even an HTTP endpoint hosted on EC2, providing a serverless architecture and reducing operational overhead.
4. **Security and Access Control:** Robust security measures are in place, including support for OAuth and AWS Identity and Access Management (IAM) for authentication and authorization. This ensures that APIs are secure and only accessible by authorized users or systems.
5. **Rate Limiting and Throttling:** API Gateway allows you to set up rate limits and throttling to control the rate at which clients can make requests. This helps in preventing abuse, ensuring fair usage, and protecting backend systems from traffic spikes.
6. **Monitoring and Logging:** AWS CloudWatch integration provides detailed monitoring and logging capabilities. Developers can monitor API usage, error rates, and latency, gaining insights into the performance of APIs and identifying areas for improvement.
7. **Custom Domain Names:** Custom domain names can be configured for APIs, providing a branded and user-friendly URL for API access. This is particularly useful in scenarios where APIs are exposed to external clients.
8. **API Documentation:** API Gateway generates and hosts comprehensive API documentation automatically. This documentation includes details about API endpoints, request/response models, and usage instructions, making it easier for developers to understand and consume the API.

Use Cases in Projects:

1. **Microservices Architecture:** API Gateway plays a vital role in creating a unified API layer for microservices. It enables the development of independently deployable services, making it easier to manage and scale.
2. **Serverless Applications:** When combined with AWS Lambda, API Gateway simplifies the creation of serverless applications. It acts as a trigger for Lambda functions, allowing developers to build powerful serverless APIs without the need for infrastructure management.
3. **Mobile and Web Applications:** API Gateway is a key component for backend services in mobile and web applications. It provides a secure and scalable interface for frontend applications to interact with backend resources.
4. **Legacy System Integration:** For projects dealing with legacy systems, API Gateway can be used to modernize the backend by exposing RESTful interfaces. This allows for gradual migration and integration with newer technologies.
5. **IoT Applications:** API Gateway can be utilized in Internet of Things (IoT) projects to manage and secure communication between devices and backend services. It facilitates the creation of scalable and secure APIs for IoT applications.

In summary, AWS API Gateway is a powerful service for managing APIs in the AWS cloud environment. Its features make it well-suited for a variety of use cases, ranging from microservices architectures to serverless applications and beyond. As a key component in modern application development, API Gateway simplifies the process of creating, securing, and scaling APIs, contributing to the overall success of cloud-based projects.

AWS SNS

AWS Simple Notification Service (SNS) is a fully managed messaging service that enables the decoupling of microservices, distributed systems, and serverless applications by providing reliable and scalable communication. SNS facilitates the sending of messages or notifications to a distributed set of recipients or subscribers through various communication protocols.

Key Features:

1. **Topic-based Messaging:** AWS SNS operates on a publish/subscribe model, allowing communication between publishers (senders) and subscribers (receivers) through named topics. This topic-based architecture promotes decoupling, flexibility, and scalability in message distribution.
2. **Protocol Support:** SNS supports multiple protocols for message delivery, including HTTP/HTTPS, email/SMTP, SMS, SQS (Simple Queue Service), Lambda, and application endpoints. This flexibility allows developers to choose the most suitable communication method for their use case.
3. **Message Filtering:** SNS enables message filtering based on message attributes, allowing subscribers to receive only the messages relevant to their interests. This feature is valuable in scenarios where multiple types of messages are published on the same topic.
4. **Fanout and Scaling:** SNS supports fanout, meaning a single message published to a topic can be delivered to multiple subscribers simultaneously. This ensures that each subscriber receives the message independently, promoting horizontal scaling and parallel processing.
5. **Delivery Retry and Dead-Letter Queues:** SNS automatically retries message deliveries to subscribers in case of delivery failures. Additionally, dead-letter queues can be configured to capture and analyze messages that couldn't be delivered after multiple attempts.
6. **Mobile Push Notifications:** AWS SNS facilitates sending push notifications to mobile devices (iOS, Android, etc.). This is particularly useful for mobile applications to keep users informed of updates or events.
7. **Message Encryption:** SNS provides options for encrypting messages in transit and at rest, ensuring the confidentiality and integrity of messages. Encryption can be configured using AWS Key Management Service (KMS).
8. **Cross-Region Replication:** SNS supports cross-region replication of topics, allowing messages to be delivered to subscribers in different AWS regions. This enhances the resilience and availability of messaging across geographical locations.

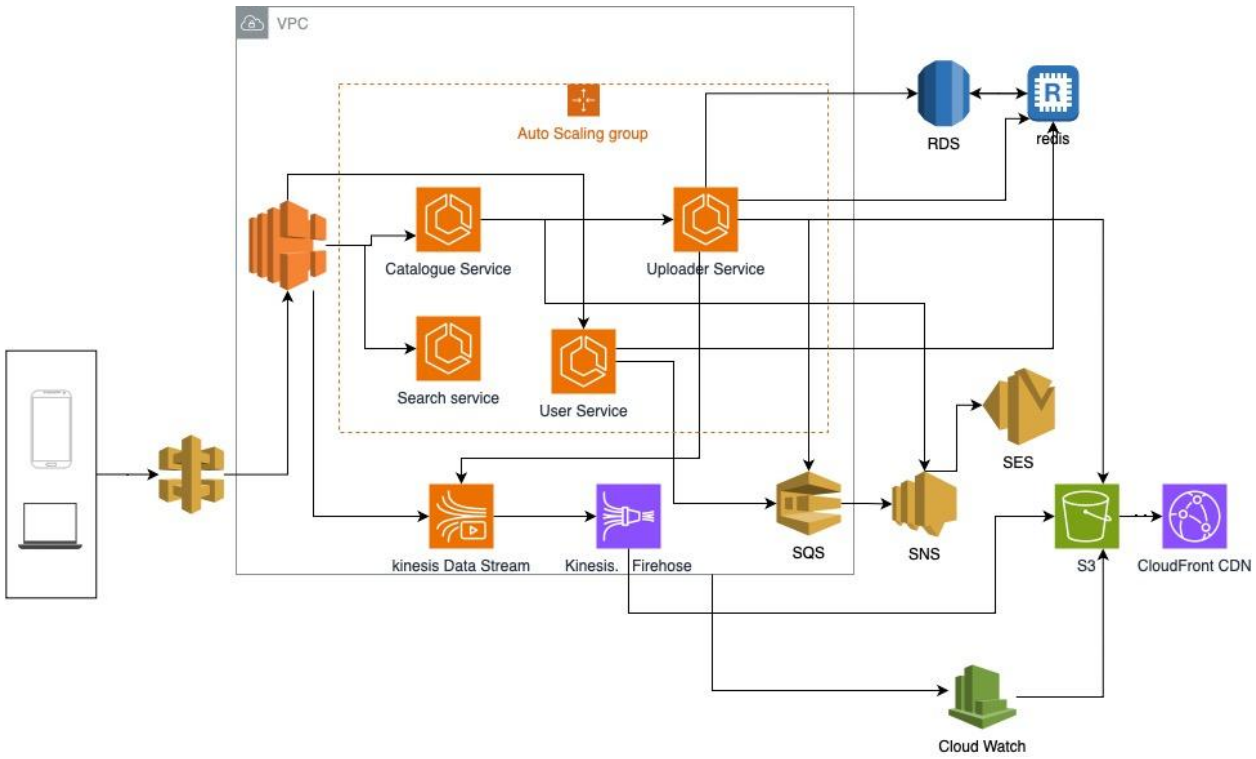
Use Cases in Projects:

1. **Event Notification:** SNS is commonly used to notify subscribers about events or updates in a system. For example, notifying users about new content, system status changes, or updates in a distributed environment.
2. **Microservices Communication:** In microservices architectures, SNS facilitates communication between loosely coupled services. It allows services to publish events, and other services can subscribe to relevant topics to react to those events.

3. **Alerting and Monitoring:** SNS is instrumental in sending alerts and notifications for monitoring purposes. It can integrate with monitoring tools to notify administrators or DevOps teams about system health, performance issues, or anomalies.
4. **Mobile App Messaging:** SNS is frequently employed to send push notifications to mobile devices. Mobile applications use SNS to notify users of new messages, updates, or other relevant information.
5. **Workflow Orchestration:** SNS is used to trigger and orchestrate workflows in serverless architectures. For instance, an SNS message can trigger an AWS Lambda function, initiating a specific sequence of actions.

In summary, AWS Simple Notification Service (SNS) is a versatile messaging service that provides a scalable and reliable solution for decoupling components in distributed systems. Its ability to support various communication protocols and offer features like message filtering and encryption makes it suitable for a wide range of use cases in modern cloud-based projects. SNS empowers developers to design resilient and responsive systems by facilitating efficient communication between components.

4 The First Design Draft



4.1 The Basic Building Blocks of the Design

The AWS components that we will be using in our architecture is as follows:

- CloudFront
- Route 53
- VPC
- Elastic Load Balancer
- Simple Email Service
- Simple Notification Service
- CloudWatch
- RDS
- Kinesis Video Stream
- EC2
- Billing Conductor
- S3
- Kinesis Firehose

4.2 Top-level, Informal Validation of the Design

SERVICE	JUSTIFICATION
CloudFront	CloudFront employs CDNs to cache content closer to end-users by leveraging its extensive edge location network, reducing latency, improving performance, optimizing bandwidth, ensuring secure content delivery, and offering flexibility for dynamic content caching and customization.
Route 53	Route 53 provides robust DNS management capabilities, flexible traffic routing policies, health checks for automatic failover, and rapid DNS change propagation. These features enable seamless domain management, automatic rerouting during domain changes, and minimize downtime by swiftly directing traffic to available resources, ensuring continuity of service.
AWS VPC	VPC provides an isolated, customizable, and secure network environment. It allows tenants to create dedicated virtual networks, apply specific security measures, provision isolated resources, and maintain segregation among tenants, ensuring privacy, security, and autonomy within their respective VPCs.
Elastic Load Balancer	Elastic Load Balancer (ELB) satisfies the requirement for global load balancing by providing a robust, scalable, and geographically distributed load balancing solution. It ensures optimal routing of user requests to the nearest available servers across multiple regions, delivering enhanced user experiences and maintaining high availability and fault tolerance across the infrastructure.
Simple Email Service	SES is a reliable, scalable, and cost-effective platform for sending email notifications and supporting SMS delivery through integration with Amazon SNS. Its features include template management, email authentication, cost-effective pricing, and seamless integration with other AWS services
Simple Notification Service	The capabilities of SNS include real-time notification delivery, event-driven architecture, flexibility in messaging, multi-protocol support, reliability, and integration with monitoring services, ensuring timely and efficient alerting for system events and failures.
CloudWatch	CloudWatch provides comprehensive log collection, centralized management, real-time monitoring with alerts, configurable log retention, integration with security services, and advanced log analysis capabilities. This centralized logging system enhances traceability during data breaches or security incidents by providing a consolidated view of log data for effective investigation and response.

RDS	RDS offers automated backup functionalities, Multi-AZ deployment for high availability, cross-region replication capabilities, point-in-time recovery options, encrypted backup storage, and compliance readiness. These features collectively facilitate the creation and storage of data backups in geographically diverse locations, ensuring data resilience, high availability, and disaster recovery preparedness.
Kinesis Video Stream	Amazon Kinesis Video Streams accumulates extensive historical usage data from video streams, providing a rich dataset for analyzing past patterns and trends. With its ability to process and analyze real-time video streams, Kinesis Video Streams enables the continuous collection of data, ensuring up-to-date historical usage information for predictive analytics. Kinesis Video Streams seamlessly integrates with AWS analytics services, facilitating the application of predictive models on historical usage data for accurate forecasting of resource needs.
EC2	Amazon EC2 is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers. EC2 instances can be used to run applications on the AWS infrastructure. Users can choose the type of instance based on their application's requirements, and they have full control over the computing resources. EC2 instances can be used for various purposes, including hosting applications, running virtual servers, and more.
Billing Conductor	Billing Conductor collects detailed usage data, generates granular reports, supports cost allocation, facilitates custom reporting and analytics, aids in budgeting, and integrates with AWS services for a comprehensive view of resource and service utilization. It provides a robust billing and invoicing system, and comprehensive reporting capabilities.
S3	Amazon S3 is a scalable object storage service that allows users to store and retrieve any amount of data from anywhere on the web. It is designed to offer durability, availability, and scalability for various use cases, such as backup and restore, data archiving, content distribution, and more. S3 provides a simple web interface to store and retrieve any amount of data at any time. It is organized as buckets, and each object in S3 is stored with a unique key within a bucket.
Kinesis Firehose	Amazon Kinesis Firehose is a fully managed service for delivering real-time streaming data to destinations such as Amazon S3, Amazon Redshift, Amazon Elasticsearch Service, and more. It can capture and automatically load streaming data into different storage services without the need for users to write custom applications. Kinesis Firehose is often used for collecting, transforming, and delivering large amounts of data from various sources, making it suitable for scenarios like log and event data processing. It simplifies the process of ingesting and managing streaming data in a scalable and cost-effective manner.

4.3 Action Items and a Rough Timeline

**** SKIPPED ****

5 The Second Design

5.1 Use of Well-architected Framework

The structured approach presented by the Well-Architected Framework (WAF) is as follows:

1. **Review the Five Pillars:**
    - **Operational Excellence:** Focuses on operational aspects such as automation, documentation, and continuous improvement.
    - **Security:** Emphasizes the implementation of robust security measures to protect data, systems, and assets.
    - **Reliability:** Ensures that the system operates continuously without disruptions and can recover from failures quickly.
    - **Performance Efficiency:** Addresses the optimization of resources and system performance to meet business requirements.
    - **Cost Optimization:** Aims to manage costs efficiently by optimizing resource usage and maximizing cost-effectiveness.
2. **Evaluate Workloads Using the Pillars:**
    - Review the architecture against each pillar, identifying strengths, weaknesses, and areas for improvement. This evaluation involves understanding how each pillar applies to the specific workload or system.
3. **Implement Best Practices:**
    - Apply best practices and design principles provided in the Well-Architected Framework to address areas that need improvement within each pillar. This involves implementing architectural patterns, utilizing AWS services effectively, and adopting recommended design principles.
4. **Create Actionable Improvement Plans:**
    - Develop actionable plans to address identified areas that need improvement. Prioritize recommendations based on their impact and feasibility for implementation.
5. **Implement Changes and Iterate:**
    - Implement improvements and iterate through the evaluation process periodically. Continuously monitor, refine, and optimize the architecture based on changing requirements, technological advancements, and new best practices.
6. **Use Well-Architected Tools and Resources:**
    - Leverage the Well-Architected Tool provided by AWS, which helps assess workloads against the framework's best practices. Additionally, access AWS documentation, whitepapers, and resources to gain insights and guidance for implementing best practices.
7. **Review and Share Results:**
    - Review the outcomes of the evaluations and improvements made. Share the results and knowledge gained with relevant teams and stakeholders to foster a culture of architectural excellence and continuous improvement.

5.2 Discussion of Pillars

The Security Pillar:

The Security pillar of the Amazon Well-Architected Framework emphasizes the importance of implementing robust security measures to protect data, systems, and assets within AWS environments. It focuses on identifying and mitigating potential risks, securing critical resources, and ensuring compliance with security best practices. The pillar is structured around five key areas:

Data Protection:

- **Encryption:** Emphasizes the use of encryption mechanisms for data at rest and in transit using AWS Key Management Service (KMS), ensuring that sensitive information remains protected.
- **Access Controls:** Utilizes AWS Identity and Access Management (IAM) to enforce least privilege access, implement strong authentication, and manage user permissions effectively.
- **Data Residency and Compliance:** Addresses the need to comply with data residency requirements by ensuring that data is stored in the appropriate geographical locations and adheres to regulatory compliance standards.

Privilege Management:

- **Least Privilege Access:** Ensures that permissions are granted on a need-to-know basis to limit access to resources, reducing the risk of unauthorized actions.
- **Credentials Management:** Focuses on securely managing and rotating credentials, such as access keys and passwords, using AWS Secrets Manager or AWS Systems Manager Parameter Store.

Infrastructure Protection:

- **Network Security:** Utilizes AWS Virtual Private Cloud (VPC) to establish network segmentation, implement security groups, and network access control lists (NACLs) to control traffic flow and protect against unauthorized access.
- **DDoS Mitigation:** Implements Distributed Denial of Service (DDoS) protection using AWS Shield to safeguard against and mitigate DDoS attacks targeting AWS resources.

Detective Controls:

- **Logging and Monitoring:** Implements comprehensive logging and monitoring using services like AWS CloudTrail, Amazon CloudWatch, and AWS Config to detect and respond to security-related events, unauthorized access attempts, or configuration changes.
- **Security Analytics:** Utilizes tools for analyzing logs and generating insights to identify security threats or anomalies, allowing for proactive response and incident management.

Incident Response:

- **Automated Incident Response:** Implements automated incident response mechanisms using AWS Lambda, AWS Step Functions, or AWS Config Rules to respond to security incidents promptly.
- **Runbooks and Response Plans:** Develops incident response runbooks, defining clear procedures and response plans to address security incidents effectively.

The Security pillar of the Well-Architected Framework encourages the adoption of a defense-in-depth approach, where multiple layers of security controls are implemented to mitigate risks comprehensively. It emphasizes continuous monitoring, automation, and adherence to security best practices to build and maintain secure architectures in AWS environments.

The Reliability Pillar:

The Reliability pillar within the Amazon Well-Architected Framework focuses on designing systems that operate continuously, are resilient to failures, and quickly recover from disruptions. It aims to ensure that applications and workloads can meet business requirements and maintain operational excellence even in the face of challenges. This pillar encompasses several key areas:

Foundations for Resilience:

- **Multi-Availability Zone (AZ) Architecture:** Designs systems across multiple Availability Zones to achieve high availability and fault tolerance. This approach ensures redundancy and minimizes the impact of failures within a single AZ.
- **Fault Tolerance:** Implements fault-tolerant architectures by incorporating redundancy, such as using load balancers, auto-scaling, and redundant components, to handle failures without affecting overall system functionality.

Change Management:

- **Automation and Testing:** Utilizes automation for deployments, updates, and testing procedures to ensure consistency and reduce the risk of errors during changes. Implements practices like blue/green deployments and canary releases to validate changes before applying them to production environments.
- **Rollback Strategies:** Establishes rollback mechanisms and procedures to revert changes in case of failures or unexpected behavior during deployments, maintaining system stability.

Failure Management:

- **Monitoring and Recovery:** Implements robust monitoring with tools like Amazon CloudWatch to detect failures or anomalies promptly. Configures automated recovery mechanisms, like Auto Scaling, to restore resources or systems to a healthy state automatically.
- **Chaos Engineering:** Conducts chaos experiments or game days to proactively simulate failures and observe how systems respond, allowing for the identification and mitigation of weaknesses before they become critical issues.

Performance Efficiency:

- **Scalability and Elasticity:** Designs systems that can scale dynamically based on demand using AWS services like Auto Scaling. Ensures that resources can handle varying workloads efficiently without impacting performance.
- **Optimized Resource Utilization:** Strives to optimize resource allocation and utilization to minimize waste and improve cost-effectiveness while maintaining performance levels.

Resilient Workload Architectures:

- **State Management:** Uses stateless architectures and separates stateful components to minimize dependencies and reduce the impact of failures.
- **Caching and Load Balancing:** Utilizes caching mechanisms and load balancing strategies to distribute traffic evenly and improve performance while maintaining resilience.

Recovery Planning:

- **Disaster Recovery:** Implements disaster recovery strategies and backup plans, ensuring data durability, and establishing recovery time objectives (RTOs) and recovery point objectives (RPOs) to guide recovery efforts in case of catastrophic events.

By focusing on these key areas, the Reliability pillar aims to design architectures that can withstand disruptions, maintain high availability, and recover swiftly from failures, ensuring a robust and reliable system that meets business needs consistently.

5.3 Use of Cloudformation Diagrams

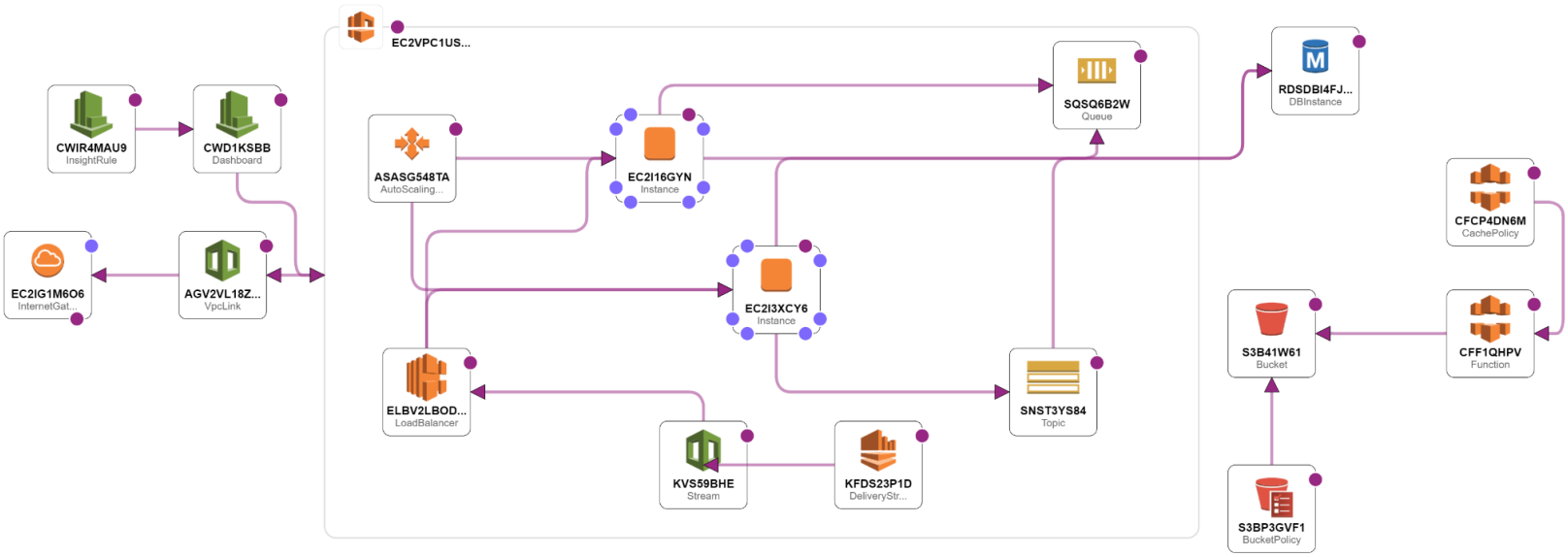


Fig 5.3.1 Cloud Formation Template Design.

5.4 Validation of the Design

TR3.2: Predictive analytics must be employed to forecast resource needs based on historical usage data.

- The **Auto Scaling group** (ASASG548TA) can be configured to use predictive scaling to automatically scale up or down the number of instances based on historical usage data.
- The **CloudFront Cache Policy** (CFCP4DN6M) can be configured to use caching to reduce the load on the EC2 instances.

TR4.1: Utilize Virtual Private Cloud (VPC) for tenant isolation.

- The EC2VPC1USKE **VPC** provides tenant isolation by creating a private network for the application.
- The AGV2VL18ZA4 **VPC link** allows the application to communicate with the internet while still maintaining tenant isolation.

TR5.1: Implement a notification system for real-time alerts on system changes and failures.

- The SNST3YS84 **SNS topic** can be used to send real-time alerts to subscribers, such as email addresses or SMS gateways.
- The SQSQ6B2W **SQS queue** can be used to buffer alerts and ensure that they are delivered even if the subscribers are unavailable.

TR5.2: Configure email and SMS notifications for timely communication.

- The SNST3YS84 **SNS topic** can be configured to send email and SMS notifications to subscribers.
- The SQSQ6B2W **SQS queue** can be used to buffer notifications and ensure that they are delivered even if the subscribers are unavailable.

TR6.1: Implement a domain name management system for seamless switching.

- The **CloudFront Function** (CFF1QHPV) can be used to **manage DNS** records and seamlessly switch between different versions of the application.

TR6.2: Set up automatic DNS rerouting for domain changes to minimize downtime.

- The **CloudFront Function** (CFF1QHPV) can be configured to **automatically reroute DNS traffic when the domain is updated.**

TR7.1: Establish a centralized logging system to support traceability in the event of data breach.

- The **Kinesis Video Stream** (KVS59BHE) can be used to capture logs from the EC2 instances.
- The **Kinesis Firehose Delivery Stream** (KFDS23P1D) can be used to deliver logs to a centralized logging system, such as **Amazon CloudWatch Logs**.

TR8.3: Select vendors charging lesser prices for higher rated/more popular services.

- The **CloudFormation template** uses AWS services that are known to be cost-effective.
- The **CloudFormation template** can be used to configure cost optimization features, such as reserved instances and spot instances.

TR8.4: Constantly keep checking and updating the services for cost optimization possibilities.

- The **CloudFormation template** can be used to monitor costs and identify opportunities for cost optimization.
- The **CloudFormation template** can be updated to take advantage of new cost optimization features.

TR11.1: Create data backups and store them in geographically diverse locations.

- The S3B41W61 **S3 bucket** can be used to store data backups.
- The **CloudFront Function** (CFF1QHPV) can be configured to replicate data backups to a different AWS region.

TR15.1: There must be a billing and invoicing system with detailed reporting of resource and service utilization.

- **AWS Cost Explorer** can be used to track and report on resource and service utilization.
- AWS Cost and Usage Reports can be used to generate detailed reports on resource and service utilization.

TR19.1: Employ content distribution networks (CDNs) to cache content closer to end-users.

- **CloudFront** can be used to cache content closer to end-users and reduce latency.
- **CloudFront** can be configured to use edge caching to cache content at the edge of the internet.

TR19.3: Implement global load balancing for directing user requests to the nearest available servers.

- The ELBV2LBODUH **load balancer** can be used to distribute user requests to the nearest available EC2 instances.
- The ELBV2LBODUH **load balancer** can be configured to use global load balancing to distribute user requests to EC2 instances in different AWS regions.

5.5 Design Principles and Best Practices Used

Defense in Depth:

- **Multiple Security Layers**: The design employs multiple security layers to protect the application from unauthorized access and attacks. This includes using security groups to restrict traffic between components, VPCs to isolate resources, and IAM roles to control access to AWS services.
- **Least Privilege**: The design adheres to the principle of least privilege by granting users and resources only the permissions they need to perform their specific tasks. This minimizes the potential impact of compromised credentials or vulnerabilities.

Favor Managed Services:

- **Managed Services for Core Functionality**: The design utilizes managed services for core functionalities, such as Amazon API Gateway and Amazon Elastic Load Balancing (ELB), to reduce operational overhead and benefit from AWS expertise.
- **Offload Non-Core Tasks**: By leveraging managed services, the design offloads non-core tasks like server management and infrastructure provisioning, allowing developers to focus on building and maintaining the core application logic.

Be Smart with State:

- **Stateless Components**: The design prioritizes stateless components for scalability and resilience. The API Gateway and ELB are stateless services, while the Kinesis Video Stream processor and Kinesis Firehose delivery stream are designed to handle transient data without persistent state.
- **Managed Stateful Services**: For components that require state, such as S3 for storing video files and CloudFront for content delivery, managed services are utilized to handle state management securely and efficiently.

Design for Automation:

- **Infrastructure as Code (IaC)**: The design utilizes **CloudFormation** to define the infrastructure as code, enabling automated deployment, configuration, and updates. This promotes consistency, reproducibility, and reduces manual intervention.
- **Automated Monitoring and Logging**: The design integrates CloudWatch for monitoring key metrics and logging application events. This enables proactive identification of issues and facilitates debugging and troubleshooting.

These principles and best practices contribute to a secure, scalable, and maintainable video streaming application on AWS.

5.6 TradeOffs Revisited

- **TR 8.3:** Select vendors charging lesser prices for higher rated/more popular services.
- **TR 8.4:** Constantly keep checking and updating the services for cost optimization possibilities.
- **TR 17.1:** Develop a multi-cloud strategy to prevent dependency on a single vendor.
- **BR 9:** Choose providers with environmental certifications.
    - **TR9.1:** Select cloud providers with recognized environmental certifications (like ISO 14001).
    - **TR9.2:** Ensure that data centers adhere to green computing practices.
    - **TR9.3:** Choose data centers with optimal climatic conditions to minimize cooling efforts.
    - **TR9.4:** Choose data centers with access to green energy resources to minimize electricity usage.

As the **provider comparison** table in **section 3.2** suggests, using services from a combination of multiple cloud providers would result in a more cost effective solution. Similarly, from the original list of TRs in **section 2.2**, the requirement to avoid vendor lock-in is also clearly stated. However, in our project, we have decided to move forward with a single cloud service provider in order to prioritize other benefits we gain from doing so.

- **Reduced complexity of maintenance:** Utilizing a single cloud provider streamlines management and integration efforts. It simplifies processes related to deployment, monitoring, and maintenance, requiring fewer resources and expertise for managing a unified environment.
- **Possible volume discounts:** Committing to a single cloud provider could lead to the possibility of certain volume discounts and better pricing arrangements for the services we use. Moreover, there is also the possibility of being a priority consumer when our entire application runs on a single cloud.

- **Optimized resource utilization:** Sticking with a single provider reduces complexities related to resource allocation, capacity planning, and load balancing across different cloud platforms, ensuring efficient resource usage.
- **Reduced challenges in integration:** Using the services from a single provider drastically reduces challenges in integrating the various components of our design and their interoperability. Multi-cloud setups might lead to additional costs to overcome difficulties in the interoperability of services and components.
- **Security measures:** Managing security, compliance and authentication is straight-forward while using a single cloud provider.

In a many-many relationship, the requirements being traded off are tabulated below. Note that all of the requirements in column 1 are traded off for all of those in column 2. The justification for prioritizing the requirements in column 2 is provided in column 3.

Requirements traded off	Requirements traded off for	Justification
TR 8.3: Select vendors charging lesser prices for higher rated/more popular services.	**TR14.1:** Implement open standards and APIs for better interoperability.	Working within a single cloud provider's environment means utilizing consistent APIs, interfaces, and management tools across various services. This consistency facilitates smoother communication and interaction between different services and reduces compatibility issues.
TR 8.4: Constantly keep checking and updating the services for cost optimization possibilities.	**TR14.2:** Establish integration frameworks for connecting with external systems.	Many cloud providers offer pre-built integrations and solutions that are designed to work together within their ecosystem. These integrated solutions allow for easier data exchange, communication, and interoperability between various services without requiring complex configurations or middleware.
TR 17.1: Develop a multi-cloud strategy to prevent dependency on a single vendor.	**TR16.1:** Conduct vendor assessments based on industry reputation and reviews.	AWS (Amazon Web Services) is considered a well-reputed cloud provider due to its extensive global infrastructure, comprehensive suite of services spanning computing, storage, databases, AI/ML, and IoT, high availability, reliability, strong security

		measures (such as encryption, IAM, and compliance certifications), continuous innovation, scalability, cost-effectiveness through pay-as-you-go pricing, and robust customer support. Its market leadership, vast customer base across industries, and proven track record in enabling businesses of all sizes contribute to its reputation as a reliable and versatile cloud platform.
BR 9: Choose providers with environmental certifications.	**TR16.2:** Verify vendors' compliance with industry standards and certifications.	AWS regularly undergoes third-party audits and certifications to validate its adherence to industry standards and compliance frameworks. Customers can leverage these compliant services to build applications and solutions while meeting regulatory requirements within their respective industries. Some of the certifications obtained by AWS are: ■ **SO/IEC 27001** ■ **SOC Reports** ■ **HIPAA** ■ **HITECH** ■ **PCI DSS** ■ **FISMA** ■ **FedRAMP** ■ **CJIS** ■ **GDPR** ■ **APAC Data Residency Compliance**
	TR16.3: Consider vendors who are compliant with the security guidelines provided by NIST.	By offering a wide array of security services, controls, and tools, AWS assists customers in implementing and maintaining security measures that align with NIST's security guidelines. AWS regularly undergoes audits and assessments to validate its adherence to industry standards, including NIST's

		security recommendations, providing customers with a secure and compliant cloud environment.
	TR13.1: Create secure data backups and store them in geographically diverse locations.	Using a single cloud provider like AWS offers centralized control and management of backup processes. It enables streamlined configuration, monitoring, and management of backup procedures, reducing complexities associated with managing multiple backup solutions across different providers. AWS offers a range of backup and data protection services like AWS Backup, Amazon S3 Versioning, and AWS Storage Gateway that allow for automated backups across various AWS services and resources. These services can be centrally managed and configured for automated backups, ensuring consistency and efficiency in backup procedures.
	TR13.2: Implement data lifecycle management to optimize storage costs.	Utilizing a single cloud provider like AWS allows for centralized management of data lifecycle policies across different storage services and resources. AWS offers a suite of data lifecycle management tools and services like Amazon S3 Object Lifecycle Management, Amazon Glacier, and AWS Storage Gateway. These services allow users to define policies for data storage tiers, automated data movement between storage classes, and data expiration, aiding in cost optimization.
	TR13.3: Implement encryption for data at rest and in transit to ensure data security.	Service providers typically offer built-in encryption features across their services. These include encryption options for data transfer, such as

		SSL/TLS, available within various services offered by the provider. AWS services inherently support encryption for data transmission. Services like Amazon S3, Amazon RDS, Amazon Redshift, and Amazon EC2 offer encryption options to enable secure data transfer over the network.
	TR13.4: Implement authentication systems to prevent unauthorized access to backup data.	Employing a single cloud provider's IAM services allows for centralized management of user identities, roles, and permissions across all services within the provider's ecosystem. This centralized control simplifies user access management and ensures consistency in authentication mechanisms across various services. Moreover, a single cloud provider generally implements standardized authentication protocols across its services, ensuring uniformity in authentication methods. This consistency simplifies the implementation and enforcement of strong authentication mechanisms, such as multi-factor authentication (MFA), to prevent unauthorized access.
	TR2.1: A secure authentication mechanism must identify tenants with 100% accuracy.	Different cloud providers may employ their own identity management systems and authentication mechanisms, leading to fragmented tenant management. Each cloud provider may store user data separately within their environments. Without a unified repository for user data, it becomes complex to accurately consolidate and synchronize user information across multiple clouds. This fragmentation can hinder the accuracy of tenant identification. Therefore, using a single cloud service provider can nullify the potential impact caused by using

		multiple providers on the accuracy of tenant identification.
	TR2.2: Identifiers and session keys may be used to monitor individual tenants' usage.	Different cloud providers offer their own monitoring and analytics tools. Using multiple providers leads to the use of disparate monitoring tools and dashboards, making it challenging to have a unified view of tenant usage across all platforms, thereby potentially affecting the predictions gathered from monitoring tenant usage.

5.7 Discussion of an Alternate Design

**** SKIPPED ****

6 Kubernetes Experimentation

6.1 Experiment Design

Selected TRs:

- TR1: Implement automatic scaling for Kubernetes Pods based on resource utilization.
- TR19.3: Implement global load balancing for directing user requests to the nearest available servers based on some rules.

Experiment Details:

1. Objective:

Validate the automatic scaling capability of Kubernetes Pods using the Horizontal Pod Autoscaler (HPA).

2. Environment Configuration:

- Kubernetes Cluster with HPA enabled.

```
jayvishaalj@Jays-Air cc % kubectl get po -A
NAMESPACE              NAME                                         READY   STATUS    RESTARTS         AGE
default                php-apache-5bdbb8dbf8-5gl7g                  1/1     Running   1 (16d ago)      16d
kube-system            coredns-5d78c9869d-zf979                     1/1     Running   7 (16d ago)      21d
kube-system            etcd-minikube                                1/1     Running   7 (16d ago)      21d
kube-system            kube-apiserver-minikube                      1/1     Running   7 (16d ago)      21d
kube-system            kube-controller-manager-minikube             1/1     Running   7 (16d ago)      21d
kube-system            kube-proxy-skg4q                             1/1     Running   7 (16d ago)      21d
kube-system            kube-scheduler-minikube                      1/1     Running   7 (16d ago)      21d
kube-system            metrics-server-7746886d4f-8cb74              1/1     Running   8 (16d ago)      21d
kube-system            storage-provisioner                          1/1     Running   15 (106s ago)    21d
kubernetes-dashboard   dashboard-metrics-scraper-5dd9cbfd69-4qph6   1/1     Running   7 (16d ago)      21d
kubernetes-dashboard   kubernetes-dashboard-5c5cfc8747-t4t9p        1/1     Running   14 (105s ago)    21d
jayvishaalj@Jays-Air cc %
```

- Application with an HPA configured, allowing a maximum of 10 Pods.

```
jayvishaalj@Jays-Air cc % kubectl get deployments
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
php-apache   1/1     1            1           16d
jayvishaalj@Jays-Air cc %
```

```
jayvishaalj@Jays-Air cc % kubectl get pods
NAME                          READY   STATUS    RESTARTS      AGE
php-apache-5bdbb8dbf8-5gl7g   1/1     Running   1 (16d ago)   16d
jayvishaalj@Jays-Air cc %
```

```
jayvishaalj@Jays-Air cc % kubectl get pods -n kube-system
NAME                               READY   STATUS    RESTARTS         AGE
coredns-5d78c9869d-zf979           1/1     Running   7 (16d ago)      21d
etcd-minikube                      1/1     Running   7 (16d ago)      21d
kube-apiserver-minikube            1/1     Running   7 (16d ago)      21d
kube-controller-manager-minikube   1/1     Running   7 (16d ago)      21d
kube-proxy-skg4q                   1/1     Running   7 (16d ago)      21d
kube-scheduler-minikube            1/1     Running   7 (16d ago)      21d
metrics-server-7746886d4f-8cb74    1/1     Running   8 (16d ago)      21d
storage-provisioner                1/1     Running   15 (4m58s ago)   21d
jayvishaalj@Jays-Air cc %
```

- Locust installed for workload generation.

3. Experiment Steps:

a. **Deploy the application with HPA settings:**

```
Unset
kubectl apply -f app.yaml
```

b. **Enable HPA for the deployment:**

1. Update your deployment YAML to include resource requests and limits.
2. Create an autoscale HPA yaml configuration file and apply it to kubectl.
3. Example snippet:

```
Unset

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
 name: cpu-autoscale
spec:
 scaleTargetRef:
  apiVersion: apps/v1
  kind: Deployment
  name: php-apache
 minReplicas: 1
 maxReplicas: 10
 metrics:
```

```
- type: Resource
  resource:
    name: cpu
    target:
      type: Utilization
      averageUtilization: 50
```

```
jayvishaalj@Jays-Air cc % kubectl get hpa
NAME         REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   0%/50%    1         10        1          16d
jayvishaalj@Jays-Air cc %
```

c. **Start Locust workload with 1 user and 1 rps**

d. **Observe scaling and performance under initial load:**

- Use kubectl get hpa to monitor HPA status.
- Check Kubernetes Events for scaling events.
- Monitor Pod metrics: kubectl top pod.

```
jayvishaalj@Jays-Air cc % kubectl get hpa --watch
NAME         REFERENCE                TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   163%/50%   1         10        4          16d
```

e.**Increase Locust workload to 10 users with a spawn rate of 2 users per second**

f. **Observe scaling and performance under increased load:**

- Continue monitoring HPA status, events, and Pod metrics.
- Verify that Pods scale up to handle increased load.

```
jayvishaalj@Jays-Air cc % kubectl get hpa --watch
NAME         REFERENCE                TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   172%/50%   1         10        3          16d
php-apache   Deployment/php-apache   172%/50%   1         10        6          16d
php-apache   Deployment/php-apache   172%/50%   1         10        7          16d
php-apache   Deployment/php-apache   101%/50%   1         10        7          16d
```

g. **Monitor HPA logs, Kubernetes Events, and Pod metrics during the experiment**

```
jayvishaalj@Jays-Air cc % kubectl get hpa --watch
NAME         REFERENCE               TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   96%/50%   1         10        10         16d
php-apache   Deployment/php-apache   96%/50%   1         10        10         16d
php-apache   Deployment/php-apache   20%/50%   1         10        10         16d
php-apache   Deployment/php-apache   0%/50%    1         10        10         16d
php-apache   Deployment/php-apache   0%/50%    1         10        10         16d
php-apache   Deployment/php-apache   0%/50%    1         10        10         16d
php-apache   Deployment/php-apache   0%/50%    1         10        4          16d
php-apache   Deployment/php-apache   2%/50%    1         10        4          16d
php-apache   Deployment/php-apache   2%/50%    1         10        4          16d
php-apache   Deployment/php-apache   8%/50%    1         10        1          16d
php-apache   Deployment/php-apache   17%/50%   1         10        1          16d
```

4. Expected Outputs:

- Successful scaling of Pods based on increased load.
- No failures observed during the scaling process.
- Pods automatically scale down after the load is reduced.

6.2 Workload Generation with Locust

Intention to Use Locust:

Locust is chosen as the workload generator for several reasons:

1. **User-Friendly:** Locust provides a user-friendly approach to load testing, allowing testers to define user behaviors using Python code.
2. **Scalability:** It supports distributed load testing, enabling the simulation of a large number of users.
3. **Open Source:** Being an open-source tool, Locust is cost-effective and easily accessible for experimentation.
4. **Versatility:** Locust supports a wide range of use cases, making it suitable for testing various types of applications.
5. **Realistic Scenarios:** The ability to define custom scenarios and user behaviors allows for the creation of realistic testing scenarios.

Locust Configuration:

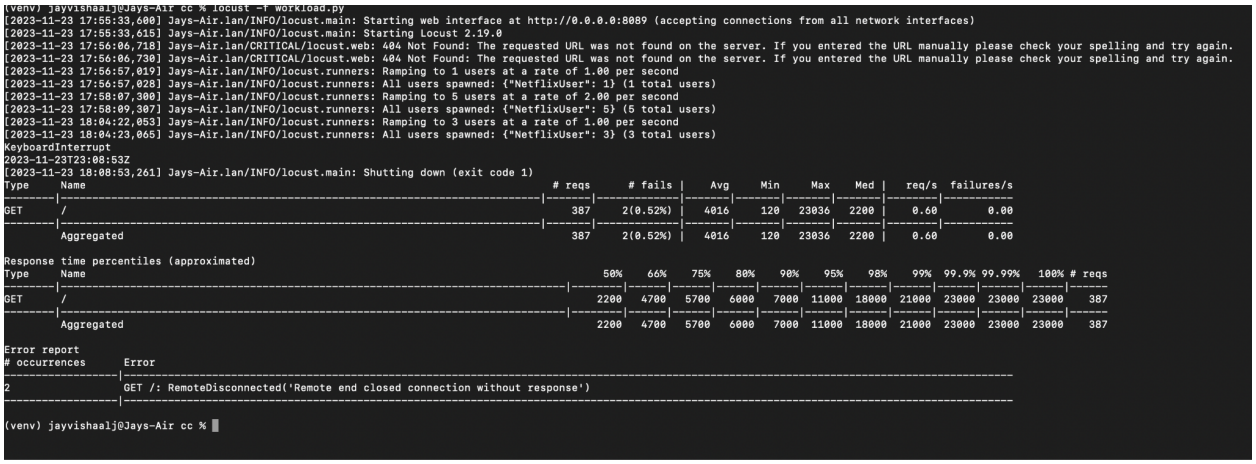
1. **Installation:**
    - Install Locust using the following command:
    - pip install locust
2. **Locustfile Configuration:**
    - Create a Python script, commonly referred to as a Locustfile, to define the workload scenario.
    - Example Locustfile (locustfile.py):

```python
from locust import HttpUser, task, between

class NetflixUser(HttpUser):
    wait_time = between(1, 5)  # Time between requests in seconds

    @task
    def load_site(self):
        # Endpoint to simulate loading the site
        self.client.get("/")
```

3. **Explanation of Locustfile:**
    - MyUser: Represents a simulated user in the Locust scenario.
    - wait_time: Defines the time between task execution (1 to 2 seconds in this example).
    - @task: Decorator indicating the task to be performed by the user.
    - my_task: Represents a simple task of making an HTTP GET request to the root (“/”) endpoint.
4. **Running Locust:**
    - Execute Locust with the specified Locustfile:
    - **locust -f locustfile.py**
    - Access the Locust web interface (typically, http://localhost:8089) to start the test, set parameters, and monitor results.

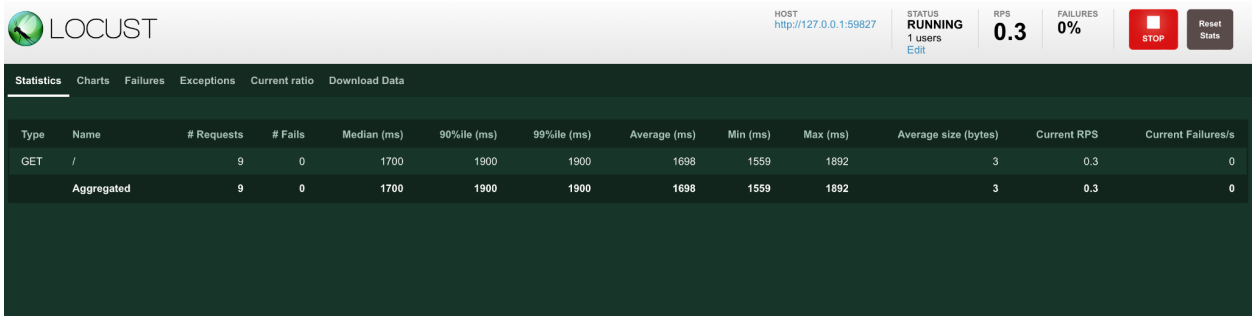


Experimentation with Different Load Combinations:

1. **Low Load (1 User, 1 Request Per Second):**
    - Simulate a minimal load to observe initial scaling behavior.
    - Command:
    - **locust -f locustfile.py --users 1 --spawn-rate 1**
2. **Increased Load (10 Users, 2 Requests Per Second):**
    - Apply increased load to assess scaling under higher demand.
    - Command:
    - **locust -f locustfile.py --users 10 --spawn-rate 2**
3. **Exploration of Scaling and Performance:**
    - Monitor Kubernetes Events, HPA logs, and Pod metrics.
    - Check the Locust web interface for real-time insights into user behavior and requests.
4. **Gradual Load Reduction:**
    - Gradually reduce the Locust workload to observe automatic scaling down.
    - Command:
    - **locust -f locustfile.py --users 1 --spawn-rate 1**
5. **Monitoring and Analysis:**
    - Utilize kubectl commands to check HPA status, Pod metrics, and logs.
    - Analyze Locust results and logs for insights into the system's behavior.

Validation:

- Ensured that scaling aligns with the defined TR for automatic scaling based on resource utilization.
- Validated that the system responds effectively to changing workloads.
- Confirmed that scaling events are recorded and Pods are created or terminated accordingly.



Conclusion:

The use of Locust provides a flexible and controlled approach to generating workloads for Kubernetes experimentation. By varying user counts and request rates, the objective is to observe the system's ability to scale dynamically and maintain stability under different load conditions. This approach allows for a comprehensive assessment of the Kubernetes cluster's automatic scaling capabilities.

6.3 Analysis of the Results

Observations:

Initial Load (1 user, 1 rps):

- Pods should scale up to handle the load.
- No failures should be observed.

Increased Load (10 users, 2 rps):

- Pods should scale up further to accommodate the increased load.
- Monitor resource utilization and scaling events.
- Verify that the application handles the load without failures.

Load Reduction:

- Gradually reduce the Locust workload.
- Observe the automatic scaling down of Pods.
- Ensure that no unnecessary Pods are retained.

Validation:

- Confirmed that the observed scaling behavior aligns with the defined TR (automatic scaling).
- Verified that scaling is based on resource utilization and responds appropriately to varying workloads.
- Ensured that the system maintains stability under different load conditions

Locust Report

Locust Test Report

During: 11/23/2023, 5:58:07 PM - 11/23/2023, 6:00:58 PM

Target Host: http://127.0.0.1:59827

Script: workload.py

Request Statistics

Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/	113	0	4357	1582	9129	3	0.7	0.0
	Aggregated	**113**	**0**	**4357**	**1582**	**9129**	**3**	**0.7**	**0.0**

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	/	4800	5300	6000	6200	6800	7000	8700	9100
	Aggregated	**4800**	**5300**	**6000**	**6200**	**6800**	**7000**	**8700**	**9100**

Charts

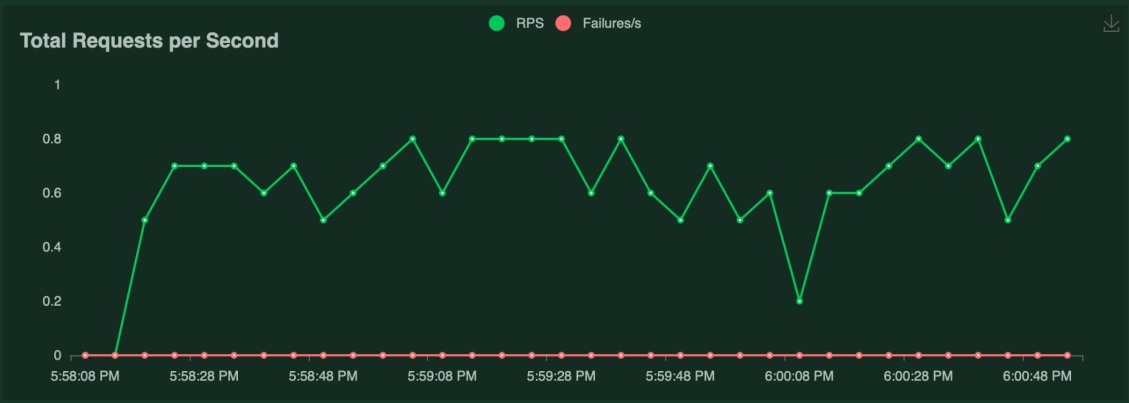


FIg 6.3.1 Total Request per seconds Graph represents the number of requests spawned by various users per second

Final ratio

Ratio per User class

- 100.0% NetflixUser
  - 100.0% load_site

Total ratio

- 100.0% NetflixUser
  - 100.0% load_site

Fig 6.3.3 Response Time for each request is being shown as a graph
FIg 6.3.4 Number of users spawned per second

Experiment on second selected TR19.3:

Experiment Setup:

1. **Environment Configuration:**
    - Use Minikube for a local Kubernetes cluster.
2. **Service Deployment:**
    a. Create Docker Images:
        - For simplicity, let's assume two simple web applications (App A and App B) with their Dockerfiles.
            - Dockerfile for App A:

```
Unset
# App 1
FROM nginx
RUN echo "This is App A" > /usr/share/nginx/html/index.html
EXPOSE 80
```

- Dockerfile for App B:

```
Unset
# App 2
FROM nginx
RUN echo "This is App B" > /usr/share/nginx/html/index.html
EXPOSE 80
```

- Build Docker images for both applications.

```
Unset
docker build -t app-a-image -f Dockerfile.appA .
docker build -t app-b-image -f Dockerfile.appB .
```

b. Deploy Applications:

- Deploy two Kubernetes Deployments for App A and App B.

```
Unset
apiVersion: apps/v1
kind: Deployment
metadata:
 name: app-a
spec:
 replicas: 1
 selector:
  matchLabels:
   app: app-a
 template:
  metadata:
   labels:
    app: app-a
  spec:
   containers:
   - name: app-a
    image: jvjdockerregistry/csc547:app-a
    imagePullPolicy: IfNotPresent
---
apiVersion: apps/v1
kind: Deployment
metadata:
 name: app-b
```

```
spec:
 replicas: 1
 selector:
  matchLabels:
   app: app-b
 template:
  metadata:
   labels:
    app: app-b
  spec:
   containers:
   - name: app-b
    image: jvjdockerregistry/csc547:app-b
    imagePullPolicy: IfNotPresent
```

c. Expose Services:

- Expose the services to make them accessible within the cluster.

Unset

```
apiVersion: v1
kind: Service
metadata:
 name: app-a
spec:
 selector:
  app: app-a
 ports:
  - protocol: TCP
   port: 80
   targetPort: 80
---
apiVersion: v1
kind: Service
metadata:
 name: app-b
spec:
 selector:
  app: app-b
 ports:
  - protocol: TCP
```

```
    port: 80
    targetPort: 80
```

```
Unset
kubectl apply -f app-deployments.yaml
kubectl apply -f app-services.yaml
```

Execution of the above commands:

```
jayvishaalj@Jays-Air TR2 % kubectl apply -f app1_2_deployments.yaml
deployment.apps/app-a created
deployment.apps/app-b created
jayvishaalj@Jays-Air TR2 % kubectl apply -f app1_2_services.yaml
service/app-a created
service/app-b created
jayvishaalj@Jays-Air TR2 %
```

```
jayvishaalj@Jays-Air TR2 % kubectl get services
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
app-a        ClusterIP   10.99.237.0     <none>        80/TCP    8s
app-b        ClusterIP   10.105.111.55   <none>        80/TCP    8s
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP   65m
jayvishaalj@Jays-Air TR2 % kubectl get deploy
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
app-a   1/1     1            1           79s
app-b   1/1     1            1           79s
jayvishaalj@Jays-Air TR2 % kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
app-a-6fff849b6c-jkbsm   1/1     Running   0          82s
app-b-8b4fdb5b8-qsw8d    1/1     Running   0          82s
jayvishaalj@Jays-Air TR2 %
```

3. Create Ingress Resource:
    - Apply the Ingress resource with routing rules.

```
Unset
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: my-ingress
```

```
spec:
 rules:
 - host: my-app.local
  http:
   paths:
   - path: /appA
    pathType: Prefix
    backend:
     service:
      name: app-a
      port:
       number: 80
   - path: /appB
    pathType: Prefix
    backend:
     service:
      name: app-b
      port:
       number: 80
```

- Apply the Ingress resource.

```
Unset
kubectl apply -f ingress-resource.yaml
```

```
jayvishaalj@Jays-Air loadbalancer % kubectl apply -f ingress.yaml
ingress.networking.k8s.io/my-ingress created
jayvishaalj@Jays-Air loadbalancer %
```

4. Update /etc/hosts:
    - Add the following entry to /etc/hosts to simulate a custom domain.

```
Unset
echo "$(minikube ip) my-app.local" | sudo tee -a /etc/hosts
```

5. Access Apps:
    - Open a web browser and access:

- http://my-app.local/appA should display content from App A.
- http://my-app.local/appB should display content from App B.

6. Observations:
    - Observe the content served by each application, confirming that traffic routing based on Ingress rules is successful.

This experiment demonstrates Traffic Routing with Kubernetes Ingress in a local Minikube environment, validating the targeted TR19.3.

```
jayvishaalj@Jays-Air loadbalancer % kubectl get svc
NAME            TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes      ClusterIP      10.96.0.1        <none>        443/TCP        32m
load-balancer   LoadBalancer   10.104.167.119   127.0.0.1     80:31446/TCP   6m46s
service-a       ClusterIP      10.106.54.105    <none>        80/TCP         7m55s
service-b       ClusterIP      10.101.242.41    <none>        80/TCP         7m51s
jayvishaalj@Jays-Air loadbalancer %
```

Fig 6.3.6 Load balancer running along with two services

```
jayvishaalj@Jays-Air loadbalancer % kubectl get pods
NAME                         READY   STATUS    RESTARTS   AGE
service-a-64788bf454-fbm58   1/1     Running   0          11m
service-a-64788bf454-hwx69   1/1     Running   0          11m
service-b-7f7b7c876b-5x4kf   1/1     Running   0          10m
service-b-7f7b7c876b-6zlgn   1/1     Running   0          10m
jayvishaalj@Jays-Air loadbalancer %
```

Fig 6.3.7 Displays all the pods and its status and running count

7 Ansible Playbooks

7.1 Description of Management Tasks

****SKIPPED****

7.2 Playbook Design

****SKIPPED****

7.3 Experiment Runs

****SKIPPED****

8 Demonstration

****SKIPPED****

9 Comparisons

****SKIPPED****

10 Conclusions

10.1 The Lessons Learned

With very little experience in cloud deployments, taking up this project was an excellent opportunity to explore uncharted territories. Stepping into the shoes of a cloud architect paved the way for us to learn several key lessons on the core of cloud computing.

Understanding Business and Technical Needs: Having worked as software engineers before, it was easy to come up with TRs once we had our BRs ready. But the initial process of thinking of BRs took up the lion's share of the overall time spent on our project. Looking at architecture design from the client's perspective was totally new to us, but the task was a challenge we took up in good spirit and managed to grasp the business needs comprehensively.

Choosing the Right Cloud Provider: Evaluating the basis on which to choose our cloud provider was a very interesting task. We learnt a lot in the process of comparing various providers; it was highly intriguing to learn about the differences in the way each provider described their services and their pricing models.

Service Selection: The process of selecting the services we would be using proved to be the perfect opportunity to browse and learn about the myriads of services that cloud providers are offering. There was a very significant knowledge gain in this section.

Designing the First Draft: Having zeroed in on the services we would be using, we were clueless about how to interconnect them to arrive at a design. To overcome that hurdle, we read and understood copious numbers of architecture design reference diagrams provided by AWS and finally understood how to connect our individual components. The validation subsection helped us identify flaws in our design and we were able to improve it based on our observations.

The Second Design Draft: This section gave us the chance to iteratively improve on our design and clearly differentiate between our milestones. Moreover, studying up on AWS WAF was a rich source of information for us, which completely changed our approach to the architecture design.

Understanding Tradeoffs: While it was easy to highlight conflicting requirements in section 2 of the project, it was not so easy to identify the requirements which we had traded off after completing the design. We traversed through each component of our design and evaluated the requirements which were lacking in those. Additionally, we came to the understanding that tradeoffs need not have a 1-1 relationship. For a collective set of requirements which we prioritized, we were able to identify a collection of requirements which we traded off and came up with a many-many relationship for the same.

Kubernetes Experiment: Having identified the TRs which we would be able to demonstrate using k8s, designing and performing the experiments gave us hands-on experience in working with tools like k8s and Locust.

Flexibility and Evolution: This project was the perfect example to prove that the design process is an iterative one. No section is complete until the project is complete. We had to iteratively go back to previous sections and make changes upon identifying flaws in our design while working on later sections. This taught us how to be flexible with our work, and how to evolve with changing requirements.

10.2 Possible Continuation of the Project

Cloud services and providers are constantly evolving due to technological advancements, market dynamics, and changing user requirements. What may appear as a superior service or provider today could quickly become outdated or less effective in the future. Cloud architectures inherently allow for continuous improvement by being scalable, adaptable, and receptive to updates. They can integrate new services, optimizations, and technologies without necessitating a complete overhaul. Through regular updates, monitoring, and feedback analysis, we can identify areas for enhancement, ensuring the architecture remains aligned with evolving standards, security needs, and business requirements. Cloud architectures' flexibility enables the incorporation of emerging technologies and methodologies, enabling them to evolve in tandem with the ever-changing landscape of cloud services and business demands.

11 References

[1] YV and YP, “ECE547/CSC547 class notes”.

[2] https://github.com/tetratelabs/getmesh and [peter jausovec](https www.peterj.dev). “Hands-on Walkthrough of Traffic Management in Istio Part 1.” *Tetrate Istio Distro*, 23 Feb. 2021, tetratelabs.io/blog/istio-traffic-management-walkthrough-p1.

[3] “Horizontal Pod Autoscaling.” *Kubernetes*, 18 May 2023, kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale.

[4] More, Yogesh. “Load Balance Microservices Using Kubernetes’ Minikube.” *Medium*, 8 Nov. 2022, medium.com/globant/load-balance-microservices-using-kubernetes-minikube-88b78dae4796.

[5] “Re:Invent 2016: Neil Hunt of Netflix Discusses How AWS Supports Deployment of New Features and Tools.” *Amazon Web Services, Inc.*, aws.amazon.com/solutions/case-studies/netflix-case-study.

[6] “Live Streaming on AWS | AWS Solutions | AWS Solutions Library.” *Amazon Web Services, Inc.*, aws.amazon.com/solutions/implementations/live-streaming-on-aws.

[7] “Building Serverless Video Workflows (58:42).” *Amazon Web Services, Inc.*, aws.amazon.com/cloudfront/streaming.

[8] PCMag. “Definition of Doherty Threshold.” *PCMAG*, www.pcmag.com/encyclopedia/term/doherty-threshold.