Cloud Computing Technology

CSC/ECE 547 - Fall 2023

Ad-Click Event Aggregation

Mohit Soni, Sukhad Joshi
(msoni, sjoshi32)
North Carolina State University

Plagiarism Declaration

We, the team members, understand that copying & pasting material from any source in our project is an allowed practice; we understand that not properly quoting the source constitutes plagiarism.

All team members attest that we have properly quoted the sources in every sentence/paragraph we have copy & pasted in our report. We further attest that we did not change words to make copy & pasted material appear as our work.

Table of Contents

1. Introduction
   - 1.1. Motivation
   - 1.2. Executive Summary
2. Problem Description
   - 2.1. The problem
   - 2.2. Business Requirements
   - 2.3. Technical Requirements
   - 2.4. Trade Offs
3. Provider Selection
   - 3.1. Criteria for choosing a provider
   - 3.2. Provider Comparison
   - 3.3. The final selection
   - 3.4. The list of services offered by the winner
4. The First design draft
   - 4.1. The basic building blocks of the design
   - 4.2. Top-level, informal validation of the design
   - 4.3. Action items and rough timeline
5. The second design
   - 5.1. Use of the Well-Architected Framework
   - 5.2. Discussion of Pillars
   - 5.3. Use of CloudFormation diagrams
   - 5.4. Validation of the design
   - 5.5. Design Principles And Best Practices Used
   - 5.6. Tradeoffs Revisited
   - 5.7. Discussion Of An Alternate Design
6. Kubernetes experimentation
   - 6.1. Experiment Design
   - 6.2. Workload generation with Locust
   - 6.3. Analysis Of The Results

7. Ansible Playbooks

8. Demonstration

9. Comparisons

10. Conclusion
    10.1. The Lessons Learned
    10.2. Possible Continuation Of The Project

11. References

1. Introduction

1.1 Motivation

Our team aimed to select a subject for our cloud computing project that would enable us to apply the principles we've acquired during our cloud computing course. In terms of the project's scope, we aspired to design a solution for a practical real-world scenario that could be implemented within a reasonable timeframe, keeping in mind the professor's repeated reminder that this course is not focused on devOps.

1. 2 Executive Summary

In today's digital landscape, advertising campaigns are a critical component of any business's marketing strategy. To optimize these campaigns and maximize ROI, advertisers need accurate and real-time insights into ad-click events. However, many organizations face challenges in aggregating, managing, and deriving meaningful insights from the vast amount of ad-click data generated across various platforms. With the rise of Facebook, YouTube, TikTok, and the online media economy, digital advertising is taking an ever-bigger share of the total advertising spending. As a result, tracking ad click events is very important.

In response to the growing demand for data-driven advertising analytics and the need for efficient event tracking, our team proposes the development of an Ad-Click Event Aggregation SaaS (Software as a Service) Cloud Application for this semester's final project. This innovative solution aims to streamline and enhance the process of gathering, analyzing, and visualizing ad-click data for businesses and marketers.

Our Ad-Click Event Aggregation SaaS Cloud Application will provide a comprehensive solution to address these challenges. Key features and components of our application include: Data Collection, Real-time processing, Data Storage, Data Security, Analytics and Reporting.

2. Problem Description

2.1 The problem

To design a SaaS Ad-Click Event aggregation service that allows users to keep track of all the ads that they have published over the internet. The service should have minimum latency and provide an analytics dashboard for analytics.

2.2 Business Requirements

2.2.1 Cloud Architect

- BR 1: The system must support the **ingestion and integration of data from multiple sources**.
- BR 2: The architecture should enable **real-time processing** of advertising data to provide timely insights and decision-making capabilities
- BR 3: The system must provide **storage solutions** for both raw and aggregated data, allowing for efficient data retrieval and analysis.
- BR 4: The architecture should support extended **data retention** periods to meet business needs and compliance requirements.
- BR 5: The system should offer **multi-tenancy** capabilities, allowing multiple users or organizations to securely access resources while maintaining isolation
- BR 6: The architecture must be **highly scalable** to accommodate increasing workloads and data volumes as the business grows.
- BR 7: The system should deliver **high performance**, ensuring fast data processing and response times for critical operations.
- BR 8: The architecture must ensure **high availability**, minimizing downtime and disruptions through redundancy and failover mechanisms.
- BR 9: **Security measures and compliance standards** should be in place to protect data, ensuring regulatory requirements are met.
- BR 10: The system should provide **robust user management and authentication** features to control access and maintain data security.
- BR 11: Comprehensive **monitoring and alerting capabilities** should be implemented to proactively track the cost and status of all resources.
- BR 12: The architecture must include **disaster recovery and backup** solutions to safeguard data and maintain business continuity.

- BR 13: **Comprehensive documentation and ongoing support** services should be available to assist users and developers.
- BR 14: The architecture should strive for **cost efficiency** by optimizing resource utilization and minimizing operational expenses.
- BR 15: The system should ensure **low latency** for data processing and access to support real-time applications.
- BR 16: The system must consistently deliver accurate results whenever they are needed, demonstrating a **high level of reliability.**

2.2.2 Software Architect

- BR 17: The system must support **custom reporting and analytics** features, allowing users to generate tailored reports and gain insights into data.
- BR 18: The system should incorporate **attribution modeling** techniques to analyze and attribute the impact of various factors on desired outcomes.
- BR 19: The software architecture should include robust **fraud detection and prevention** mechanisms to safeguard against fraudulent activities.
- BR 20: The system must have comprehensive **billing and subscription management** functionalities to handle customer billing, subscriptions, and payment processing.
- BR 21: The software should facilitate **seamless integration with external tools**, APIs, and services to enhance its functionality and interoperability.

2.3 Technical Requirements [1]

- TR 1.1: The system should facilitate data ingestion from diverse sources by accommodating multiple data formats (e.g., CSV, JSON, and XML) and protocols, including REST APIs and batch data uploads.
- TR 1.2: It should provide data quality checks and validation during the ingestion process, ensuring that incoming data meets predefined standards before processing.
- TR 1.3: The system must transform different data formats to a unified format (e.g. CSV) for further processing.
- TR 2.1: The real-time processing component must have low-latency capabilities, with the ability to process incoming ad data within milliseconds of receipt.
- TR 2.2: It should support event-driven architectures, enabling immediate reactions to injection of data.
- TR 2.3: Real-time data processing should include automated anomaly detection and alerting mechanisms to identify and respond to abnormal data patterns promptly.
- TR 3.1: The system should employ a distributed and redundant storage infrastructure that ensures data durability and high availability.
- TR 3.2: It must support efficient data compression techniques (e.g. lossy and lossless) to reduce storage costs while maintaining data integrity.
- TR 3.3: Aggregated data should be stored in a format optimized for analytics, such as columnar storage or data warehouses, to facilitate fast querying and reporting.
- TR 4.1: The system should have well defined data retention policies for different types of data, taking into account regulatory requirements and business needs.
- TR 4.2: The data archiving process should include automated indexing to facilitate easy retrieval and management of archived data.
- TR 4.3: The system should provide mechanisms for data versioning and auditing to track changes and ensure data integrity over extended retention periods.
- TR 5.1: The system must support the creation of multiple isolated tenant accounts, each representing a different advertiser or client, with distinct user access controls, data segregation, and configuration settings.
- TR 5.2: The system must implement a secure and unique tenant identification mechanism, such as a unique identifier or token, to distinguish and associate users or entities with their respective tenant profiles and data.
- TR 6.1: The system architecture should be designed for horizontal scalability, enabling the addition of new processing nodes, servers, or cloud resources dynamically.
- TR 6.2: Scalability testing should be conducted regularly to identify system bottlenecks and resource limitations, with automatic scaling triggers and policies in place to handle increased workloads.

[1] This section is written with the assistance of LLM & AWS Framework.

- TR 6.3: It should utilize containerization and orchestration technologies to simplify the deployment and scaling of microservices.
- TR 7.1: The system should employ distributed caching mechanisms to accelerate data retrieval and reduce the load on backend data stores.
- TR 7.2: It must support parallel processing of complex queries and calculations to ensure rapid response times for ad performance analytics.
- TR 7.3: Performance testing should be conducted under simulated high-load scenarios to validate the system's ability to maintain low-latency responses even during peak usage.
- TR 8.1: The system architecture should include redundant components such as data replicas and load balancers to ensure uninterrupted service availability in case of failures.
- TR 8.2: It should implement geographically distributed data centers or cloud regions to provide geographic redundancy and minimize the impact of regional outages.
- TR 8.3: High availability testing and disaster recovery drills should be conducted regularly to validate the system's ability to recover gracefully from failures and minimize downtime.
- TR 9.1: The system must enforce role-based access control (RBAC) to restrict user access to sensitive data and functions based on their roles and responsibilities.
- TR 9.2: It should implement data encryption at rest and in transit to protect data confidentiality and integrity.
- TR 9.3: The system must be designed and configured to fully adhere to regulatory laws, including GDPR (General Data Protection Regulation), with a focus on data privacy and compliance.
- TR 10.1: The system must support multi-factor authentication (MFA) to enhance user account security and prevent unauthorized access.
- TR 10.2: User authentication should integrate with industry-standard identity providers (e.g., OAuth, LDAP) to streamline user management and authentication processes.
- TR 10.3: User account provisioning and deprovisioning should be automated, with support for user role assignment and password policy enforcement.
- TR 11.1: The system should implement comprehensive monitoring for indicators such as CPU utilization, data storage size, and other resources metrics.
- TR 11.2: It must have automated alerting mechanisms that notify administrators and relevant stakeholders of system anomalies, errors, or performance degradation.
- TR 11.3: The system should integrate with a centralized logging and log analysis platform to facilitate proactive issue identification and debugging.
- TR 12.1: A disaster recovery plan should be documented and regularly updated, including procedures for data backup, system recovery, and failover to a secondary data center or cloud region.
- TR 12.2: Backup and recovery tests should be conducted periodically to ensure data integrity and the effectiveness of disaster recovery processes.

- TR 12.3: The system should support automated backup scheduling and versioning, with data retention policies aligned with business requirements.
- TR 13.1: Comprehensive system documentation, including user manuals, API documentation, and architecture diagrams, should be maintained and accessible to system administrators and users.
- TR 13.2: A user support portal or ticketing system should be established to provide timely assistance, address user inquiries, and track and resolve issues.
- TR 13.3: The system should include self-service tools for users to access help resources, FAQs, and community forums to encourage self-support.
- TR 14.1: Cost monitoring and optimization tools should be implemented to track resource usage and identify opportunities for cost reduction, such as idle resources or over-provisioned services.
- TR 14.2: The system should support resource scaling based on usage patterns to optimize resource allocation and reduce operational costs during periods of lower demand.
- TR 14.3: Regular cost audits and budget reviews should be conducted to align expenses with the defined cost constraints and objectives.
- TR 15.1: Network latency should be minimized through content delivery networks (CDNs) and edge computing, ensuring that users receive content and responses quickly.
- TR 15.2: The system should implement data caching and load balancing to reduce response times for data retrieval and processing.
- TR 15.2: Latency testing should be conducted to measure and optimize system response times, with defined performance targets for various system functions.
- TR 16.1: The system must maintain an uptime of at least 99.9% to ensure that ad event data can be reliably accessed and processed whenever required.
- TR 17.1: The system must employ error detection and correction mechanisms to guarantee the accuracy of ad event data, preventing data discrepancies and ensuring correct results.

2.4 Tradeoffs

1. Tradeoff 1: Scalability vs. Cost Efficiency
   While the system must be highly scalable to accommodate increasing workloads (TR 6.1), achieving high scalability may come with increased infrastructure costs. Balancing scalability with cost efficiency requires careful resource provisioning (TR 14.2) and monitoring to optimize resource utilization and minimize operational expenses (TR 14.1).

2. Tradeoff 2: Low Latency vs. Cost
   Ensuring low latency for data processing and access (TR 15.1) may necessitate the use of high-performance, but potentially more expensive, infrastructure components. Achieving low latency must be weighed against cost constraints (TR 14.1), and organizations may need to invest in faster hardware or content delivery networks (CDNs) to meet latency targets (TR 15.2).

3. Tradeoff 3: Low Latency vs. Real-time anomaly detection
   Achieving low latency (TR 15.2) is critical for real-time applications, but real-time anomaly detection (TR 2.3) can introduce latency due to the additional processing required. Balancing low latency with timely anomaly detection is a tradeoff.

4. Tradeoff 4: High Availability vs. Complexity
   Ensuring high availability often involves redundant components and failover mechanisms (TR 8.1), which can increase system complexity. Organizations must weigh the benefits of high availability against the added complexity and maintenance efforts required to implement and manage redundancy.

5. Tradeoff 5: Real-Time Processing vs. Resource Intensiveness
   Achieving real-time processing of advertising data often requires significant computational resources, especially for complex real-time analytics (TR 2.1). Organizations must consider the resource costs and availability when opting for real-time processing (TR 13.3 & TR 8.1).

6. Tradeoff 6: Data Retention vs. Storage Costs
   Extending data retention periods (TR 4.1) may lead to increased storage costs, particularly if historical data is stored in high-performance storage solutions. Balancing data retention requirements with storage cost considerations is essential (TR 14.1).

3. Provider Selection

3.1 Criteria for choosing a provider

We defined a few criteria to help with the selection of cloud provider:

1. Data Ingestion and Transformation:
    a. Consider the provider's support for diverse data formats and protocols, as well as data quality checks and validation during ingestion (TR 1.1 and TR 1.3).
    b. Providers should offer data quality checks and validation (TR 1.2).
2. Real-Time Processing:
    a. Assess the provider's ability to handle low-latency real-time data processing, event-driven architectures, and anomaly detection/alerting mechanisms (TR 2.1, TR 2.2, and TR 2.3)
3. Storage and Analytics:
    a. Evaluate the provider's storage solutions for data durability and high availability (TR 3.1 and TR 3.3).
    b. Consider support for data compression techniques and data warehousing for analytics (TR 3.2).
    c. Data retention policies and data versioning (TR 4.1, TR 4.2, and TR 4.3).
4. Multi-Tenant Support:
    a. Assess the provider's support for creating isolated tenant accounts with role-based access control, user authentication, and tenant identification mechanisms (TR 9.1).
    b. The Provider supports multi-factor authentication (MFA) and integrates with standard identity providers (TR 10.1 and TR 10.2).
    c. Tenant isolation support (TR 5.1 and TR 5.2).
5. Scalability:
    a. Evaluate the provider's horizontal scalability, scalability testing, and support for containerization and orchestration technologies (TR 6.1, TR 6.2, TR 6.3).
6. High Availability and Disaster Recovery:
    a. Examine the provider's options for ensuring high availability through redundancy, geographic distribution, and disaster recovery capabilities (TR 8.1, TR 8.2, TR 12.1 and TR 12.2).
7. Security and Compliance:
    a. Assess the provider's security features, including data encryption at rest and in transit, and compliance with regulatory laws such as GDPR (TR 9.2 and TR 9.3).

8. Monitoring and Logging:
    a. Examine the provider's monitoring and alerting capabilities, as well as integration with centralized logging and log analysis platforms (TR 11.1, TR 11.2 and TR 11.3).
9. Cost Optimization:
    a. Assess the provider's cost monitoring and optimization tools, as well as support for resource scaling based on usage patterns and budget constraints (TR 14.1, TR 14.2, and TR 14.3).
10. Performance and Latency:
    a. Evaluate the provider's ability to minimize network latency through content delivery networks, data caching, and load balancing (TR 15.1).
    b. The Provider supports data caching and load balancing (TR 15.2).
11. Uptime and Data Accuracy:
    a. Assess the provider's uptime guarantees and error detection and correction mechanisms to ensure data accuracy and service reliability (TR 16.1 and TR 17.1).

3.2 Provider Comparison [2]

The table below offers a comparative assessment of the three major cloud providers based on the mentioned criteria. Each provider is assigned a score ranging from 1 (lowest) to 3 (highest).

Provider	AWS		GCP		Azure	
Parameter	Justification	Score	Justification	Score	Justification	Score
Data Ingestion and Transformation	AWS excels in data ingestion and transformation with services like Amazon Kinesis and AWS Glue, offering diverse data handling and ETL capabilities, supported by its extensive ecosystem and experience.	3	Azure, with Azure Data Factory and Stream Analytics, is strong for data ingestion and transformation. Integrated data quality features benefit Microsoft ecosystem users.	3	GCP excels in data ingestion and transformation with Dataflow and Dataprep. Ideal for diverse data and quality checks, advantageous for existing Google Cloud users.	2
Real-Time Processing	AWS stands out for real-time processing, featuring low-latency capabilities with AWS Lambda and Amazon Kinesis. Its ecosystem supports event-driven architectures and anomaly detection, ensuring reliability.	3	Azure excels in real-time processing with low-latency features such as Azure Functions and Stream Analytics. It's a strong choice for Microsoft ecosystem-integrated organizations with competitive event-driven and anomaly detection capabilities.	3	GCP offers strong real-time processing capabilities with Google Cloud Functions and Dataflow. It supports event-driven architectures and anomaly detection, making it ideal for Google Cloud-focused organizations.	3

[2] This section is written with the assistance of LLM

Storage and Analytics	AWS leads in this category with Amazon S3's durable storage, strong data integrity, and availability. Its ecosystem includes Amazon Redshift for speedy analytics, and features for data retention policies and versioning for effective data management.	3	Azure is a strong choice for storage and analytics, with Azure Blob Storage and Azure Synapse Analytics. It supports data governance with retention policies and versioning for effective data management..	2	GCP provides reliable storage with Google Cloud Storage, supports analytics through BigQuery, and offers data governance features like retention policies and versioning for effective data management.	3
Multi-Tenant Support	AWS excels in supporting multi-tenant environments, offering isolated tenant accounts, role-based access control, user authentication, tenant identification mechanisms, multi-factor authentication (MFA), and integration with identity providers, ensuring strong security and tenant isolation.	3	Azure is a strong option for multi-tenant support with isolated tenant accounts, role-based access control, user authentication, secure tenant identification, multi-factor authentication, and integration with identity providers, enhancing identity and access management in multi-tenant environments.	2	GCP provides multi-tenant support with isolated tenant accounts and user authentication. While it offers secure tenant identification, it has fewer identity management options compared to AWS and Azure. GCP maintains security through multi-factor authentication and integration with standard identity providers in multi-tenant environments.	3
Scalability	AWS is known for robust horizontal scalability, enabling the dynamic addition of	2	Azure offers horizontal scalability, supports scalability testing,	3	GCP offers horizontal scalability, supports scalability testing,	3

	resources. It provides scalability testing tools to identify bottlenecks, supports containerization, and orchestration with Amazon ECS and EKS, making it a reliable choice for scalable projects.		and provides containerization and orchestration with services like Azure Kubernetes Service (AKS) and Azure Container Instances, making it a flexible choice for projects with scalability needs.		and provides containerization and orchestration through Google Kubernetes Engine (GKE), making it a strong choice for projects with dynamic scalability requirements.	
High Availability and Disaster Recovery	AWS excels in high availability with redundancy, disaster recovery drills, and geographically distributed data centers or cloud regions. Services like Amazon Route 53 and AWS Global Accelerator ensure uninterrupted service availability, making it a reliable choice for disaster recovery and high availability.	3	Azure offers high availability through redundancy and supports disaster recovery with a global presence and Azure Site Recovery, ensuring service availability during disasters, making it a competitive choice.	2	GCP provides redundancy and disaster recovery capabilities with geographically distributed data centers and Google Cloud's Site Reliability Engineering (SRE) practices, ensuring high availability and disaster recovery, suitable for projects with such requirements.	3
Security and Compliance	AWS excels in security with data encryption, regulatory compliance focus (e.g., GDPR), making it a top choice for projects needing strong data protection and regulatory adherence.	3	Azure provides a comprehensive security package, including encryption and GDPR compliance, making it competitive for projects with stringent security and compliance needs.	3	GCP offers strong security with encryption but has a less established compliance track record compared to AWS and Azure. It's suitable for security and compliance needs, though AWS and Azure may have more mature solutions.	3

Monitoring and Logging	AWS excels in monitoring and alerting with AWS CloudWatch, automated alerting for anomaly detection, and seamless integration with centralized logging platforms, making it a strong choice for projects needing robust monitoring and logging capabilities.	2	Azure offers monitoring and alerting via Azure Monitor, automated alerting for anomaly detection, and integrates effectively with centralized logging platforms, making it a competitive choice for projects with monitoring and logging requirements.	3	GCP offers monitoring and logging via Google Cloud Monitoring and Logging, but AWS and Azure may have more extensive integrations with centralized logging and log analysis platforms, making them suitable for projects with diverse integration needs.	3
Cost Optimization	AWS provides robust cost monitoring and optimization tools, supporting usage-based resource scaling and aligning expenses with budget constraints, making it a strong choice for cost-conscious projects.	3	Azure offers cost monitoring and optimization tools with usage-based resource scaling and supports cost-efficiency. It also encourages regular cost audits and reviews to align expenses with budget objectives.	3	GCP provides cost monitoring and optimization tools, resource scaling, and features for cost audits and budget reviews. While it's suitable for cost optimization, AWS and Azure may have more mature cost management tools.	3
Performance and Latency	AWS minimizes network latency with CDNs, data caching, and load balancing services. Amazon CloudFront's global infrastructure ensures fast content delivery, while Amazon Elastic	2	Azure is strong in minimizing network latency with data caching, load balancing, and Azure CDN for fast content delivery. Azure Load Balancer optimizes traffic routing, making it	3	GCP provides solutions to minimize network latency with data caching, load balancing, and Google Cloud CDN for fast content delivery. Google Cloud Load Balancing	3

	Load Balancing optimizes traffic distribution, making it ideal for low-latency projects.		a solid choice for projects focusing on low-latency performance.		optimizes traffic distribution, making it suitable for low-latency performance-focu sed projects.	
Uptime and Data Accuracy	AWS provides a 99.9% service level agreement (SLA) for reliable service access, along with error detection and correction mechanisms for data accuracy, making it a trustworthy option for projects with high uptime and data accuracy requirements.	3	Azure offers a 99.9% SLA for service reliability and includes error detection and correction mechanisms to maintain data accuracy, meeting high uptime and data accuracy requirements.	3	GCP offers a 99.95% SLA for uptime, error detection, and correction mechanisms, making it a suitable choice for projects demanding high uptime and data accuracy.	2

3.3 The final selection

While the major cloud providers offer similar services, we choose AWS for its comprehensive services, security, and scalability. AWS excels in data processing, storage, and high availability, offering reliability for our project's success. Its extensive ecosystem and strong track record make it the ideal cloud platform.

3.3.1 The List Of Services Offered By The Winner [1]

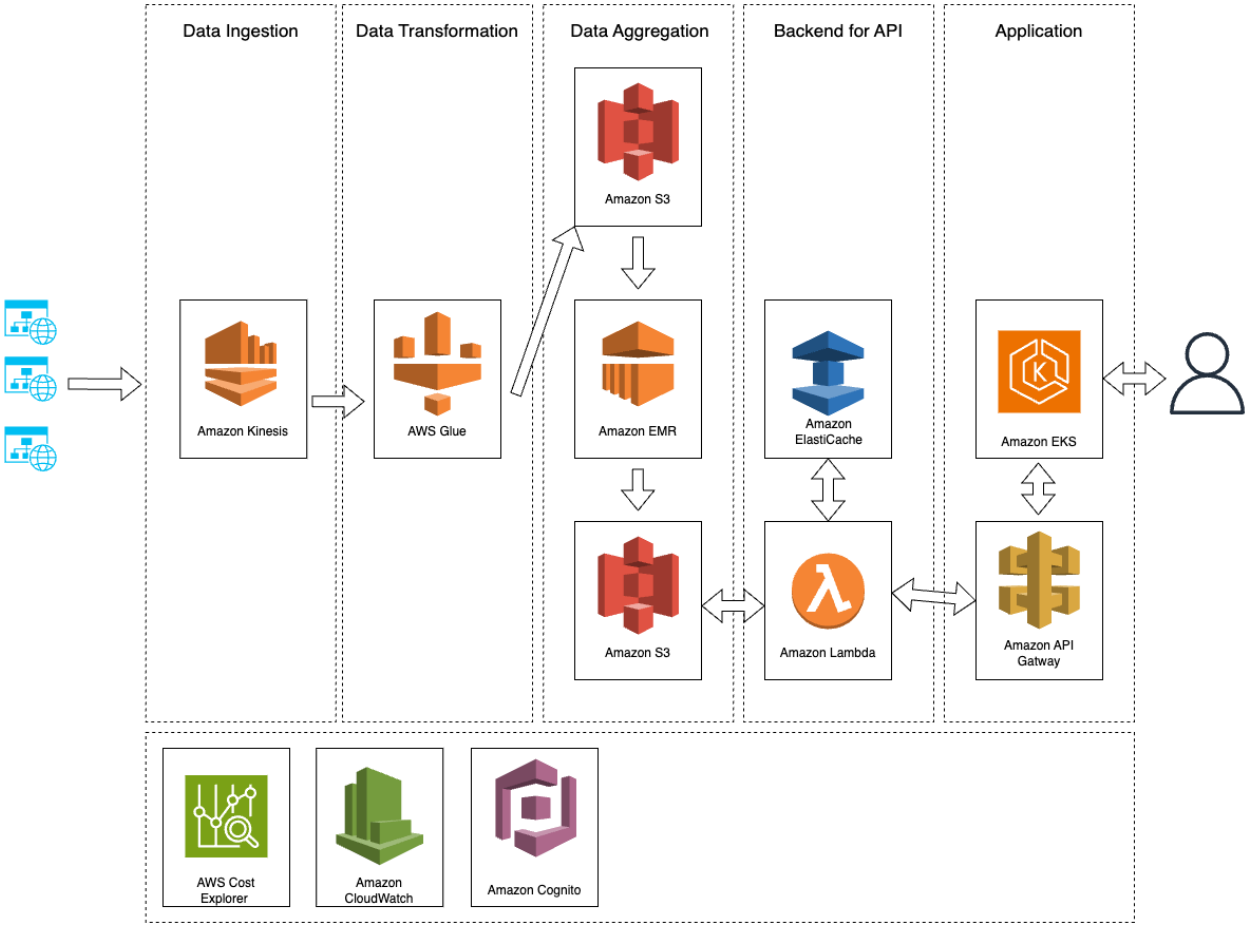
1. API Gateway
    a. Amazon API Gateway simplifies the process of creating and managing APIs. You can define API structure, endpoints, and stages for different environments.
    b. API Gateway provides robust request and response processing capabilities, including request validation, data transformation, and integration with backend services.
    c. API Gateway offers comprehensive monitoring and analytics, including logging, CloudWatch integration, and detailed API usage reports to track performance and usage.
2. Amazon Kinesis
    a. Amazon Kinesis is a managed service that enables real-time data streaming and processing. It is designed to handle large volumes of streaming data, making it ideal for use cases like IoT device telemetry, log processing, and event-driven applications.
    b. Kinesis offers multiple components, including Kinesis Data Streams, Kinesis Data Firehose, and Kinesis Data Analytics, for data ingestion, transformation, and analysis. Data streams are used to ingest data, and data analytics provides the capability to perform real-time processing and analysis.
    c. Kinesis is highly scalable and elastic, allowing you to scale up or down based on the volume of incoming data. It can handle real-time data from thousands of sources and automatically provisions and manages the required resources.
3. Amazon S3
    a. Amazon S3 is an object storage service that allows you to store and retrieve data in the form of objects. It is designed for durability, availability, and scalability of data.
    b. S3 is known for its high durability, with data automatically distributed across multiple availability zones to ensure resilience against hardware failures and outages. It is designed to provide 99.999999999% (11 nines) durability for objects.
    c. S3 also provides Glacier Storage for archiving.
4. Amazon DynamoDB
5. Amazon Aurora

6. AWS Lambda
    a. AWS Lambda is a serverless compute service that allows you to run code in response to events without the need to manage servers. It abstracts infrastructure management, enabling developers to focus solely on code development.
    b. Lambda functions are triggered by various events such as HTTP requests via API Gateway, changes in data on Amazon S3, updates to Amazon DynamoDB tables, and more. This event-driven architecture allows you to build responsive, scalable applications.
    c. Lambda automatically scales the execution of functions based on incoming event workload. You're billed only for the compute time consumed by your functions, making it cost-efficient and suitable for a wide range of workloads, from small to large-scale applications.
7. AWS Glue
8. Amazon EMR
9. Amazon RedShift
10. Amazon RDS
11. Amazon ElastiCache
12. Amazon CloudFront
13. Amazon CloudWatch
14. Amazon SNS
15. Amazon Cognito
16. Amazon IAM
17. AWS KMS
18. Elastic Load Balancing
19. AWS Cost Explorer

4. The First Design Draft

We select the following TRs from the list of TRs supplied in section 2.3.

- TR 1.1: The system should facilitate data ingestion from diverse sources by accommodating multiple data formats (e.g., CSV, JSON, and XML) and protocols, including REST APIs and batch data uploads.
- TR 2.1: The real-time processing component must have low-latency capabilities, with the ability to process incoming ad data within milliseconds of receipt.
- TR 2.2: It should support event-driven architectures, enabling immediate reactions to injection of data.
- TR 3.1: The system should employ a distributed and redundant storage infrastructure that ensures data durability and high availability.
- TR 4.1: The system should have well defined data retention policies for different types of data, taking into account regulatory requirements and business needs.
- TR 5.2: The system must implement a secure and unique tenant identification mechanism, such as a unique identifier or token, to distinguish and associate users or entities with their respective tenant profiles and data.
- TR 6.1: The system architecture should be designed for horizontal scalability, enabling the addition of new processing nodes, servers, or cloud resources dynamically.
- TR 8.1: The system architecture should include redundant components such as data replicas and load balancers to ensure uninterrupted service availability in case of failures.
- TR 10.2: User authentication should integrate with industry-standard identity providers (e.g., OAuth, LDAP) to streamline user management and authentication processes.
- TR 11.1: The system should implement comprehensive monitoring for indicators such as CPU utilization, data storage size, and other resources metrics.
- TR 14.1: Cost monitoring and optimization tools should be implemented to track resource usage and identify opportunities for cost reduction, such as idle resources or over-provisioned services.
- TR 16.1: The system must maintain an uptime of at least 99.9% to ensure that ad event data can be reliably accessed and processed whenever required.


Data Ingestion
Data Transformation
Data Aggregation
Backend for API
Application
Amazon S3
Amazon Kinesis
AWS Glue
Amazon EMR
Amazon ElastiCache
Amazon EKS
Amazon S3
Amazon Lambda
Amazon API Gatway
AWS Cost Explorer
Amazon CloudWatch
Amazon Cognito

4.1 The Basic Building Blocks Of The Design

1. Compute:
    a. AWS Lambda
        i. TR 2.1: Low-latency capabilities for real-time processing.
        ii. TR 2.2: Supports event-driven architectures for immediate reactions to data injection.
        iii. TR 6.1: Designed for horizontal scalability with dynamic addition of processing nodes
        iv. TR 16.1: The system must maintain an uptime of at least 99.9% to ensure that ad event data can be reliably accessed and processed whenever required.

    b. Amazon EKS
        i. TR 6.1: Designed for horizontal scalability with the ability to dynamically add new processing nodes.

    c. Amazon EC2
        i. TR 2.1: Low-latency capabilities for real-time processing.
        ii. TR 6.1: Scale up or down by launching or terminating instances as needed.

2. Storage:
    a. Amazon S3
        i. TR 1.1: Accommodates multiple data formats and protocols for diverse data ingestion.
        ii. TR 3.1: Employs a distributed and redundant storage infrastructure for data durability and high availability.
        iii. TR 4.1: Well-defined data retention policies for different types of data.

    b. Amazon ElastiCache (Redis)

3. Networking:
    a. Amazon API Gateway
        i. TR 1.1: Accommodates multiple data formats and protocols for diverse data ingestion.
        ii. TR 2.2: Supports event-driven architectures for immediate reactions to data injection.

b. Amazon Elastic Load Balancing
    i. TR 8.1: Architecture includes load balancers for uninterrupted service availability.

c. Amazon VPC

4. Analytics:
    a. Amazon Kinesis
        i. TR 1.1: Accommodates multiple data formats and protocols for diverse data ingestion.
        ii. TR 2.2: Supports event-driven architectures for immediate reactions to data injection.
        iii. TR 11.1: Comprehensive monitoring for resource metrics.

    b. AWS Glue
        i. TR 1.1: Accommodates multiple data formats and protocols for diverse data ingestion.

    c. Amazon EMR
        i. TR 1.1: Accommodates multiple data formats and protocols for diverse data ingestion.

5. Monitoring:
    a. Amazon CloudWatch
        i. TR 11.1: Comprehensive monitoring for resource metrics.

    b. AWS Cost Explorer
        i. TR 14.1: Cost monitoring and optimization tools for tracking resource usage and identifying opportunities for cost reduction.

6. Security, Identity & Compliance:
    a. Amazon Cognito
        i. TR 5.2: Implements a secure and unique tenant identification mechanism

4.2 Top-level, informal validation of the design [3]

1. Compute:
    a. AWS Lambda
        i. Executes serverless functions for various purposes, including real-time data transformation, triggering Lambda functions for dynamic data retrieval, and more.
        ii. Lambda charges you based on the exact amount of compute time your code uses, down to the millisecond.

    b. Amazon EKS
        i. Hosts containerized applications, enabling scalability and orchestration of microservices in a Kubernetes environment.
        ii. EKS connects to a data plane, consisting of managed EC2 node groups, accessible by simple API calls. These groups have auto-scaling features which provide elasticity.

    c. Amazon EC2
        i. Used together with EKS to run Kubernetes worker nodes in the data plane.

2. Storage:
    a. Amazon S3
        i. Store the raw, transformed data in Amazon S3.
        ii. Each Lambda function can save the data in S3 buckets.
        iii. Amazon S3 provides 99.999999999% (11 nines) durability for objects and high availability. While Amazon EFS and Amazon EBS offer storage solutions, they may not match S3's object durability.
        iv. S3 scales automatically to accommodate data growth, whereas services like Amazon RDS or Amazon DynamoDB may require more manual scaling adjustments.

    b. Amazon ElastiCache (Redis)
        i. Caches frequently accessed data to accelerate data retrieval and reduce the load on backend services like S3 and EMR.

[3] This section is written with the assistance of LLM.

3. Networking:
    a. Amazon API Gateway
        i. Provides a front-end for user access, allowing clients to interact with your data via APIs and route requests to backend services.

    b. Amazon Elastic Load Balancing.
        i. Balances incoming traffic across your EKS instances to ensure high availability and reliability.

    c. Amazon VPC
        i. To set up public and private subnets, perform ingress routing and internal IP addressing. An ELB can run together with a VPC to distribute traffic across subnets in multiple availability zones.
4. Analytics:
    a. Amazon Kinesis
        i. Used for real-time data ingestion, collecting data from multiple websites and streaming it to subsequent processing stages.
        ii. It is a better choice than Amazon SQS or Amazon SNS, which are primarily designed for messaging and queuing, not real-time data processing.
        iii. Kinesis can automatically scale to handle very high data volumes, whereas services like Amazon Direct Connect or Amazon Storage Gateway are not intended for real-time data processing and scaling in the same way.

    b. AWS Glue
        i. Performs data transformation and cleansing on the raw data received from Kinesis streams, ensuring it's ready for storage and analysis.
        ii. AWS Glue offers serverless extract, transform, and load (ETL) capabilities, making it a more streamlined and scalable choice compared to manually configuring ETL jobs using Amazon EC2 instances or other services.
        iii. Glue's Data Catalog is a fully managed metadata repository, offering superior data cataloging and schema management capabilities compared to manual metadata management in Amazon S3 or other storage services.

c. Amazon EMR
    i. Handles data aggregation, processing large datasets, and producing aggregated data that can be served to users.
    ii. Amazon EMR (previously called Amazon Elastic MapReduce) is a managed cluster platform that simplifies running big data frameworks, such as Apache Hadoop and Apache Spark, on AWS to process and analyze vast amounts of data.
    iii. EMR offers a fully managed Hadoop ecosystem, which is more convenient than manually setting up and managing Hadoop clusters on EC2 instances.

d. Amazon CloudWatch
    i. Monitors the health and performance of your AWS resources, providing insights into system metrics and log data.

e. AWS Cost Explorer
    i. Helps you optimize costs by providing insights into resource usage and spending patterns within your AWS environment.

5. Security, Identity & Compliance:
    a. Amazon Cognito
        i. Provides user authentication and identity management for your application, enhancing security and user access control.

4.3 Action items and rough timeline

Skipped

5. The second design

5.1 Use of the Well-Architected Framework

The AWS Well-Architected Framework [2] suggests following steps in the context of our project:

Operational Excellence [3]:

1. Prepare:

- Define operational processes for data ingestion, real-time processing, and scaling.
- Develop playbooks for incident response and recovery procedures.

2. Operate:

- Implement automated monitoring using CloudWatch for resource metrics.
- Respond to events, such as data injection or system failures, following predefined procedures.

3. Evolve:

- Continuously review and update operational documentation based on learnings.
- Conduct regular reviews to identify areas for improvement in operational processes.

Security [4]:

1. Implement a Strong Identity Foundation:

- Utilize Amazon Cognito for secure and unique tenant identification.
- Enforce least privilege access across AWS services.

2. Enable Traceability:

- Implement logging and tracing mechanisms for data flows and security events.
- Monitor and audit security-related events using CloudWatch.

3. Apply Security at All Layers:

- Implement security measures at the network (Amazon VPC), application (AWS Lambda, EKS), and data layers (Amazon S3).

4. Automate Security Best Practices:

- Leverage AWS services such as Amazon API Gateway, Lambda, and EKS for automated security enforcement.
- Regularly test and automate security configurations.

Reliability [5]:

1. Test Recovery Procedures:

- Regularly test backup and recovery procedures for data stored in Amazon S3.\
- Simulate failure scenarios for event-driven architectures (Amazon Kinesis, Lambda).

2. Automatically Recover from Failures:

- Leverage auto-scaling features of EKS and EC2 for dynamic resource provisioning.
- Design for graceful degradation in case of component failures.

Performance Efficiency [6]:

1. Use Multi-Tier Architectures:

- Utilize AWS Lambda, EKS, and EC2 for a multi-tier architecture.
- Distribute workloads across scalable components.

2. Implement Elasticity:

- Implement auto-scaling for services like Lambda, EKS, and EC2.
- Use on-demand scaling to match resources to demand.

3. Optimize Over Time:

- Regularly review and optimize resource usage using AWS Cost Explorer.
- Implement monitoring for identifying and addressing performance bottlenecks.

Cost Optimization [7]:

1. Implement Cost-Effective Resources:

- Select the right types and sizes of resources based on performance requirements.
- Use managed services like AWS Glue for cost-effective serverless ETL.

2. Match Supply and Demand:

- Implement auto-scaling for dynamic resource provisioning.
- Utilize reserved instances for predictable workloads.

3. Optimize Over Time:

- Leverage AWS Cost Explorer for continuous monitoring and optimization of costs.
- Identify and address opportunities for cost reduction, such as idle resources or over-provisioned services.

These steps align with the pillars of the Well-Architected Framework and provide guidance for designing and maintaining a well-architected system on AWS.

5.2 Discussion of Pillars

The AWS Well-Architected Framework serves as a valuable resource for gaining insights into the advantages and considerations associated with decisions made during the construction of systems on AWS. Utilizing this framework provides a comprehensive understanding of architectural best practices, enabling the effective design and operation of systems that are reliable, secure, efficient, cost-effective, and sustainable within the cloud environment.

The AWS Well-Architected Framework is based on 6 pillars :

1. Operational Excellence
2. Security
3. Reliability
4. Performance Efficiency
5. Cost Optimization
6. Sustainability

We will now discuss 2 pillars in detail that aligns with our project requirements -

1. **Reliability Pillar**

As mentioned in the AWS WAF document, the reliability pillar addresses the capacity of a workload to consistently and accurately execute its intended function within expected parameters. This involves the capability to manage and assess the workload across its entire lifecycle. This document offers detailed and best-practice guidance for establishing dependable workloads on the AWS platform.

The reliability pillar talks about 5 major design principles -

i) Test recovery procedures: Within a cloud setting, you can evaluate potential failure points within your workload and validate recovery procedures. The use of automation enables the simulation of diverse failure scenarios or the recreation of situations that previously led to failures. Our application relies on Amazon S3 for storing both raw and transformed data. Consistently testing backup and restore procedures for S3 data ensures a robust recovery process in the event of failures.

ii) Automatically Recover from Failures: By observing a workload's key performance indicators (KPIs), automation can be initiated upon breaching a predefined threshold. These KPIs should reflect business value rather than technical aspects of service operation. This facilitates automatic notification and failure tracking, enabling automated recovery processes that either circumvent or rectify the encountered failure. The inclusion of services like AWS Lambda, Amazon EKS, and

Amazon EC2, with their respective auto-scaling capabilities, aligns well with the principle of automatic recovery from failures. These services can dynamically adjust resources based on demand and ensure continuous operation.

iii) Scale Horizontally to Increase Aggregate System Availability: Replacing one large resource with multiple small resources reduces the risk of a single point of failure. Services like AWS Lambda, EKS, and EC2 provide fully-managed or semi-managed autoscaling capabilities that align with the principle of scaling horizontally to increase overall system availability and fault tolerance.

iv) Stop Guessing Capacity: A common cause of failure is excessive demand as compared to the system's capacity. For example, at the time of peak traffic there are not enough computing resources to serve all of the customers. The use of auto-scaling and dynamic resource provisioning aligns with the principle of avoiding the guesswork of resource capacity. The project can dynamically adjust resources based on actual demand, optimizing both performance and cost.

v) Manage Change in Automation: This principle directs that the changes in the infrastructure should be made using automation to avoid any manual errors. The adoption of services like AWS Lambda and automation tools for various purposes, including real-time data transformation, aligns with the principle of managing change through automation. This ensures controlled and repeatable changes to the environment.

2. Security Pillar

The Security pillar involves securing data, systems, and assets, leveraging cloud technologies to enhance overall security measures.

The security pillar talks about 7 major design principles -

i) Implement a strong identity foundation: Adopt the principle of least privilege and ensure the separation of duties by assigning appropriate permissions for each interaction with AWS resources. Centralized identity management, striving to reduce dependence on long-term static credentials. The use of Amazon Cognito for user authentication aligns with the principle of implementing a strong identity foundation. This service provides secure and unique tenant identification, enhancing user access control and security.

ii) Maintain Traceability: Real-time monitoring, alerting, and auditing of actions and changes in your environment are essential. Integrating log and metric collection systems enables the automatic investigation and response. The inclusion of CloudWatch for monitoring and logging aligns with the principle of enabling traceability. Monitoring and auditing security-related events using CloudWatch ensures visibility into system metrics and security events.

iii) Apply Security at All Layers: This principle emphasizes the importance of implementing security measures across multiple layers of your architecture network, application, and data layers to create a comprehensive and robust security posture. The use of Amazon VPC for network security, AWS Lambda for application security, and Amazon EKS for containerized applications collectively enhances the overall security posture.

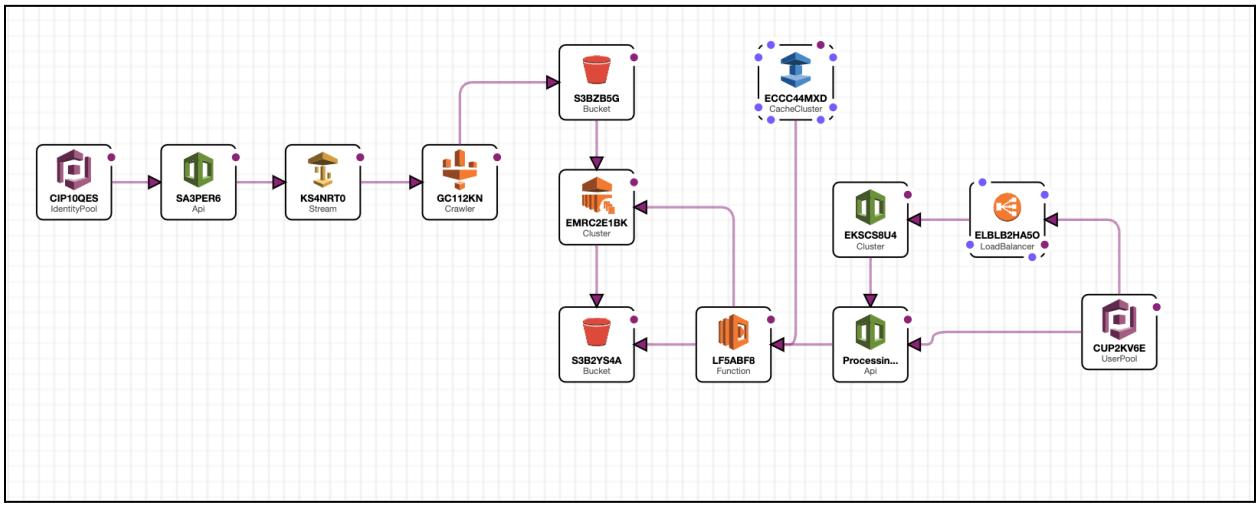
iv) Automate security best practices: This principle encourages organizations to use automation tools and processes to enforce and maintain security measures consistently across their cloud infrastructure. The goal is to reduce manual intervention, ensure adherence to security policies, and enhance overall security posture. The utilization of AWS Lambda and Amazon EKS for automated security enforcement aligns with the principle of automating security best practices.

v) Protect Data in Transit and at Rest: This principle encourages the architect to classify data into sensitivity levels and use mechanisms, such as encryption, tokenization, and access control where appropriate. The project's use of Amazon VPC for securing data in transit and Amazon S3 for storing raw and transformed data aligns with the principle of protecting data both in transit and at rest.

vi) Keep people away from data: Leveraging cloud-native architectures and managed services, such as Lambda and S3, aligns with the principle of minimizing physical presence in data centers. This approach reduces the need for manual interventions and enhances overall security.

vii) Prepare for security events: This principle focuses on proactively establishing measures to anticipate, detect, respond to, and recover from security incidents effectively. It emphasizes the importance of having a well-defined incident response plan and the necessary tools and processes in place to address potential security events. The project's use of CloudWatch for comprehensive monitoring and setting up alarms aligns with the principle of preparing for security events.

5.3 Use of CloudFormation diagrams



5.4 Validation of the design

We now justify why our design allows us to meet the selected TRs in the previous section.

i) Authentication:

We employ Amazon Cognito to authenticate entities, encompassing both the website responsible for data injection and the users accessing the application to retrieve aggregated counts of ad-events. This fulfills our technical requirements, specifically meeting the criteria for implementing a secure and unique tenant identification mechanism (TR 5.2). Additionally, Cognito facilitates user authentication through industry-standard identity providers, aligning with our technical requirement for integration with such providers (TR 10.2). The identity pool limit of around 50,000 identities offered by Cognito is well-suited for the project's needs (source: Cognito Limits).

ii) Real-time processing and event-driven architecture:

The application demands a real-time processing component capable of providing users with instantaneous responses (TR 2.1). Furthermore, it necessitates an asynchronous, event-driven architecture wherein processes are triggered solely upon the introduction of new data or when a user requests aggregation with specific filters TR (2.2). To fulfill these requirements, we have

implemented a Kinesis stream for real-time data ingestion and a Lambda function to deliver real-time aggregation to consumers. This means that the real-time processing component will have low-latency capabilities, with the ability to process incoming ad data within milliseconds of receipt. Despite the fact that Amazon EMR is not inherently a real-time component, we contend that, given our substantial dataset and reliance on map-reduce as a foundation, EMR is a suitable choice for our architecture.

iii) Storage:

We aim to store both raw and transformed/aggregated data in our system. While end users typically receive aggregated data, they have the option to apply filters and request the aggregation of new data. To achieve this, the system utilizes AWS Glue to transform disparate data formats into a unified format, which is then stored in Amazon S3 (meeting TR 1.2 and 1.3). S3 offers the capability for both same-region and cross-region replication, ensuring data durability and high availability, aligning with TR 3.1. Additionally, developers have the flexibility to establish a lifecycle policy, enabling the periodic archiving and deletion of old data, as specified in TR 4.1. S3 also provides SLA of 99.999% of durability and availability (meeting TR 16.1).

iv) Application:

We leverage AWS EMR to aggregate substantial volumes of event data. Given that EMR is fully managed by AWS, it automatically scales to accommodate increased incoming data for aggregation, addressing the requirements of TR 6.1. Furthermore, our web application, accessed by clients, is hosted on AWS EKS, providing seamless horizontal scalability as outlined in TR 6.1. The incorporation of a load balancer guarantees the even distribution of traffic, preventing strain on any single instance.

v) Monitoring and alerting

We employ CloudForm for comprehensive monitoring of diverse Key Performance Indicators (KPIs) and the systematic logging of transactions. This approach not only enables straightforward tracking of undesirable behaviors but also ensures a meticulous observation of the overall system performance and associated costs. By adhering to the stipulations outlined in TR 11.1 and 14.1, we enhance our ability to proactively identify and address issues, optimize resource utilization, and maintain cost-effectiveness across the system.

5.5 Design Principles And Best Practices Used

1. Operational Excellence:
    a. Design Principle: Make frequent, small, reversible changes [8]

       Best Practice: OPS11-BP03 Implement feedback loops [9]

       Explanation: Our project embraced automated monitoring, incident response reviews, and continuous updates to operational documentation. Auto-scaling and AWS Cost Explorer were employed with regular reviews for dynamic resource provisioning and cost optimization. Security measures included logging and tracing for ongoing monitoring and auditing, promoting continuous system enhancements.

    b. Design Principle: Use managed services [8]

       Best Practice: OPS04-BP01 Identify key performance indicators [10]

       Explanation: In our project, we strategically employed AWS managed services to alleviate operational burdens. Leveraging services like AWS Lambda, Amazon RDS, and Amazon S3, we reduced the need for manual maintenance and optimized resource utilization. Operational procedures were tailored around interactions with these managed services, enhancing efficiency and streamlining system management.

2. Security:
    a. Design Principle: Implement a strong identity foundation [11]

       Best Practice: SEC02-BP01 Use strong sign-in mechanisms [12]

       Explanation: We established fine-grained access controls and permissions across our AWS environment. Utilizing services like Amazon IAM, we defined and enforced precise access policies, ensuring that users and entities only had the necessary permissions for their roles. Regular access reviews and audits were conducted to identify and promptly address any unnecessary or outdated privileges, maintaining a secure and well-governed identity framework in alignment with the AWS Well-Architected Framework's recommendations.

b. Design Principle: Apply security measures at all layers, including network, application, and data [11]

   Best Practice: SEC05-BP02 Control traffic at all layers [13]

   Explanation: At the network layer, we leveraged Amazon VPC to establish secure and isolated environments. For the application layer, security measures were implemented using services like AWS Lambda and Amazon EKS, ensuring secure code execution and container orchestration. At the data layer, sensitive information stored in Amazon S3 was encrypted, and access controls were enforced to safeguard data integrity. Regular security assessments and audits were conducted across all layers to identify and address potential vulnerabilities, aligning with the AWS Well-Architected Framework's security principles.

3. Reliability:
   a. Design Principle: Automatically Recover from Failure [14]

      Best Practice: REL01-BP05 Automate quota management [15]

      Explanation: Automation is key to automatically recovering from failure. By automating the management of service quotas, especially those related to resource scaling, we ensure that your system can dynamically respond to changing demands and recover from potential resource saturation.

   b. Design Principle: Test Recovery Procedures [14]

      Best Practice: REL06-BP04 Automate responses [16]

      Explanation: Automating responses, including recovery procedures, is crucial for testing how your workload fails and validating recovery strategies. By automating responses, we can simulate different failure scenarios, assess the effectiveness of your recovery procedures, and proactively address potential issues before they impact the system.

4. Performance Efficiency:
   a. Design Principle: Democratize Advanced Technologies [17]

      Best Practice: PERF01-BP01 Learn about and understand available cloud services and features [18]

Explanation: Democratizing advanced technologies involves delegating complex tasks to the cloud vendor. Learning about and understanding available cloud services and features is crucial for making informed decisions on consuming technologies as services. This best practice supports the principle of making advanced technology implementation smoother for the team.

b. Design Principle: Go Global in Minutes [17]

Best Practice: PERF04-BP06 Choose your workload's location based on network requirements [19]

Explanation: Going global in minutes requires selecting the appropriate locations based on network requirements. This best practice emphasizes the importance of choosing the workload's location strategically to provide lower latency and a better experience for users worldwide, aligning well with the principle of deploying workloads in multiple AWS Regions.

5. Cost Optimization:
   a. Design Principle: Adopt a Consumption Model [20]

   Best Practice: COST05-BP05 Select components of this workload to optimize cost in line with organization priorities [21]

   Explanation: Adopting a consumption model involves selecting components of the workload to optimize costs in line with organization priorities. This best practice ensures that cost optimization efforts are strategically aligned with organizational goals and priorities, allowing for efficient resource usage.

   b. Design Principle: Stop Spending Money on Undifferentiated Heavy Lifting [20]

   Best Practice: COST02-BP03 Implement an account structure [22]

   Explanation: The principle of stopping spending money on undifferentiated heavy lifting aligns with the best practice of implementing an account structure. By structuring accounts effectively, organizations can delegate responsibilities and shift focus to business projects, reducing the operational burden of managing IT infrastructure and promoting cost efficiency.

5.6 Tradeoffs Revisited [23]

Tradeoff 1: Scalability vs. Cost Efficiency

Our project is navigating the challenge of scalability versus cost efficiency, a decision that resonates with the even-swap approach. We recognize the need for a highly scalable system to handle increasing workloads. However, we're also aware that achieving high scalability might bump up our infrastructure costs. To strike the right balance, we're diligently provisioning resources and keeping a close eye on their usage. This aligns perfectly with the even-swap method, where careful evaluation and iterative consideration of alternatives are key.

Tradeoff 2: Low Latency vs. Cost

In grappling with the decision between low latency and cost, our project is applying a thoughtful approach inspired by the even-swap methodology. We understand that ensuring low latency, while crucial for efficient data processing, might lead us to invest in more expensive infrastructure components. To make a well-informed decision, we're focusing on the specifics of the tradeoff and maintaining logical consistency. This mirrors the principles of the even-swap method, emphasizing a measured and strategic decision-making process.

Tradeoff 3: Low Latency vs. Real-time Anomaly Detection

Our project is facing the challenge of balancing low latency and real-time anomaly detection, and we're tackling it with a clear understanding of our needs. This decision isn't arbitrary; instead, it's grounded in a rational evaluation influenced by the even-swap method. We're seeking dominance and making consistent swaps, reflecting the iterative nature of our decision-making process. Each step is carefully considered, ensuring a balanced approach to low latency and timely anomaly detection.

Tradeoff 4: High Availability vs. Complexity

The tradeoff between high availability and complexity in our project aligns with the even-swap method's recommendation to create dominance intelligently. We're evaluating the benefits of high availability against the added complexity, taking a strategic and measured approach. Our systematic evaluation of alternatives reflects the principles advocated in the even-swap method, ensuring that we consider each component's contribution to high availability against the complexities introduced.

Tradeoff 5: Real-Time Processing vs. Resource Intensiveness

Our project's decision to balance real-time processing with resource intensiveness showcases a pragmatic approach influenced by the even-swap method. We're concentrating on the specifics of the tradeoff and making consistent decisions to pursue real-time processing without compromising resource efficiency. This aligns with the even-swap method's guidance on

determining the relative value of consequences, especially when it comes to computational resource costs.

Tradeoff 6: Data Retention vs. Storage Costs

When considering data retention versus storage costs, our project is taking a systematic approach, echoing the principles of the even-swap method. We're making even swaps and carefully assessing the relative importance of each consequence. This tradeoff exemplifies our commitment to a thoughtful and measured decision-making process, drawing inspiration from the even-swap methodology.

5.7 Discussion Of An Alternate Design

Skipped.

6. Kubernetes experimentation

6.1 Experiment Design[4]

We employ ad event aggregation to consolidate substantial amounts of data, aiding businesses in informed decision-making. A crucial feature of this application is its requirement for horizontal scalability (as specified in TR 6.1). In our experimentation, we focus on achieving horizontal scalability using Kubernetes, specifically exploring the capabilities of the autoscaler to dynamically adjust resources based on fluctuating request loads.

In the context of a web application, end-users access aggregated data and insights through a user-friendly interface. This application communicates with an API layer built using Lambda, fetching required data from an S3 bucket or triggering new aggregations on custom-filtered data. For the sake of simplicity in this experiment, we chose a straightforward application that calculates the digits of pi and the sum of squares. These operations, being computationally intensive, serve as a representative model for the ad event aggregation application.

A typical flow of the application would be -

1. Users interact with the application through a centralized point, the load balancer.
2. The load balancer directs requests to one of the pods within the Kubernetes cluster.
3. The container on the selected pod processes the request and returns the response to the client.

Nevertheless, in instances where the volume of requests experiences a sudden surge, relying on a single container may prove insufficient to meet the demand. It becomes imperative to horizontally scale up the number of pods, enabling the application to handle a higher concurrent request load. Subsequently, as the request volume returns to normal levels, it is essential to scale down the number of pods to their typical count. This dynamic scaling mechanism ensures optimal resource utilization in response to varying levels of demand.

Experiment Setup:

1. Kubernetes: In our experimental configuration, we deploy containers within a Kubernetes cluster. To simulate and assess this setup, we leverage Minikube [24] on our local machine. Minikube effectively emulates the Kubernetes environment locally by utilizing a Docker [25] container.

2. Autoscaling: Employing the HPA feature in Kubernetes, we automate the scaling process, dynamically adjusting the number of pods based on CPU utilization. Our chosen metric

[4] The code for this experiment is available here - Github

3. for autoscaling and downsizing is CPU utilization. Throughout the experiment, we set the minimum pod count to 1 and cap the maximum at 10.

4. Input: The application is hosted locally on the localhost, and requests can be directed to a specific endpoint through a designated port.

5. Output: Upon successful communication with the application, the expected outcome is an HTTP status code of 200, indicating a successful response.

6.2 Workload generation with Locust [26]

To systematically introduce and evaluate the load on our system, we employ the Locust load generator with the following configuration:

- Request: GET /
- Target Host: http://127.0.0.1:57979
- Number of Users (Peak Concurrency): 5000
- Spawn Rate (Users Started/Second): 10
- Total Number of Requests: 151587
- Requests per Second: 430

Steps to Run the Locust Test:

i. Create a Locust Client Template:

Develop a Locust client template that dispatches requests with a delay ranging from 5 to 15 seconds.

ii. Run the locustfile.py:

Execute the Locustfile.py, initiating the Locust server.

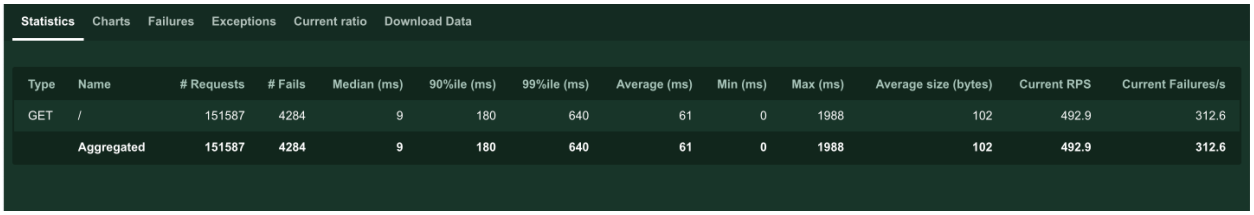
iii. Configure Test Parameters:

Specify the following test parameters:
- Peak Concurrency: 5000
- Spawn Rate: 10 users started per second
- Target Host URL: http://127.0.0.1:57979

These steps collectively establish a comprehensive load generation scenario using Locust, enabling the assessment of our Kubernetes cluster's performance under dynamic conditions.

6.3 Analysis Of The Results

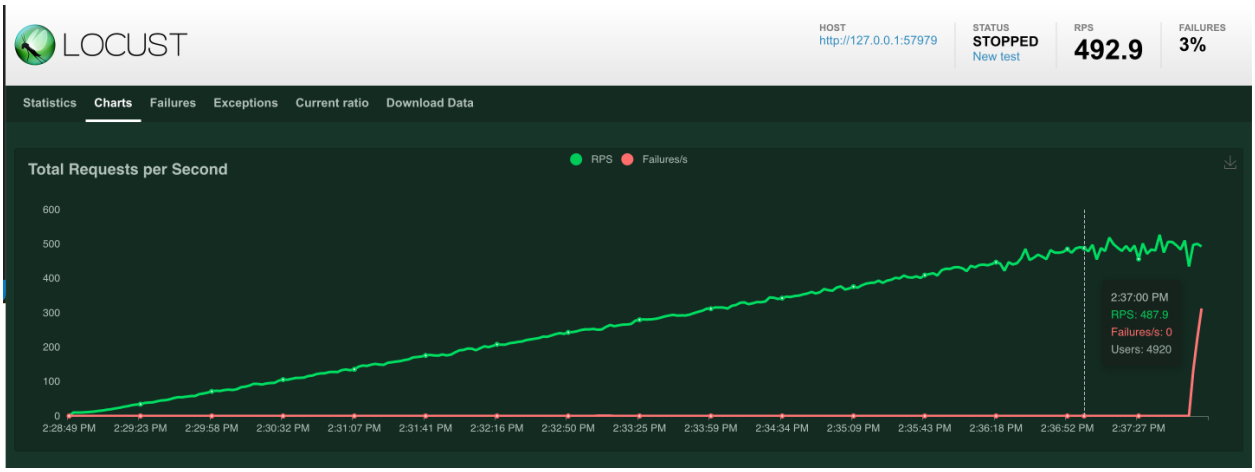
The locust load generation -



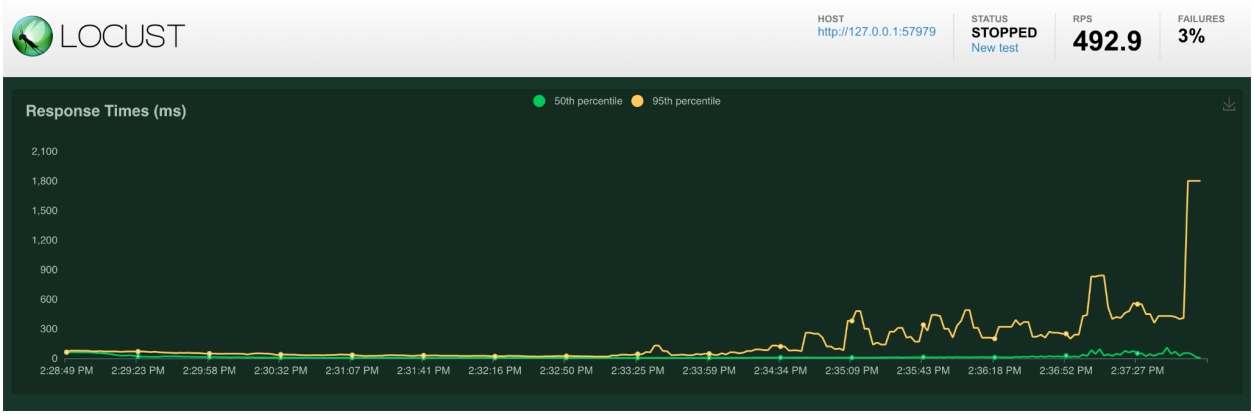
Statistics | Charts | Failures | Exceptions | Current ratio | Download Data

Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/	151587	4284	9	180	640	61	0	1988	102	492.9	312.6
	Aggregated	**151587**	**4284**	**9**	**180**	**640**	**61**	**0**	**1988**	**102**	**492.9**	**312.6**

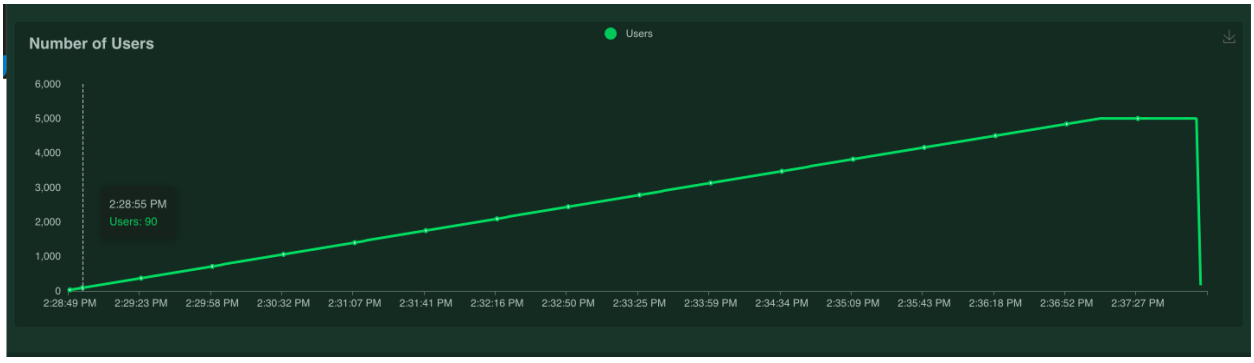
Observing the experiment, it's evident that Locust generated a total of 151587 requests. However, it's noteworthy that towards the conclusion of the experiment, some requests failed. This failure can be attributed to the issue previously discussed in the threat to validity, specifically, the local setup with Flask and Kubernetes on the localhost.



The screenshot above illustrates the cumulative number of requests per second throughout the entire experiment. It demonstrates a gradual and incremental increase in the number of requests over the course of the experiment.



The provided screenshot displays the response time in milliseconds. Notably, due to pod replication facilitated by the autoscaler, the average response time is consistently maintained between 0 to 300 milliseconds. However, it becomes apparent that beyond a certain point, the response time starts to increase.



The command line output demonstrating autoscaling -

```
(sukhadjoshi@Sukhads-MacBook-Air csc547 % kubectl get deployments
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
ad-events-agg   1/1     1            1           24s
(sukhadjoshi@Sukhads-MacBook-Air csc547 % kubectl get services
NAME           TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
ad-events-agg   ClusterIP   10.109.42.46   <none>        5000/TCP   29s
kubernetes     ClusterIP   10.96.0.1      <none>        443/TCP    18d
(sukhadjoshi@Sukhads-MacBook-Air csc547 % kubectl autoscale deployment ad-events-agg --cpu-percent=50 --min=1 --max=10
horizontalpodautoscaler.autoscaling/ad-events-agg autoscaled
(sukhadjoshi@Sukhads-MacBook-Air csc547 % kubect get hpa
zsh: command not found: kubect
(sukhadjoshi@Sukhads-MacBook-Air csc547 % kubectl get hpa
NAME           REFERENCE                  TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
ad-events-agg   Deployment/ad-events-agg   <unknown>/50%   1         10        0          13s
(sukhadjoshi@Sukhads-MacBook-Air csc547 % kubectl get hpa
NAME           REFERENCE                  TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
ad-events-agg   Deployment/ad-events-agg   <unknown>/50%   1         10        1          17s
(sukhadjoshi@Sukhads-MacBook-Air csc547 % kubectl get hpa
NAME           REFERENCE                  TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
ad-events-agg   Deployment/ad-events-agg   <unknown>/50%   1         10        1          76s
(sukhadjoshi@Sukhads-MacBook-Air csc547 % kubectl get hpa
NAME           REFERENCE                  TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
ad-events-agg   Deployment/ad-events-agg   0%/50%    1         10        1          102s
(sukhadjoshi@Sukhads-MacBook-Air csc547 % kubectl get hpa --watch
NAME           REFERENCE                  TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
ad-events-agg   Deployment/ad-events-agg   0%/50%    1         10        1          3m1s
ad-events-agg   Deployment/ad-events-agg   5%/50%    1         10        1          3m31s
ad-events-agg   Deployment/ad-events-agg   61%/50%   1         10        1          4m16s
ad-events-agg   Deployment/ad-events-agg   61%/50%   1         10        2          4m31s
ad-events-agg   Deployment/ad-events-agg   44%/50%   1         10        2          5m16s
ad-events-agg   Deployment/ad-events-agg   53%/50%   1         10        2          6m16s
ad-events-agg   Deployment/ad-events-agg   58%/50%   1         10        2          7m16s
ad-events-agg   Deployment/ad-events-agg   58%/50%   1         10        3          7m31s
ad-events-agg   Deployment/ad-events-agg   47%/50%   1         10        3          8m16s
ad-events-agg   Deployment/ad-events-agg   55%/50%   1         10        3          9m16s
ad-events-agg   Deployment/ad-events-agg   55%/50%   1         10        4          9m31s
ad-events-agg   Deployment/ad-events-agg   49%/50%   1         10        4          10m
ad-events-agg   Deployment/ad-events-agg   56%/50%   1         10        4          11m
ad-events-agg   Deployment/ad-events-agg   56%/50%   1         10        5          11m
ad-events-agg   Deployment/ad-events-agg   42%/50%   1         10        5          12m
ad-events-agg   Deployment/ad-events-agg   0%/50%    1         10        5          13m
ad-events-agg   Deployment/ad-events-agg   0%/50%    1         10        5          17m
ad-events-agg   Deployment/ad-events-agg   0%/50%    1         10        4          17m
ad-events-agg   Deployment/ad-events-agg   0%/50%    1         10        4          18m
ad-events-agg   Deployment/ad-events-agg   0%/50%    1         10        1          18m
```

Locust systematically ramps up client numbers until hitting 5000. The autoscaler, keeping an eye on pod CPU usage, dynamically spawns new replicas when the average cpu usage exceeds 50%. In our experiment, the maximum number of replicas created are 5. Once Locust concludes its load generation, the autoscaler smartly scales down pod numbers to a minimum of 1. This adaptive behavior ensures an efficient use of resources, aligning with the fluctuating demands on the Kubernetes cluster. This experiment validates the use of kubernetes for horizontal scaling.

Threat to validity:

Despite the autoscaler effectively adjusting the number of pods in response to usage, a notable threat to validity emerges. Specifically, there is a sudden surge in the request failure rate once the client count surpasses 5000. We believe that this issue stems from running Kubernetes and the

Flask application locally. Flask, when operating on the localhost, terminates connections upon receiving an excessive number of requests. It's crucial to recognize that this challenge extends beyond the purview of the cloud architecture, emphasizing a limitation inherent to the local environment setup.

7. Ansible playbooks

Skipped.

8. Demonstration

Skipped.

9. Comparisons

Skipped.

10. Conclusion

10.1 The Lessons Learned

Throughout the course of this project, valuable lessons have been gleaned that contribute to our understanding of effective decision-making and project management.

- Identifying the difference between business requirements and technical requirements and how business requirements are different for cloud architect and software architect.
- Identifying conflicting requirements emerged as a challenging yet immensely rewarding undertaking throughout the project.
- Selecting the optimal provider for our project proved to be a significant and intriguing task.
- Recognizing the significance of the design draft and the cloud formation diagram was also a crucial and enlightening aspect of our project.
- AWS WAF proved to be an unexpectedly valuable resource in formulating requirements. It not only supplied us with a wealth of pre-established, field-tested technical requirements applicable to real-world projects but also offered accompanying examples, benefits, best practices, and comprehensive documentation. The WAF stands out as a tool we intend to leverage in future endeavors, potentially adopting various roles outlined in the course.

10.2 Possible continuation of the project

As we conclude this phase of the project, there is considerable potential for its continuation and further refinement.

Continuing the project could involve refining the tradeoff strategies based on evolving technological landscapes and organizational priorities. Further exploration into emerging technologies, resource provisioning methodologies, and advancements in anomaly detection could enhance the project's overall effectiveness.

11. References

[1] Cloud, Amazon. “AWS Cloud Products.” *AWS*, https://aws.amazon.com/products/?aws-products-all.sort-by=item.additionalFields.productNameLowercase&aws-products-all.sort-order=asc&awsf.re%3AInvent=*all&awsf.Free%20Tier%20Type=*all&awsf.tech-category=*all

[2] AWS Well-Architected Framework
https://docs.aws.amazon.com/wellarchitected/latest/framework/welcome.html

[3] Operational Excellence
https://docs.aws.amazon.com/wellarchitected/latest/framework/operational-excellence.html

[4] Security
https://docs.aws.amazon.com/wellarchitected/latest/framework/security.html

[5] Reliability
https://docs.aws.amazon.com/wellarchitected/latest/framework/reliability.html

[6] Performance Efficiency
https://docs.aws.amazon.com/wellarchitected/latest/framework/performance-efficiency.html

[7] Cost Optimization
https://docs.aws.amazon.com/wellarchitected/latest/framework/cost-optimization.html

[8] Operational Excellence - Design Principles
https://docs.aws.amazon.com/wellarchitected/latest/framework/oe-design-principles.html

[9] Operational Excellence - Best practices - OPS11-BP03
https://docs.aws.amazon.com/wellarchitected/2023-10-03/framework/ops_evolve_ops_feedback_loops.html

[10] Operational Excellence - Best practices - OPS04-BP01
https://docs.aws.amazon.com/wellarchitected/2023-10-03/framework/ops_observability_identify_kpis.html

[11] Security - Design Principles
https://docs.aws.amazon.com/wellarchitected/2023-10-03/framework/sec-design.html

[12] Security - Best practices - SEC02-BP01

https://docs.aws.amazon.com/wellarchitected/2023-10-03/framework/sec_identities_enforce_mechanisms.html

[13] Security - Best practices - SEC05-BP02
https://docs.aws.amazon.com/wellarchitected/2023-10-03/framework/sec_network_protection_layered.html

[14] Reliability - Design Principles
https://docs.aws.amazon.com/wellarchitected/2023-10-03/framework/rel-dp.html

[15] Reliability - Best practices - REL01-BP05
https://docs.aws.amazon.com/wellarchitected/2023-10-03/framework/rel_manage_service_limits_automated_monitor_limits.html

[16] Reliability - Best practices - REL06-BP04
https://docs.aws.amazon.com/wellarchitected/2023-10-03/framework/rel_monitor_aws_resources_automate_response_monitor.html

[17] Performance Efficiency - Design Principles
https://docs.aws.amazon.com/wellarchitected/2023-10-03/framework/perf-dp.html

[18] Performance Efficiency - Best practices - PERF01-BP01
https://docs.aws.amazon.com/wellarchitected/2023-10-03/framework/perf_architecture_understand_cloud_services_and_features.html

[19] Performance Efficiency - Best practices - PERF04-BP06
https://docs.aws.amazon.com/wellarchitected/2023-10-03/framework/perf_networking_choose_workload_location_network_requirements.html

[20] Cost Optimization - Design Principles
https://docs.aws.amazon.com/wellarchitected/2023-10-03/framework/perf-dp.html

[21] Cost Optimization - Best practices - COST05-BP05
https://docs.aws.amazon.com/wellarchitected/2023-10-03/framework/cost_select_service_select_for_cost.html

[22] Cost Optimization - Best practices - COST02-BP03
https://docs.aws.amazon.com/wellarchitected/2023-10-03/framework/cost_govern_usage_account_structure.html

[23] Even Swaps: A Rational Method for Making Trade-offs
https://hbr.org/1998/03/even-swaps-a-rational-method-for-making-trade-offs

[24] Minikube
https://minikube.sigs.k8s.io/docs/start/

[25] Docker
https://www.docker.com/

[26] Locust
https://locust.io/