# CSC/ECE 547 - Cloud Project

# CLOUD-NATIVE SMART CITY INFRASTRUCTURE

**Team member 1:**
**Name:** Viraj Sanap
**Unity ID:** vasanap
**Student ID:** 200540546

**Team member 2:**
**Name:** Tejas Prabhu
**Unity ID:** tprabhu2
**Student ID:** 200478629

***"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."***

# Contents

# 1. Introduction

## 1.1. Motivation

This project provides an opportunity to delve deeply into real-world cloud architectural challenges and solutions, offering a significant leap in learning and understanding of cloud computing paradigms. With urban areas becoming increasingly congested, the idea of smart cities is rapidly gaining traction. Engaging in a project that has real-world applications and the potential to improve the quality of life in cities is highly motivating. The technical challenges posed by the need for real-time data processing, analytics, and delivering actionable insights in a cloud-based smart city infrastructure project are intriguing. The project opens avenues for innovative solutions and prepares for future technological advancements in smart city technologies.

## 1.2. Executive summary

The project entails designing a **cloud-native smart city infrastructure** aimed at managing and analyzing data from various city services including traffic management, waste management, and emergency services. Leveraging the power of cloud computing, the objective is to enable real-time data processing and analytics to provide actionable insights that can significantly improve city operations. The architecture should be scalable, reliable, and capable of integrating with various city service systems. This project is aimed at not only addressing the current challenges cities face in managing services efficiently but also paving the way for future smart city initiatives. The outcome of this project will demonstrate the effectiveness and the potential of cloud computing in transforming urban management and improving the quality of life.

# 2. Problem Description

## 2.1. The problem

The high-level problem being addressed in this project is the lack of a centralized, scalable, and real-time data processing and analytics platform for managing various city services. Urban areas today are grappling with the challenges of managing growing populations, traffic congestions, waste management, and emergency response services.

The traditional systems are often siloed, lack real-time data processing capabilities, and are unable to provide actionable insights that are crucial for improving operational efficiencies and citizen satisfaction. The cloud-based smart city infrastructure project aims to address these challenges by leveraging cloud computing technologies to create a scalable, real-time data processing, and analytics platform that can integrate with various city service systems and provide actionable insights for improving city operations.

## 2.2. Business Requirements

**Operational Excellence:**
BR1: Ensure real-time data processing.
BR2: Provide actionable insights to improve city operations.
BR3: Ensure accurate and timely data collection from various city services.
BR4: Facilitate effective communication between different city departments.
BR5: Facilitate integration capabilities with external systems.
BR6: Enable automated operations.
BR7: Facilitate easy deployment and maintenance of the platform.

**Security:**
BR8: Maintain data confidentiality and integrity across the platform.
BR9: Detect and respond to unauthorized access promptly.
BR10: Ensure secure data transmission across the platform.
BR11: Ensure secure access controls and authentication mechanisms.

**Reliability:**
BR12: Ensure high availability and reliability of the platform.
BR13: Ensure data resiliency and continuity.
BR14: Ensure system resilience during network failures.

**Performance Efficiency:**
BR15: Ensure low latency in data access and analytics.
BR16: Ensure scalability to accommodate growth of data volumes.
BR17: Optimize resource utilization to maximize performance.

**Cost Optimization:**
BR18: Optimize operational costs while ensuring system performance.
BR19: Enable cost monitoring and optimization.

**Compliance:**
BR20: Ensure system compliance with industry standards and regulations.

BR21: Ensure data retention and archiving according to legal and operational requirements.

**Future Adaptability:**
BR22: Ensure the platform supports historical data analysis and reporting.

**Sustainability:**
BR23: Minimize environmental impact through energy-efficient operations.
BR24: Reduce carbon emissions associated with platform operations.
BR25: Ensure compliance with environmental regulations and standards.
BR26: Ensure precise Tenant Identification for resource and cost allocation.

## 2.3. Technical Requirements

**Operational Excellence:**

**BR1: Ensure real-time data processing.**
TR1.1: Implement a real-time data processing engine.
TR1.2: Establish a low-latency data ingestion pipeline.
TR1.3: Design real-time data analytics dashboards.
TR1.4: Set up real-time monitoring for data processing workflows.

These TRs collectively help in achieving the immediate processing and visibility of data which is vital for operational decision-making and response in a smart city context. These TRs are adequate as they cover the core aspects of real-time data handling including processing, ingestion, visualization, and monitoring.

**BR2: Provide actionable insights to improve city operations.**
TR2.1: Establish data analytics and processing workflows.
TR2.2: Develop dashboards for displaying actionable insights.
TR2.3: Set up alerting systems for critical insights.

The trio of TRs covers the process of generating, visualizing, and alerting on insights, which are key to driving operational improvements in city services.

**BR3: Ensure accurate and timely data collection from various city services.**
TR3.1: Implement robust data collection pipelines

TR3.2: Establish data validation and verification processes.
TR3.3: Set up data synchronization mechanisms to ensure timeliness.
TR3.4: Implement monitoring for data collection workflows.

The specified TRs aim to build a robust infrastructure for data collection, validation, synchronization, and monitoring to ensure data accuracy and timeliness.

**BR4: Facilitate effective communication between different city departments.**
TR4.1: Implement a centralized messaging system
TR4.2: Enable API integrations for inter-departmental data sharing.
TR4.3: Establish a secure, real-time communication channel.

These TRs cover the fundamental aspects of inter-departmental communication, providing a framework for secure seamless interaction and data sharing which is essential for coordinated operations.

**BR5: Facilitate integration capabilities with external systems.**
TR5.1: Implement data transformation and mapping tools.
TR5.2: Implement standardized data exchange formats.
TR5.3: Establish secure data transmission channels.

These TRs provide a solid foundation for integrating external systems, ensuring compatibility, standardization, and security in data exchanges, which are vital for leveraging external data and services in a reliable and secure manner.

**BR6: Enable automated operations.**
TR6.1: Implement automated deployment pipelines.
TR6.2: Automate routine maintenance tasks.
TR6.3: Automate system health checks and alerts.
TR6.4: Implement automated scaling based on system load.

These TRs collectively contribute to enhancing operational efficiency, reliability, and the ability to respond rapidly to changing conditions.

**BR7: Facilitate easy deployment and maintenance of the platform.**
TR7.1: Utilize Infrastructure as Code (IaC) for easy deployment.
TR7.2: Implement centralized logging and monitoring.
TR7.3: Implement automated patch management.
TR7.4: Establish a centralized configuration management system.

The TRs chosen for this BR cover the fundamental areas that contribute to ease of deployment and maintenance. They build a framework for simplified deployment and maintenance which is crucial for managing the platform efficiently over time.

## Security:

**BR8: Maintain data confidentiality and integrity across the platform.**

TR8.1: Implement encryption at rest and in transit.
TR8.2: Apply data masking and tokenization.
TR8.3: Implement robust access control mechanisms.

These TRs ensure secure data handling both at rest and during transmission, fulfilling basic security necessities. They collectively provide a structure for access control and data protection, essential for meeting security standards and compliances.

**BR9: Detect and respond to unauthorized access promptly.**
TR9.1: Employ real-time intrusion detection systems.
TR9.2: Implement an incident response plan.
TR9.3: Conduct regular security audits and vulnerability assessments.

The chosen TRs enable proactive detection of and response to security threats, which is crucial for maintaining platform integrity and trustworthiness over time.

**BR10: Ensure Secure Data Transmission Across the Platform.**
TR10.1: Implement encryption for data in transit.
TR10.2: Ensure network isolation for sensitive data processing environments.

These TRs aim at securing data during transmission, a fundamental requirement to prevent data breaches and ensure data integrity, supporting the overall security posture of the platform.

**BR11: Ensure Secure Access Controls and Authentication Mechanisms.**
TR11.1: Implement multi-factor authentication for sensitive system access.
TR11.2: Utilize role-based access control for system resources
TR11.3: Employ simplified and secure user authentication.

The selected TRs are geared towards strong access management and authentication, vital for preventing unauthorized access and ensuring only authorized personnel can access certain data.

## Reliability:

**BR12: Ensure High Availability and Reliability of the Platform.**
TR12.1: Ensure deployment across multiple availability zones.
TR12.2: Ensure load distribution across multiple resources.
TR12.3: Ensure automatic scaling of resources based on demand.
TR12.4: Ensure regular health checks of the system components.

The TRs under this BR cover deployment across multiple zones and regions, load distribution, and automatic scaling, which are fundamental for ensuring continuous service availability and reliability. They represent a comprehensive approach to achieving a fault-tolerant and resilient platform.

**BR13: Ensure Data Resiliency and Continuity.**
TR13.1: Implement automatic backups and snapshots.
TR13.2: Establish a disaster recovery plan.
TR13.3: Utilize deployments across multiple regions for critical services.
TR13.4: Implement data replication across regions.
TR13.5: Employ redundant storage solutions.

The TRs chosen encompass data backup, disaster recovery, and data replication which are crucial for data resiliency and business continuity, ensuring the platform can recover swiftly from potential data loss or system failures.

**BR14: Ensure System Resilience During Network Failures.**
TR14.1: Ensure DNS failover capabilities.
TR14.2: Ensure resilient content delivery.
TR14.3: Ensure edge computing for local processing.
TR14.4: Ensure dedicated network connections.
TR14.5: Ensure network isolation capabilities.

The TRs outlined provide a strategy for maintaining system operations during network disruptions, which is essential for a smart city platform where real-time data and communication are critical for city operations.

## Performance Efficiency:

**BR15: Ensure low latency in data access and analytics.**
TR15.1: Optimize data querying processes to ensure rapid data retrieval.
TR15.2: Utilize efficient data indexing and storage solutions for quick data access.
TR15.3: Employ real-time analytics solutions for immediate insight generation.
TR15.4: Ensure streamlined data processing pipelines for low-latency analytics.
TR15.5: Optimize network configurations to minimize data transmission delays.

The selected TRs aim to optimize data retrieval, storage, and processing to minimize latency. They encompass various aspects necessary for near-real-time data handling, which is crucial for timely decision-making in smart city operations.

**BR16: Ensure scalability to accommodate growth of data volumes**
TR16.1: Employ scalable cloud resources to accommodate varying loads.
TR16.2: Utilize auto-scaling solutions for resource allocation based on demand.
TR16.3: Employ load balancing solutions to distribute traffic and reduce latency.
TR16.4: Optimize data storage solutions for scalable data management.
TR16.5: Implement scalable data processing pipelines to handle increasing data volumes.
TR16.6: Ensure the scalability of data ingestion pipelines to accommodate growing data volumes.

The TRs are intended to ensure the platform can handle increased load efficiently, covering scalable resource allocation, load balancing, and data management. This comprehensive approach ensures that as the city services and data volumes grow, the platform can scale to meet the demand without performance degradation.

**BR17: Optimize resource utilization to maximize performance.**
TR17.1: Implement resource monitoring and optimization tools to ensure efficient resource utilization.
TR17.2: Employ performance tuning practices to optimize resource allocation.
TR17.3: Utilize cost-optimized resource types and configurations to maximize performance.
TR17.4: Ensure optimized data processing pipelines for efficient resource utilization.

The TRs focus on monitoring and optimizing resource utilization, which is crucial for balancing performance with cost. By ensuring efficient resource allocation and identifying performance bottlenecks, the platform can maintain high performance while keeping operational costs in check.

## Cost Optimization:

**BR18: Optimize operational costs while ensuring system performance.**
TR18.1: Implement mechanisms to monitor and analyze resource usage.
TR18.2: Employ cost-effective storage solutions for data.
TR18.3: Set up budget alerts to monitor spending.
TR18.4: Utilize reserved instances for predictable workloads.
TR18.5: Implement spot instances for temporary workloads.
TR18.6: Optimize data transfer costs.

TR18.7: Implement Tenant Identification mechanisms for precise resource and cost allocation.

By implementing monitoring mechanisms and employing cost-effective solutions, the set of TRs help in achieving cost efficiency. The inclusion of Tenant Identification is particularly notable as it enables precise cost allocation, which is a critical aspect of cost optimization.

**BR19: Enable cost monitoring and optimization.**
TR19.1: Establish a framework for continuous cost monitoring.
TR19.2: Implement cost optimization practices across the platform.
TR19.3: Implement cost allocation tags.
TR19.4: Employ cost management tools for budget tracking and forecasting.

Through the establishment of a continuous monitoring framework and the use of cost management tools, these TRs ensure that cost optimization practices are implemented and budget tracking is facilitated, aiding in better financial management and forecasting.

## Compliance:

**BR20: Ensure system compliance with industry standards and regulations.**
TR20.1: Conduct regular compliance audits.
TR20.2: Employ cloud services that adhere to industry standards
TR20.3: Ensure encryption in transit and at rest.
TR20.4: Employ compliant cloud services for sensitive or regulated data.

The TRs collectively ensure the platform's compliance with industry standards and regulations which is vital for legal adherence and trust-building with stakeholders. Regular compliance audits ensure ongoing adherence, while employing compliant cloud services and ensuring encryption cater to data security and privacy requirements stipulated by various regulatory bodies.

**BR21: Ensure data retention and archiving according to legal and operational requirements.**
TR21.1: Establish and maintain data retention policies.
TR21.2: Ensure secure data archiving.
TR21.3: Implement automated data archiving solutions.
TR21.4: Ensure easy retrieval of archived data for auditing purposes.

These TRs aim at ensuring proper data retention and archiving in line with legal and operational requirements. Establishing data retention policies helps in adhering to legal mandates, secure archiving and easy retrieval of data ensure

operational effectiveness during auditing processes, and automated archiving solutions can help in reducing manual errors and operational overheads.

## Future Adaptability:

**BR22: Ensure the platform supports historical data analysis and reporting.**
TR22.1: Establish data warehousing solutions for historical data storage and analysis.
TR22.2: Implement data visualization and reporting tools.
TR22.3: Ensure data analytics platforms can handle large-scale historical data.
TR22.4: Provide APIs for third-party integrations for data analysis and reporting.

The focus of these TRs is to ensure the platform can effectively support historical data analysis and reporting which is crucial for informed decision-making over time. By establishing data warehousing solutions and implementing data visualization tools, the platform can provide comprehensive insights from historical data. Moreover, providing APIs for third-party integrations enhances the flexibility and extensibility of data analysis and reporting capabilities.

## Sustainability:

**BR23: Minimize environmental impact through energy-efficient operations.**
TR23.1: Employ serverless architectures to reduce idle resource usage.
TR23.2: Utilize energy analytics to monitor and optimize energy consumption.
TR23.3: Employ renewable energy sources for cloud operations where feasible.

These TRs are aimed at minimizing energy usage and promoting renewable energy in cloud operations, aligning with modern eco-friendly practices. They cover employing energy-efficient resources, serverless architectures to reduce idle usage, utilizing analytics for energy monitoring, and leveraging renewable energy where possible.

**BR24: Reduce carbon emissions associated with platform operations.**
TR24.1: Utilize carbon footprint monitoring tools to measure emissions.
TR24.2: Optimize data routing to reduce energy consumption and emissions.

The selected TRs focus on measuring and reducing carbon emissions associated with platform operations. Utilizing monitoring tools to measure emissions and optimizing data routing to reduce energy consumption are pragmatic steps towards reducing the platform's carbon footprint.

**BR25: Ensure compliance with environmental regulations and standards.**
TR25.1: Conduct regular environmental compliance audits.

Conducting regular environmental compliance audits using cloud-based GRC platforms provides a structured approach to monitoring and ensuring compliance with environmental regulations and standards.

**BR26: Ensure precise Tenant Identification for resource and cost allocation.**

TR26.1: Implement a multi-tenant architecture that supports tenant-specific data segregation and access controls.

TR26.2: Develop tenant identification protocols that accurately track usage metrics for billing and analytics purposes.

TR26.3: Design and implement a dashboard for administrators to monitor and manage tenant-specific resource usage and cost implications in real-time.

## 2.4. TradeOffs

### 2.4.1. Operational Excellence vs Cost Optimization:

Ensuring real-time data processing (BR1) and automated operations (BR6) may lead to increased operational costs (BR18). Real-time processing and automation often require sophisticated, higher-cost technologies and resources. Balancing between operational excellence and cost optimization might necessitate a thorough analysis to determine the most cost-effective solutions that still meet operational goals.

### 2.4.2. Security vs Performance Efficiency:

Maintaining data confidentiality and integrity across the platform (BR8) might impact the system's performance efficiency (BR15). Implementing robust security measures like encryption and complex access control mechanisms can add computational overhead, potentially affecting system performance. It's crucial to design a system that upholds security standards while still meeting performance efficiency requirements.

### 2.4.3. Reliability vs Cost Optimization:

Ensuring high availability and reliability of the platform (BR12) may lead to increased operational costs (BR18). High availability often requires redundant resources and multi-region deployments, which can escalate

costs. An optimal balance needs to be found to ensure system reliability while keeping costs under control.

#### 2.4.4. Future Adaptability vs Cost Optimization:

Establishing a platform that supports historical data analysis and reporting (BR22) might increase operational costs (BR18). Data warehousing and analytics services can be costly, and as data volumes grow, so do costs. A cost-effective data analytics solution that aligns with future adaptability goals will be crucial.

# 3. Provider Selection

## 3.1. Criteria for choosing a provider

**Service Availability:** Availability of services that align with the Technical Requirements (TRs) specified in Section 2.3. The more services a provider has that match our TRs, the better.

**Cost Effectiveness:** Pricing structure of the services offered by the provider, including any available cost optimization and budget management tools. This aligns with TR18.1 to TR18.7, which focus on cost monitoring and optimization. A provider that offers cost-effective solutions will help in implementing these TRs effectively.

**Performance Efficiency:** Capability of the provider's services to meet the performance requirements such as low latency, high throughput, and scalability. Performance-related TRs like TR15.1 to TR15.5, which emphasize low latency and efficient data processing, and TR16.1 to TR16.6, which focus on scalability, are satisfied by this criterion.

**Security and Compliance:** Security features, certifications, and compliance capabilities of the provider to ensure data confidentiality, integrity, and compliance with industry standards. This criterion aligns with TR8.1 to TR8.3 for maintaining data confidentiality and integrity, TR9.1 to TR9.3 for detecting and responding to unauthorized access, and TR20.1 to TR20.4 for compliance with industry standards.

**Reliability:** The reliability and availability of the provider's services, including their SLAs (Service Level Agreements), redundancy, and failover capabilities.

Reliability is crucial and ties directly to TR12.1 to TR12.4 which are about ensuring high availability and system resilience, and TR13.1 to TR13.5 which focus on data resiliency and continuity.

**Sustainability:** Environmental impact considerations, energy efficiency, and carbon emission reduction capabilities of the provider. This criterion is aligned with TR23.1 to TR23.3 which aim to minimize environmental impact, and TR24.1 to TR24.2 which focus on reducing carbon emissions, ensuring the provider's operations support sustainability goals.

**Community:** Availability of support, documentation, and a strong community which can be beneficial for troubleshooting and gaining insights. While not directly tied to specific TRs, a strong community and support system are indirectly related to the successful implementation of all TRs. They provide the resources and knowledge base that can help troubleshoot and optimize the use of provider services.

**Ecosystem and Integrations:** Compatibility with other tools, systems, and a well-established ecosystem for seamless integration. This criterion ensures that TR5.1 to TR5.3 for integration capabilities with external systems can be met. It also supports TR22.4 which requires APIs for third-party data analysis and reporting integrations.

## 3.2. Provider Comparison

**(Summarized by chatgpt)**

**Table 1: Comparison Based on High-Level Building Blocks**

| Building Blocks | AWS | Azure | GCP |
|---|---|---|---|
| **Data Ingestion** | 1 (Kinesis for real-time data streaming) | 2 (Event Hubs for large-scale event processing) | 3 (Cloud Pub/Sub for event ingestion but less mature than AWS and Azure) |
| **Data Processing** | 1 (Broad range of services like EMR, Lambda) | 2 (HDInsight and Azure Functions for diverse processing needs) | 3 (Dataflow and Dataproc, but with a narrower focus compared to AWS and Azure) |

| | | | |
|---|---|---|---|
| **Data Analytics & ML** | 1 (SageMaker, Redshift for advanced analytics) | 3 (Azure ML less comprehensive than AWS) | 2 (Strong ML offerings with AI Platform, BigQuery) |
| **Security & Compliance** | 1 (Comprehensive security services and certifications) | 2 (Strong compliance but slightly behind AWS in terms of range) | 3 (Robust security but fewer certifications than AWS and Azure) |
| **Multi-Tenancy Support** | 1 (Advanced IAM and Cognito for secure multi-tenancy) | 2 (Azure Active Directory and multi-tenant features) | 3 (Solid support, but less extensive than AWS and Azure) |
| **IoT Solutions** | 1 (IoT Core, FreeRTOS, Greengrass for extensive IoT ecosystem) | 2 (IoT Hub and IoT Edge for integrated solutions) | 3 (Cloud IoT Core, but less comprehensive IoT solutions) |
| **Scalability & Reliability** | 1 (Global infrastructure with extensive scalability) | 2 (Large global presence but slightly behind AWS) | 3 (Good scalability but a smaller global footprint) |
| **Integration & APIs** | 1 (Extensive set of APIs and integration options) | 2 (Strong integration capabilities, especially with Microsoft products) | 3 (Good integration but not as extensive as AWS or Azure) |

**[1] [2] [3]**

**Table 2: Comparison Based on Provider Selection Criteria**

| Criteria | AWS Rank & Justification | Azure Rank & Justification | GCP Rank & Justification |
|---|---|---|---|
| **Service Availability** | 1 (200+ fully-featured services) | 2 (200+ products but slightly less comprehensive than AWS) | 3 (100+ products, focuses on specific areas like ML, big data) |

| | | | |
|---|---|---|---|
| **Cost Effectiveness** | 1 (Flexible pricing models and cost optimization tools) | 2 (Flexible payment models, but less granular than AWS) | 3 (Competitive but less flexible compared to AWS and Azure) |
| **Performance Efficiency** | 1 (High-performance computing options and low-latency network) | 2 (Robust performance but slightly behind AWS) | 3 (Good performance but not on par with AWS and Azure) |
| **Reliability** | 1 (Highest number of availability zones ensuring reliability) | 2 (Strong reliability but fewer availability zones than AWS) | 3 (Good reliability, but lags behind AWS and Azure) |
| **Sustainability** | 1 (Commitment to sustainability and energy-efficient operations) | 2 (Sustainable practices but less comprehensive than AWS) | 3 (Efforts in sustainability but not as advanced as AWS and Azure) |
| **Community and Support** | 1 (Extensive community support and documentation) | 2 (Strong community, especially among enterprise users) | 3 (Growing community but not as extensive as AWS and Azure) |
| **Ecosystem and Integrations** | 1 (Wide range of integrations and compatibility with various tools) | 2 (Strong ecosystem, especially with Microsoft products) | 3 (Solid ecosystem but more focused on open-source and DevOps) |

## 3.3. The final selection

Given the broad service availability, market presence, and alignment with our TRs, AWS is selected as the preferred provider. AWS's comprehensive service offerings, including its well-established infrastructure, security, and machine learning services, provide a strong foundation to meet our diverse and specific technical requirements.

### 3.3.1. The list of services offered by the winner

**Operational Excellence:**

TR1.1: Amazon Kinesis for real-time data processing.
TR1.2: AWS Direct Connect for low-latency data ingestion.
TR1.3: Amazon QuickSight for designing real-time analytics dashboards.
TR1.4: Amazon CloudWatch for setting up real-time monitoring.

**Actionable Insights:**

TR2.1: AWS Glue for establishing data analytics workflows.
TR2.2: Amazon QuickSight for developing insightful dashboards.
TR2.3: Amazon SNS for alerting systems.

**Data Collection and Accuracy:**

TR3.1: AWS Data Pipeline for robust data collection.
TR3.2: AWS Lambda for data validation processes.
TR3.3: AWS Database Migration Service for data synchronization.
TR3.4: Amazon CloudWatch for monitoring data collection.

**Effective Communication:**

TR4.1: Amazon SNS for centralized messaging.
TR4.2: Amazon API Gateway for API integrations.
TR4.3: AWS Direct Connect for secure communication.

**External System Integration:**

TR5.1: AWS AppFlow for data transformation and mapping.
TR5.2: Amazon S3 for standardized data exchange.
TR5.3: AWS VPN for secure data transmission.

**Automated Operations:**

TR6.1: AWS CodePipeline for automated deployments.
TR6.2: AWS Systems Manager for automating maintenance tasks.
TR6.3: AWS CloudWatch for automated system health checks.
TR6.4: AWS Auto Scaling for load-based automation.

**Easy Deployment and Maintenance:**

TR7.1: AWS CloudFormation for Infrastructure as Code.
TR7.2: Amazon CloudWatch for centralized logging.
TR7.3: AWS Systems Manager for automated patch management.
TR7.4: AWS Config for configuration management.

**Security:**

TR8.1: AWS KMS for encryption at rest and in transit.
TR8.2: Amazon RDS for data masking.
TR8.3: AWS IAM for robust access control.

**Unauthorized Access Detection:**

TR9.1: Amazon GuardDuty for intrusion detection.
TR9.2: AWS Incident Manager for incident response.
TR9.3: AWS Security Hub for regular audits.

**Secure Data Transmission:**

TR10.1: AWS Certificate Manager for encryption in transit.
TR10.2: Amazon VPC for network isolation.

**Secure Access and Authentication:**

TR11.1: AWS IAM for multi-factor authentication.
TR11.2: AWS IAM for role-based access control.
TR11.3: Amazon Cognito for user authentication.

**Reliability:**

TR12.1: Amazon EC2 across multiple availability zones.
TR12.2: AWS Load Balancer for load distribution.
TR12.3: AWS Auto Scaling for resource scalability.
TR12.4: AWS CloudTrail for regular health checks.

**Data Resiliency and Continuity:**

TR13.1: AWS Backup for implementing automated backups and snapshots.
TR13.2: Amazon Route 53 and AWS CloudEndure for a comprehensive disaster recovery approach.
TR13.3: Amazon EC2 and Amazon S3 used across multiple regions for resilient service deployment.

TR13.4: AWS Database Migration Service for replicating data across regions.
TR13.5: Amazon S3 for redundant storage solutions, ensuring data availability.

**System Resilience During Network Failures:**

TR14.1: Amazon Route 53 for DNS failover capabilities to maintain consistent service access.
TR14.2: Amazon CloudFront for resilient and efficient content delivery.
TR14.3: AWS Outposts for edge computing solutions, reducing reliance on central networks.
TR14.4: AWS Direct Connect for establishing dedicated network connections for critical operations.
TR14.5: Amazon VPC for creating isolated network environments to enhance security and control.

**Performance Efficiency:**

TR15.1 - TR15.5: Amazon RDS for optimized data querying, Amazon ElastiCache for efficient data storage and retrieval, AWS Lambda for real-time analytics, and AWS Direct Connect to minimize data transmission delays.

**Scalability:**

TR16.1 - TR16.6: Amazon EC2 Auto Scaling for dynamic resource allocation, Elastic Load Balancing for traffic distribution, Amazon S3 for scalable storage solutions, and AWS Glue for scalable data processing pipelines.

**Resource Utilization and Performance:**

TR17.1 - TR17.4: AWS Cost Explorer for monitoring resource usage, AWS Compute Optimizer for resource optimization, Amazon EC2 Spot Instances for cost-effective resource utilization, and AWS Step Functions for efficient data processing.

**Cost Optimization:**

TR18.1 - TR18.7: AWS Cost Management for comprehensive cost analysis, Amazon S3 for efficient storage solutions, AWS Budgets for budget tracking and alerts, AWS Reserved Instances for predictable workloads, AWS Spot Instances for cost savings on temporary workloads,

AWS Direct Connect for reducing data transfer costs, and AWS Control Tower for precise resource and cost allocation.

**Cost Monitoring and Optimization:**

TR19.1 - TR19.4: AWS Budgets for continuous cost monitoring, AWS Trusted Advisor for optimization recommendations, AWS Cost Allocation Tags for detailed cost tracking, and AWS Cost and Usage Report for budget management and forecasting.

**Compliance:**

TR20.1: AWS Audit Manager for conducting regular compliance audits.
TR20.2: AWS Compliance Solutions for employing cloud services that adhere to industry standards.
TR20.3: AWS Key Management Service (KMS) for ensuring encryption in transit and at rest.
TR20.4: AWS Identity and Access Management (IAM) for using compliant cloud services for sensitive or regulated data.

**Data Retention and Archiving:**

TR21.1: AWS Data Lifecycle Manager for establishing and maintaining data retention policies.
TR21.2: Amazon S3 Glacier for secure data archiving.
TR21.3: AWS Storage Gateway for implementing automated data archiving solutions.
TR21.4: Amazon S3 for easy retrieval of archived data for auditing purposes.

**Historical Data Analysis and Reporting:**

TR22.1: Amazon Redshift for establishing data warehousing solutions.
TR22.2: Amazon QuickSight for implementing data visualization and reporting tools.
TR22.3: AWS Athena for ensuring the data analytics platform can handle large-scale historical data.
TR22.4: Amazon API Gateway for providing APIs for third-party integrations for data analysis and reporting.

**Sustainability:**

TR23.1: AWS Lambda for employing serverless architectures to reduce idle resource usage.

TR23.2: AWS Cost Explorer for utilizing energy analytics to monitor and optimize energy consumption.
TR23.3: AWS's various sustainability initiatives for employing renewable energy sources in cloud operations.

**Carbon Emission Reduction:**

TR24.1: AWS's carbon footprint monitoring tools for measuring emissions.
TR24.2: Amazon CloudFront and AWS Route 53 for optimizing data routing to reduce energy consumption and emissions.

**Environmental Compliance:**

TR25.1: AWS Compliance Solutions for conducting regular environmental compliance audits.

**Tenant Identification for Resource and Cost Allocation:**

TR26.1: AWS IAM for implementing a multi-tenant architecture.
TR26.2: AWS Cost Allocation Tags for developing tenant identification protocols.
TR26.3: Amazon QuickSight for designing and implementing dashboards for tenant-specific resource monitoring.

## Service Descriptions:

**(refined by chatgpt)**

**Amazon Kinesis:**
AWS Kinesis is a scalable, managed service offered by Amazon Web Services that allows for the real-time processing of streaming data. It enables developers to ingest large amounts of data, such as logs, events, and video streams, from various sources into the AWS ecosystem for analytics, machine learning, and other applications.

Kinesis is designed to handle high-throughput, high-volume data streams, offering the ability to process data on the fly, rather than in batches. This facilitates immediate analysis and decision-making, which is critical in scenarios like financial transactions, social media feeds, or connected devices in IoT ecosystems.

The service is divided into four primary components: Kinesis Data Streams, Kinesis Data Firehose, Kinesis Data Analytics, and Kinesis

Video Streams. Each serves a unique purpose—from capturing and storing data streams to enabling real-time analytics on the streamed data.

Kinesis integrates seamlessly with other AWS services, making it a pivotal part of a cloud-based data pipeline. Its ability to scale automatically with the incoming data volume ensures that applications can handle any load and that data is available when needed. With Kinesis, businesses can unlock insights from their streaming data without the complexity of setting up and managing the underlying infrastructure.[4]

**Amazon Aurora:**
Amazon Aurora is a fully managed relational database engine that's compatible with MySQL and PostgreSQL. It combines the speed and reliability of high-end commercial databases with the simplicity and cost-effectiveness of open-source databases. Aurora is part of Amazon Relational Database Service (RDS) and is designed to deliver performance up to five times faster than MySQL and three times faster than PostgreSQL, making it ideal for applications that need high throughput and low latency.

Aurora stands out due to its distributed, fault-tolerant, and self-healing storage system, which automatically replicates data across multiple Availability Zones and continuously backs up data to Amazon S3. This design enhances both availability and durability, with Aurora automatically recovering from physical storage failures; database recovery is instantaneous, and there's no need for crash recovery or rebuild times.

Another key feature of Aurora is its scalability. It can start with a small instance and automatically scale up to accommodate growing data up to 128 TiB (for MySQL) or 64 TiB (for PostgreSQL), making it suitable for a wide range of applications, from enterprise-level to mobile and web-based applications.

Aurora provides several advanced security features such as encryption at rest and in transit, and it integrates with AWS Identity and Access Management (IAM) for fine-grained access control. Its performance, combined with availability and security features, make Aurora a compelling choice for businesses looking to leverage the power of cloud computing for their relational database needs.[5]

**Amazon QuickSight:**
Amazon QuickSight is a cloud-based business intelligence (BI) service provided by AWS, designed to make data analysis and visualization fast,

easy, and cost-effective[6]. It allows users to create and publish interactive dashboards, perform ad-hoc analysis, and get business insights from their data using a serverless architecture. QuickSight seamlessly integrates with a wide array of data sources in the cloud and on-premises, including relational databases, file-based data, and third-party APIs.

A standout feature of QuickSight is its pay-per-session pricing model, which makes it accessible for all levels of users within an organization, from casual viewers to data scientists. It offers robust data preparation tools, enabling users to clean and transform data without needing to write code. The service also incorporates machine learning capabilities, like anomaly detection and forecasting, enabling more advanced analysis.

QuickSight is designed for scalability and security. It can handle large datasets and high concurrency, providing fast, responsive insights even with large numbers of users. Its integration with AWS's security infrastructure ensures that data and analytics are protected with industry-standard encryption and access controls.

With its user-friendly interface, QuickSight empowers users to visualize and interact with their data effectively. It's suitable for a wide range of business intelligence applications, from dashboards and reports to deeper analytics, catering to businesses of all sizes and across various industries.

**AWS Simple Queue Service (SQS):**
AWS Simple Queue Service (SQS) is a fully managed message queuing service offered by Amazon Web Services[7]. It provides a secure, durable, and available host for storing messages as they travel between different application components or microservices, thereby decoupling and scaling microservices, distributed systems, and serverless applications.

SQS eliminates the complexity and overhead associated with managing and operating message-oriented middleware, allowing developers to focus on building applications. It supports two types of message queues: Standard Queues, which offer maximum throughput, best-effort ordering, and at-least-once delivery; and FIFO (First-In-First-Out) Queues, which ensure messages are processed exactly once, in the exact order they are sent.

A key benefit of SQS is its high scalability and reliability. It can handle virtually unlimited numbers of messages per queue and retain messages

from a few seconds up to 14 days. This makes it ideal for applications that need to process large volumes of messages without loss or downtime.

SQS also offers features like message delay, message timers, and dead-letter queues, enhancing message processing based on specific requirements. Additionally, it integrates seamlessly with other AWS services like AWS Lambda and Amazon S3, enabling the creation of event-driven architectures.

With its simple API, SQS provides a robust solution for handling message queuing tasks, ensuring secure, reliable, and timely communication between different parts of a cloud application. This service is instrumental in building scalable, decoupled, and efficient cloud-native applications.

# 4. The first design draft

## 4.1. The basic building blocks of the design

The design draft for the smart city platform leverages Amazon AWS services as foundational elements to meet our project's technical requirements (TRs). Each service is chosen for its ability to address specific aspects of the system, from real-time processing to security and sustainability.

**1. Data Ingestion and Processing:**

**Amazon Kinesis (TR1.1, TR1.2, BR1):**
Handles real-time data ingestion from city sensors and devices, ensuring low-latency data pipelines for immediate processing.

**AWS Lambda (TR1.1, TR1.4, BR1, BR23):**
Manages real-time data processing and supports serverless architecture, minimizing idle resource usage and contributing to energy-efficient operations.

**2. Data Storage:**

**Amazon S3 (TR1.3, TR22.1, BR22):**
Serves as a data lake for unstructured or semi-structured data, facilitating analytics, long-term storage, and historical data analysis.

**Amazon Aurora (TR1.3, TR2.1, BR12):**

Provides a relational database for structured data requiring complex queries, supporting high availability and reliability.

**3. Backend Services:**

**Amazon EKS (TR4.2, TR5.1, BR4, BR5):**
Runs containerized applications forming the backend, facilitating API integrations for inter-departmental data sharing and external system integrations.

**4. API Management and Integration:**

**Amazon API Gateway (TR4.1, TR4.2, BR4):**
Acts as an interface for external and internal APIs, managing secure access and communication between city departments and services.

**Amazon SQS:** As an intermediary message queuing service, SQS can be positioned between Kinesis and Lambda to decouple and manage the message load. This allows for additional flexibility and reliability in processing, aligning with the requirements for robust data flow management (BR1) and systematic message handling (TR1.1).

**5. Data Analytics and Visualization:**

**Amazon QuickSight (TR1.3, TR2.2, BR2):**
Provides data analytics and visualization capabilities for creating interactive dashboards, delivering actionable insights for city operations.

**6. Security:**

**AWS IAM (TR8.3, TR11.1, BR8, BR11):**
Ensures robust access control, allowing only authorized users and services to access specific AWS resources, maintaining data confidentiality and integrity.

**Amazon Route53 and CloudFront (TR14.1, TR14.2, BR14):**
Manage DNS and content distribution with additional security features, ensuring system resilience during network failures.

**AWS Certificate Manager:** Works in conjunction with CloudFront and Route53 to manage SSL/TLS certificates, ensuring encrypted and secure communication, which is paramount for maintaining the security posture (TR11.1) and user trust (BR11).

**7. Network Configuration and Isolation:**

**AWS VPC (TR10.2, BR10):**
Provides network isolation for sensitive data processing environments, ensuring secure data transmission across the platform.

Each of these AWS services is chosen to meet specific technical requirements and business requisites. The architecture's design ensures the platform's operational excellence, security, reliability, performance efficiency, cost optimization, compliance, future adaptability, and sustainability.

## 4.2. Top-level, informal validation of the design

In the holistic design of our smart city platform, we have meticulously selected AWS services that serve as the structural and functional pillars, ensuring they address the core building blocks: data flow management, computing architecture, data visualization and analytics, messaging and communication, API management, security, database management, elasticity and load balancing, and cost management. The trade-offs, rationale, and strategic placement of these services are considered to ensure that the system operates cohesively and effectively under various conditions that a dynamic urban environment presents.

**1. Data Ingestion and Processing:**

**Amazon Kinesis** has been incorporated to address the requirements for real-time data ingestion (TR1.1, TR1.2, BR1). It's chosen for its proven capability to handle large streams of data with low latency, a necessity for the immediacy required in smart city operations.[8] The service allows the platform to quickly process and react to live data from sensors, ensuring timely responses to city-wide events.

**AWS Lambda** complements Kinesis by providing the computational logic for real-time data processing (TR1.1, TR1.4, BR1, BR23). Its serverless nature minimizes resource wastage, aligning with sustainability goals (BR23). Lambda's event-driven model ensures that processing power scales with demand, offering a cost-effective solution that directly aligns with the need for operational excellence in processing workflows.[9]

**2. Data Storage:**

**Amazon S3** is the storage cornerstone, acting as a data lake for various forms of data (TR1.3, TR22.1, BR22). It is selected for its durability, scalability, and data lifecycle management capabilities, facilitating not just current analytical needs but also future-proofing the platform for historical data analysis.

**Amazon Aurora** is the database service of choice for structured data (TR1.3, TR2.1, BR12), due to its high performance and availability. Its selection over traditional databases is justified by the requirement for a managed, scalable solution that provides the resilience and reliability necessary for mission-critical city data operations.

**3. Backend Services:**

**Amazon EKS** is utilized for running the backend services (TR4.2, TR5.1, BR4, BR5), which are essential for the smart city platform's core functionality. It supports containerization, which is key for a microservices architecture, allowing for scalable, resilient, and manageable application deployment that facilitates seamless API integrations and inter-service communication.

**4. API Management and Integration:**

**Amazon API Gateway** is the facilitator for all API interactions (TR4.1, TR4.2, BR4), chosen for its ability to provide secure, scalable, and managed API access. It enforces the necessary access control, rate limiting, and API monitoring, which are integral for maintaining the platform's security and ensuring efficient communication between city departments and external services.

**Amazon SQS** is introduced into the data flow to act as a buffer and message queue between Kinesis and Lambda (TR1.5), which adds a layer of resilience by decoupling the ingest and processing components. This ensures that if Lambda functions are temporarily overwhelmed due to a spike in data or if there's a need for scheduled maintenance, incoming data is not lost but queued for subsequent processing, enhancing the system's reliability (BR1) and ensuring consistent data flow management (TR1.1).

**5. Data Analytics and Visualization:**

**Amazon QuickSight** is selected for data analytics and visualization (TR1.3, TR2.2, BR2), allowing stakeholders to derive actionable insights through interactive dashboards. The choice is based on QuickSight's deep integration with AWS data services and ease of use, enabling city officials to make data-driven decisions without the complexity of managing a business intelligence infrastructure.

**6. Security:**

**AWS IAM** is the security foundation (TR8.3, TR11.1, BR8, BR11), ensuring that access to AWS resources is tightly controlled and monitored. IAM's comprehensive policy management capabilities are central to enforcing the principle of least privilege, a key security best practice.

**Amazon Route53** and **CloudFront**, along with **AWS Certificate Manager**, work in tandem to ensure resilient DNS services (TR14.1) and secure content delivery (TR14.2), which are paramount for the system's responsiveness and security (BR14). Their inclusion is strategic for defending against DDoS attacks and ensuring high availability across the platform.

**7. Network Configuration and Isolation:**

**AWS VPC** is the network isolation mechanism (TR10.2, BR10), providing a private network space in the cloud. The smart city platform leverages VPC to create segmented networks for sensitive data processing, ensuring that data in transit within the platform is shielded from external threats, thereby bolstering the overall security posture.

In summary, each AWS service in this architecture is pivotal to creating a solution that is not only technically robust and secure but also efficient, cost-effective, and sustainable. The services collectively form an ecosystem that meets the high demands of a smart city platform, ensuring that it can operate seamlessly, scale as needed, and evolve over time.

### 4.3. Action items and rough timeline

**SKIPPED**

# 5. The second design

### 5.1. Use of the Well-Architected framework

**(Summarized by ChatGPT)**

The AWS Well-Architected Framework offers strategic guidance for architects shaping the cloud-native infrastructure of a smart city. Comprising six critical pillars—Operational Excellence, Performance Efficiency, Cost Optimization, Reliability, Security, and Sustainability—it serves as a compass to build robust, scalable, and secure systems on Amazon Web Services (AWS). The steps for each pillar are:

1. Operational Excellence:

- Perform Operations as Code: Automate operational tasks for efficiency.
- Enable Frequent, Reversible Changes: Implement continuous deployment for agility.
- Refine Operational Procedures: Continuously improve procedures for optimal performance.
- Anticipate Failure: Design fault-tolerant systems for uninterrupted services.
- Learn from Operational Failures: Use incidents as learning opportunities.

2. Performance Efficiency:

- Leverage Advanced Technologies: Utilize IoT, AI, and analytics for improved services.
- Global Scalability: Design for consistent service availability across locations.
- Serverless Architectures: Reduce operational overhead with serverless models.
- Encourage Experimentation: Foster innovation for continual service enhancement.
- Mechanical Sympathy: Optimize applications for better resource utilization.

3. Cost Optimization:

- Implement Financial Management: Monitor and optimize spending on infrastructure.
- Adopt Consumption Model: Manage costs effectively with a pay-as-you-go model.
- Measure Overall Efficiency: Assess and optimize city services' efficiency.
- Attribute Expenditure: Identify and eliminate unnecessary spending.

4. Reliability:

- Automated Failure Recovery: Design systems for automated recovery.
- Test Recovery Procedures: Regularly validate recovery procedures.
- Horizontal Scalability: Enhance workload availability and reliability.
- Capacity Planning Automation: Efficiently manage infrastructure needs.
- Automate Change Management: Use automation to minimize disruptions.

5. Security:

- Strong Identity Foundation: Implement robust access management.
- Traceability: Ensure auditability of activities for security compliance.
- Layered Security Approach: Apply security measures across all infrastructure layers.
- Automate Security Best Practices: Employ automation for best practice enforcement.
- Data Protection: Secure data in transit and at rest to protect citizen information.

6. Sustainability:

- Understand Environmental Impact: Assess the environmental impact of infrastructure.
- Establish Sustainability Goals: Set clear initiatives for sustainable infrastructure.
- Maximize Resource Utilization: Optimize resource usage to reduce waste.
- Adopt Efficient Technologies: Embrace energy-efficient hardware and software.
- Managed Services for Efficiency: Use managed services for operational efficiency.
- Reduce Downstream Impact: Minimize environmental impact of cloud workloads.

## 5.2. Discussion of pillars

**(Summarized by chatgpt)**

### 5.2.1. Performance Efficiency

The Performance Efficiency pillar of the AWS Well-Architected Framework is a critical component that focuses on the efficient utilization of computing resources to meet system requirements and adapt to changes in demand and technology. This pillar ensures that cloud architectures deliver sustained performance over time, a challenging feat in traditional on-premises environments.

**Design Principles**

There are five key design principles for performance efficiency in the cloud:

**Democratize Advanced Technologies:** Simplify the implementation of advanced technologies by using them as cloud services, allowing teams to focus on product development instead of resource management.

**Go Global in Minutes:** Deploy workloads in multiple AWS Regions to provide low latency and better customer experience at minimal cost.

**Use Serverless Architectures:** Eliminate the need to manage physical servers for traditional compute activities, reducing operational burden and transactional costs.
**Experiment More Often:** Utilize virtual and automatable resources for comparative testing using different types of instances, storage, or configurations.
**Consider Mechanical Sympathy:** Use technologies that align best with workload goals, considering factors like data access patterns when selecting databases or storage approaches.

**Best Practice Areas**
The best practices in the Performance Efficiency pillar are spread across five areas:
**Architecture Selection:** Tailoring solutions to specific workloads, combining multiple approaches, and leveraging AWS's variety of resources and configurations.
**Compute and Hardware:** Choosing the right compute resources based on application design, usage patterns, and configuration settings.
**Data Management:** Selecting appropriate data management solutions considering data type, access patterns, throughput requirements, and frequency of access and updates.
**Networking and Content Delivery:** Optimizing network solutions based on latency, throughput requirements, jitter, and bandwidth. AWS offers various networking features to optimize network traffic and reduce network distance or jitter.
**Process and Culture:** Implementing infrastructure as code, using continuous integration/deployment pipelines, setting up well-defined metrics, performing automatic performance tests, generating load, ensuring performance visibility, and utilizing visualization techniques for performance analysis.

### 5.2.2. Reliability

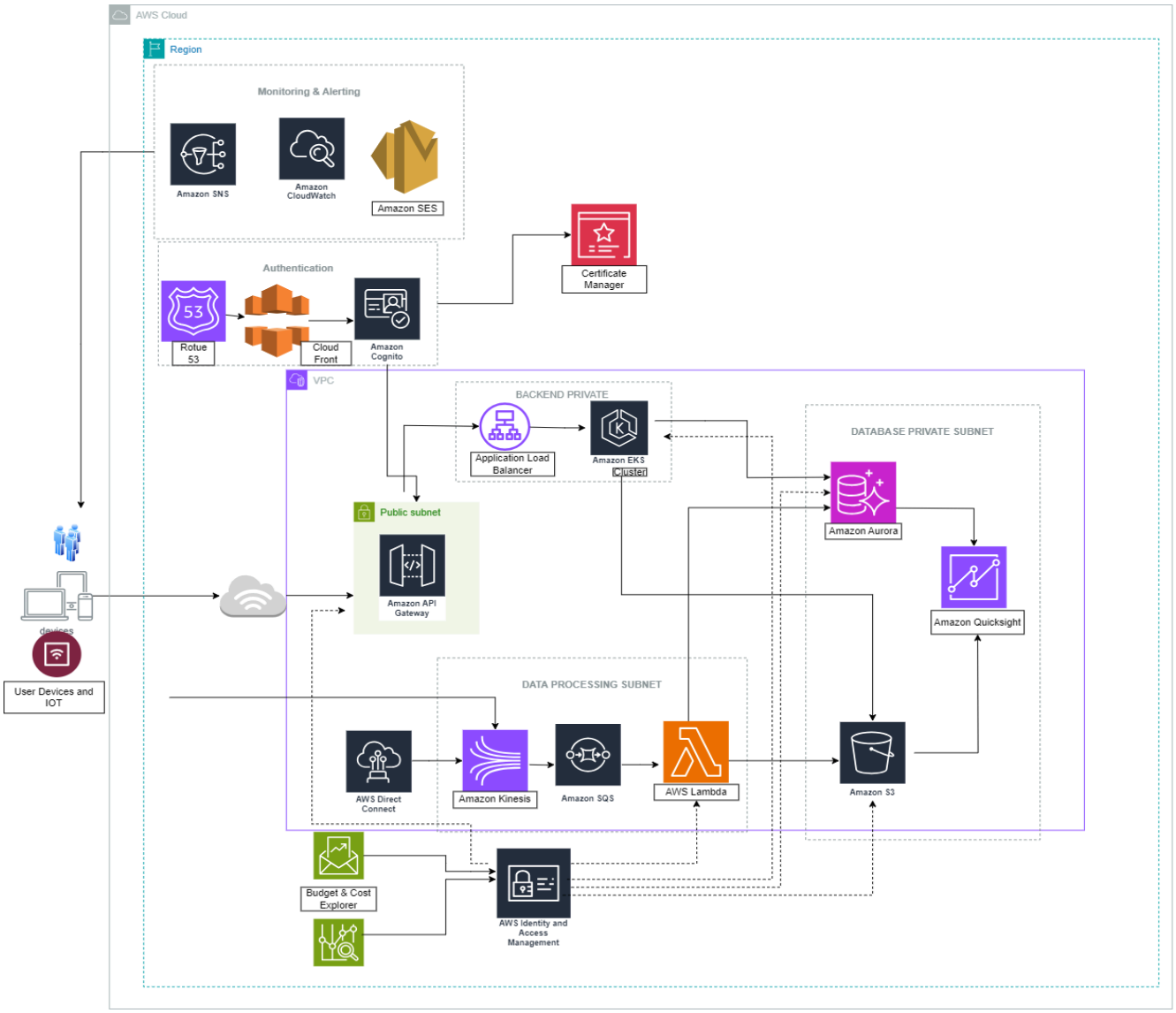
The Reliability pillar of the AWS Well-Architected Framework is a crucial aspect that ensures the ability of systems to perform their intended functions under varying operational conditions. This pillar is especially significant in cloud environments, where reliability can be more challenging due to potential single points of failure, lack of automation, and limited elasticity. The Reliability pillar provides guidance for designing, delivering, and maintaining reliable AWS environments.

**Design Principles**
The key design principles for reliability in the cloud include:
**Automatically Recover from Failure:** Implementing automation to monitor key performance indicators (KPIs) and trigger automated recovery processes.

**Test Recovery Procedures:** Regularly testing how a workload fails and validating recovery procedures to uncover and fix failure pathways.
**Scale Horizontally:** Distributing requests across multiple, smaller resources to mitigate the impact of single failures.
**Stop Guessing Capacity:** Using cloud capabilities to monitor demand and workload utilization, automating resource adjustments to meet demand effectively.
**Manage Change Through Automation:** Implementing changes to infrastructure using automation for better management and tracking.

**Best Practice Areas**
The Reliability pillar encompasses four primary best practice areas:
**Foundations:** Ensuring foundational requirements like sufficient network bandwidth are in place. In AWS, many of these requirements are integrated into the cloud infrastructure.
**Workload Architecture:** Design decisions for both software and infrastructure impact workload reliability. AWS SDKs and APIs aid in implementing reliability best practices.
**Change Management:** Anticipating and accommodating changes in workload or environment, including demand spikes and internal updates like feature deployments and security patches.
**Failure Management:** Developing a proactive stance towards failures, enabling workloads to detect and automatically address failures, thereby maintaining availability and functionality.

## 5.3. Use of Cloudformation diagrams




[10] [11]

## 5.4. Validation of the design

### 1. Data Ingestion and Processing:

**Amazon Kinesis:**
Kinesis is the preferred service for real-time data ingestion, chosen over alternatives like AWS IoT Core or direct ingestion to a database because of its ability to handle high throughput and large volumes of data from various city

sensors. It's specifically designed for stream processing, making it apt for the initial capture of IoT and user device data. Kinesis feeds into AWS Lambda for processing, ensuring that data is not only ingested but also acted upon in real-time, facilitating immediate responsiveness required by smart city applications.

**AWS Lambda:**
Lambda is selected for its event-driven model, which excels over EC2, ECS, or EKS for tasks that require immediate execution in response to events. The use of Lambda allows for scaling precisely with the event load, which can vary significantly, ensuring that compute resources are utilized efficiently. The processed data can be routed to different storage solutions, like S3 for unstructured data or Amazon Aurora for transactional needs, based on the processing logic defined within the Lambda functions[12].

## 1. Data Storage:

**Amazon S3:**
S3 is used as a data lake due to its high durability and scalability. It is chosen over Amazon Elastic Block Store (EBS) or Amazon Elastic File System (EFS) for its ability to store vast amounts of unstructured data cost-effectively and serve as a central repository accessible by various analytical and processing services. S3 integrates with Amazon QuickSight for analytics, Amazon Athena for ad-hoc querying, and is also accessible by Lambda for further processing needs, providing a flexible and scalable storage solution.[13]

**Amazon Aurora:**
Aurora provides the performance and availability of high-end commercial databases with the simplicity and cost-effectiveness of open-source databases. It is preferred over Amazon RDS for standard use cases due to its better performance, scaling capabilities, and seamless failover mechanisms, which are critical for the smart city platform's database layer.

## 2. Backend Services:

**Amazon EKS:**
EKS is the service of choice for managing containerized applications that form the backend of the platform. EKS is favored over AWS ECS due to its community support, flexibility, and alignment with cloud-native practices. It provides the orchestration required to run complex microservices that can interact with the database and storage layers efficiently. EKS serves as the operational hub, with the API Gateway acting as the entry point to these backend services, routing requests to the appropriate services within the EKS clusters.

## 3. API Management and Integration:

**Amazon API Gateway:**
API Gateway is the intermediary that connects the frontend and backend services, chosen for its managed service offering that includes features such as throttling, monitoring, and security. Unlike direct service-to-service communication, API Gateway provides a single point of governance and control for API traffic. It ties together services like Lambda, EKS, and data storage endpoints, providing a structured and secure way for applications to access backend services.

**Amazon SQS:**
Amazon SQS is a critical component in our smart city platform's data processing architecture. The decision to include SQS is the result of a careful trade-off analysis that considers system complexity against the benefits of resilience, reliability, and message management. It decouples the data ingestion component from the processing component. This means that if Lambda is temporarily unable to process incoming data due to a spike in volume or a maintenance event, Kinesis can continue to send data to SQS[14]. This queue then holds the data until Lambda is ready to process it. SQS helps in smoothing out the bursts of data inflow, a scenario common in smart city operations where sensor readings can spike due to various events. Should a Lambda function fail to process a message correctly, SQS can be configured to send that message to a dead letter queue. This allows for the isolation and subsequent analysis of problematic data, which is important for maintaining data quality and system reliability.

## 4. Data Analytics and Visualization:

**Amazon QuickSight:**
QuickSight is utilized for its fully managed data visualization service capabilities. It's chosen over other AWS offerings like Amazon Elasticsearch Service because it provides an easy-to-use interface for creating dashboards without the overhead of managing the underlying infrastructure. QuickSight integrates directly with S3 and Aurora, pulling in data to create meaningful visualizations that aid in decision-making and planning for city officials.

## 5. Security:

**AWS IAM:**

IAM is the cornerstone for managing security credentials and permissions across AWS services. It's selected due to its ability to provide fine-grained access control to resources in AWS, ensuring that the principle of least privilege is adhered to across the platform. IAM policies are applied to all AWS services in use, including Kinesis, Lambda, EKS, and API Gateway, to enforce security at every layer of the architecture.

## 6. Network Configuration and Isolation:

**AWS VPC:**
VPC is used to create a secure and isolated network environment. It's preferred over a default network setup for its capability to define private IP address ranges, set up route tables, network gateways, and configure subnets, which are all necessary for the secure operation of a smart city platform. The rationale behind this subnet distribution is to enhance security by limiting public access to sensitive data processing and storage resources. By using VPC endpoints, the architecture ensures internal AWS traffic for services like S3 does not traverse the open internet, reducing exposure to external threats. This setup also allows for fine-grained network policies, such as security groups and network ACLs, providing a robust defense-in-depth approach. Additionally, this configuration supports compliance with regulations that dictate data must not leave a controlled environment, ensuring integrity and confidentiality.

## 7. Load Balancing Strategy for Smart City Platform:

We chose ALB for its advanced routing capabilities and integration with AWS services over other solutions like NLB or direct service invocation. The tradeoff is the slightly higher latency due to complex routing logic, which is justified by the need for intelligent traffic distribution.ALB sits between the API Gateway and the EKS cluster. While API Gateway handles throttling and authorization, ALB ensures that the requests passed through are effectively balanced across the containers in the EKS cluster.

**Load Definition:** In our Smart City Platform, the load is defined as the number of active connections and requests being handled by the system at any given time. This encompasses API requests, data processing tasks, and user queries.
**Load Metric:** The primary load metrics we focus on are the Request Count Per Target and Latency for API Gateway, and CPU Utilization and Memory Usage for the EKS cluster. These metrics help us monitor the system's health and responsiveness.

**Points to Measure Load:**

Ingress Point (Kinesis): The load is measured by the input data rate, specifically the number of records per second.

API Gateway: Measures the number of incoming requests, ensuring they stay within the threshold to prevent throttling.

EKS Cluster: Monitors resource utilization across the container instances to ensure they are not overwhelmed.

**Goal and Criterion:**
Our primary goal is to maintain system responsiveness below a threshold of 200 ms for API requests and ensure the data processing pipeline's latency remains minimal to support real-time analytics.The criterion for successful load balancing is the uniform distribution of load across all available resources, maintaining the defined performance thresholds.

**Load Balancing Action:**
ALB is strategically placed to distribute incoming API requests across multiple backend services hosted on the EKS cluster. ALB uses the Least Outstanding Requests routing algorithm, which routes each new request to the target with the fewest ongoing requests. This helps to ensure a fair distribution of load, especially during traffic spikes.

**Feedback Loop:**
The ALB continuously monitors the health and load on each target and adjusts traffic distribution in real-time. The feedback loop is established through health checks and metrics like HealthyHostCount, ensuring high availability and fault tolerance.

**Trade Offs:**

By employing ALB, we avoid overloading a single service instance, which could occur with direct invocation methods. The decision to use ALB comes with increased complexity in setup and cost but provides better performance under variable loads. To prevent the ALB from becoming a bottleneck, especially for a highly utilized system like a smart city platform, we would consider deploying multiple ALBs in different Availability Zones. This would ensure even higher availability and fault tolerance.

## 5.5. Design principles and best practices used

**Scalability (Principle 1: Think Adaptive and Elastic)**
The Smart City Platform leverages services like Kinesis and Lambda that

automatically scale with the volume of data and number of requests, respectively. EKS is used to manage and scale containerized applications, with ALB distributing the load evenly across the deployed containers.

**Disposable Resources (Principle 2: Treat servers as disposable resources)**
By employing Lambda and EKS, the architecture takes advantage of AWS's capability to create and dispose of resources on-demand. This means that resources are allocated when needed and released when they're not, optimizing for both performance and cost.

**Automation (Principle 3: Automate Automate Automate)**
Automation is a key theme in the architecture, with services like Lambda, which automatically runs code in response to triggers, and the orchestration capabilities of EKS. AWS CloudFormation or similar services could be used to automate infrastructure provisioning and deployment processes.

**Loose Coupling (Principle 4: Implement loose coupling)**
The architecture uses API Gateway to decouple the front-end from the back-end services, allowing each to evolve independently. This is further enhanced by using message queues (not shown but commonly implemented) to decouple inter-service communications.

**Services not Servers (Principle 5: Focus on services, not servers)**
The platform focuses on leveraging fully managed services like Aurora and QuickSight, which abstract the underlying servers away from the architect, focusing instead on the capabilities and data flows.

**Database as a Foundation (Principle 6: Database is the base of it all)**
Aurora is chosen for its performance and reliability as a managed database solution, acting as the foundational data store for transactional data, which is critical for the platform's operations.

**Elimination of Single Points of Failure (Principle 7: Be sure to remove single points of failure)**
The architecture ensures high availability by deploying services across multiple Availability Zones and using Amazon Aurora for its fault-tolerant database cluster.

**Cost Optimization (Principle 8: Optimize for cost)**
The architecture includes the use of AWS Budgets and Cost Explorer to monitor and optimize costs, ensuring the platform remains within budget while still meeting performance requirements.

**Security (Principle 10: AWS Cloud Architecture Security)**

Security is woven throughout the architecture using IAM for fine-grained access control, VPC for network isolation, and encryption measures for data at rest and in transit, following AWS's security best practices.

## 5.6. Tradeoffs revisited

**1. Data Ingestion and Processing: Amazon Kinesis and AWS Lambda**
In the domain of data ingestion and processing, we chose Amazon Kinesis and AWS Lambda, recognizing a critical trade-off between the performance efficiency of real-time data processing and cost optimization. The decision to use Kinesis stems from its high-throughput capabilities, crucial for the vast data streams from city sensors. While alternatives like direct database ingestion could potentially reduce costs, they fall short in handling the volume and velocity of data required by a smart city infrastructure. We accepted the higher cost for the superior performance of Kinesis.

AWS Lambda's event-driven execution model was preferred over services like EC2 or ECS[15]. The trade-off here is operational excellence and scalability against the more predictable cost structure of EC2. Lambda's scaling abilities align with the variable load, which is paramount in a smart city context. We emphasized the operational benefits of a serverless architecture, where the cost premiums of Lambda invocations are justified by the flexibility and reduced management overhead it provides.

Consequences Table:

| **Objectives** | **AWS Lambda (Option A)** | **Amazon EC2 (Option B)** |
|---|---|---|
| **Reliability** | High (auto-scaling) | Medium (manual scaling) |
| **Cost** | Variable based on use | Fixed with reserved instances |
| **Performance Efficiency** | High (event-driven) | Medium (depends on setup) |
| **Operational Complexity** | Low (managed service) | High (requires management) |
| **Security** | High (built-in AWS features) | High (customizable but complex) |
| **Scalability** | Instant | Gradual |

**2. Data Storage: Amazon S3 and Amazon Aurora**

When it comes to data storage, Amazon S3 was chosen as a data lake solution over alternatives like EBS or EFS. This selection reflects a trade-off between cost optimization and operational excellence. While EBS or EFS could potentially lower the cost for certain data storage patterns, S3 provides unmatched scalability and durability, which are non-negotiable for the long-term storage needs of a smart city platform. Here, the superior operational capabilities of S3 – including its integration with various analytics services – outweigh the potential cost savings from other storage services.

For relational data, Amazon Aurora was selected over Amazon RDS. The balance tipped towards reliability and performance efficiency. Aurora's performance and seamless failover mechanisms offer a robust foundation critical for the smart city's transactional operations. This decision accepts the potentially higher costs associated with Aurora for its reliability advantages, aligning with our prioritization of uninterrupted service and data integrity.

Consequences Table:

| **Objectives** | **Amazon S3 (Option A)** | **Amazon EFS (Option B)** |
|---|---|---|
| **Cost** | Lower (pay per use) | Higher (fixed cost) |
| **Scalability** | Highly scalable | Limited scalability |
| **Performance** | Optimized for large files | Optimized for smaller files |
| **Data Access Flexibility** | High (supports various data formats) | Lower (file-system based) |
| **Durability** | Extremely high | High |
| **Integration with Analytics Tools** | High (native integration with AWS analytics tools) | Lower |

**3. Backend Services: Amazon EKS**

In the backend services, Amazon EKS was favored over ECS due to its community support and alignment with cloud-native practices, which resonates with the principle of operational excellence. The trade-off made here is between the higher operational complexity of managing Kubernetes against the simplicity of ECS. The decision was driven by the need for a flexible and modern orchestration layer that can adapt to the evolving needs of a smart city infrastructure, even if it demands a steeper learning curve and potentially higher operational costs.

Consequences Table:

| **Objectives** | **Amazon EKS (Option A)** | **Amazon ECS (Option B)** |
|---|---|---|
| **Community Support** | Strong | Moderate |
| **Flexibility** | High (supports various orchestrators) | Lower (AWS specific) |
| **Operational Complexity** | Higher | Lower |
| **Integration with AWS Services** | Good | Excellent |
| **Cost** | Higher (due to added features) | Lower |
| **Scalability** | High | High |

**4. API Management and Integration: Amazon API Gateway**

For API management, Amazon API Gateway was chosen to act as the controlled entry point to the backend services. This decision involves a trade-off between security and cost optimization. API Gateway comes at a higher price point compared to direct service integration but provides valuable features such as throttling, caching, and user authentication. These features contribute to a more secure and managed interaction with the city's data services, which takes precedence over the cost-saving potential of a less secure, direct integration pattern.

Consequences Table:

| **Objectives** | **Amazon API Gateway (Option A)** | **Direct Communication (Option B)** |
|---|---|---|

| | | |
|---|---|---|
| **Security** | High (built-in features) | Variable (depends on implementation) |
| **Ease of Management** | High (centralized control) | Lower (requires manual setup) |
| **Flexibility** | Moderate | High |
| **Cost** | Higher (managed service) | Lower (no additional cost) |
| **Performance** | Optimized for API management | Potentially higher (direct links) |
| **Scalability** | High | Variable |

**5. Data Analytics and Visualization: Amazon QuickSight**

Amazon QuickSight was selected for data analytics and visualization over other tools like Amazon Elasticsearch Service due to its managed service offering. This is a trade-off between performance efficiency and cost optimization. QuickSight provides a user-friendly interface and is fully managed, which means less time spent on setup and maintenance – a clear win for operational efficiency. While it might come at a higher operational cost compared to Elasticsearch, the ease of creating dashboards and generating insights for city officials made QuickSight the preferred choice.

Consequences Table:

| **Objectives** | **Amazon QuickSight (Option A)** | **Alternative BI Tools (Option B)** |
|---|---|---|
| **Integration with AWS Services** | Excellent | Variable |
| **User Interface and Ease of Use** | Intuitive and Simple | Complex (Varies by tool) |
| **Cost** | Predictable and often lower | Potentially higher |
| **Customization** | Standard | High |
| **Scalability** | Good with AWS ecosystem | Depends on tool |
| **Data Processing Capabilities** | Optimized for AWS data sources | Broad but requires configuration |

**6. Security: AWS IAM**

AWS IAM is central to managing access and enforces the principle of least privilege across the platform. This decision prioritizes security over operational simplicity. IAM's fine-grained access control introduces additional complexity in policy management. However, given the sensitivity of a smart city's data and operations, this complexity is a warranted trade-off to ensure that stringent security measures are in place.

Consequences Table:

| **Objectives** | **AWS IAM (Option A)** | **Manual Security Management (Option B)** |
|---|---|---|
| **Cost** | Lower (part of AWS services) | Higher (requires additional resources) |
| **Complexity** | Lower (integrated with AWS) | Higher (manual setup and maintenance) |
| **Customization** | Moderate (within AWS constraints) | High (fully customizable) |
| **Scalability** | High (automated scaling) | Lower (manual scaling) |
| **Security Robustness** | High (AWS standards) | Variable (depends on implementation) |
| **Ease of Implementation** | High (user-friendly) | Lower (requires expertise) |

**7. Network Configuration and Isolation: AWS VPC**

AWS VPC is used to create a secure and isolated network environment, prioritizing security and reliability over the simplicity of a default network setup. The trade-off accepted here is increased complexity and potential cost for network setup and maintenance in exchange for the assurance of a secure, isolated environment that protects the smart city's operations from external threats and unauthorized access.

Consequences Table:

| Objectives | AWS VPC (Option A) | Default Network Setup (Option B) |
|---|---|---|
| **Security** | High (customizable) | Standard |
| **Complexity of Setup** | Higher (requires configuration) | Low |
| **Network Control** | Extensive | Limited |
| **Integration with AWS Services** | Seamless | Basic |
| **Cost** | Variable (based on usage) | Lower (standard features) |

**8. Load Balancing Strategy: Application Load Balancer (ALB)**
The decision to implement ALB for load balancing involves a nuanced trade-off between performance efficiency and cost optimization. ALB offers advanced routing and SSL termination, which are critical for handling the incoming traffic to the EKS cluster. While this choice might come at a higher cost compared to a simpler solution like Network Load Balancer (NLB), the advanced features of ALB, such as content-based routing and support for microservices, justify the additional expense.

In conclusion, the trade-offs made in designing the smart city platform reflect a series of complex, interdependent decisions. By applying the "Even Swaps" method, we ensured that each trade-off was made with a deliberate, rational approach, considering the overarching goals and requirements of the project.

## 5.7. Discussion of an alternate design

SKIPPED

# 6. Kubernetes Experimentation

## 6.1. Experiment Design

In this experimental setup we are evaluating the autoscaling functionality of horizontal pods within a MiniKube and Docker environment. Our objective is to examine the behaviors of pod scalings in response to varying loads and fulfill the requirements TR 16.1 and TR 16.2.
During the experiment when the system experiences an increase in total load surpassing a predetermined threshold, the number of pods dynamically scales up. Conversely, upon halting the load generator and reducing the overall load, the system automatically scales down the number of pods. This evaluation aims to assess the efficiency and responsiveness of horizontal pod scaling in adapting to fluctuating workloads within the Minikube and Docker environment[16]. The findings contribute valuable information into the effectiveness of auto scaling mechanisms in managing resource allocation based on real-time demand.

## 6.2 Workload Generation

This experiment begins by deploying a straightforward application on MiniKube using Docker containers. Subsequently, we employ Locust, a load testing tool, to gauge how Kubernetes' Horizontal Pod Autoscaler (HPA) dynamically adapts the pod count in response to workload variations. The focus lies in comprehending how the HPA efficiently scales the number of pods within the Kubernetes cluster based on observed workload changes. This approach aims to evaluate the system's responsiveness and scalability under varying operational demands, shedding light on Kubernetes' resource management capabilities in handling fluctuating workloads.

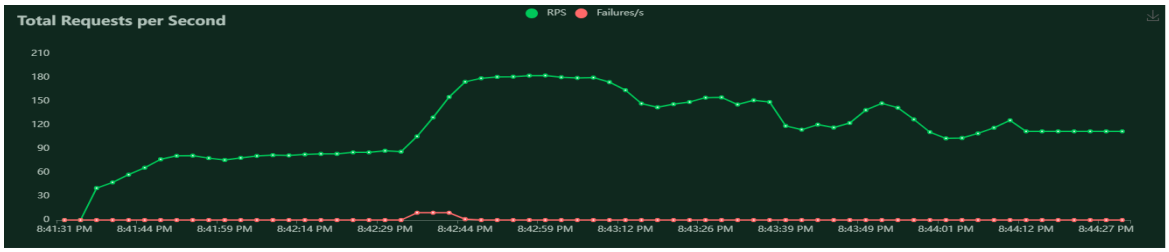
```
C:\Users\Viraj\OneDrive\Desktop\Cloud Project 2>minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

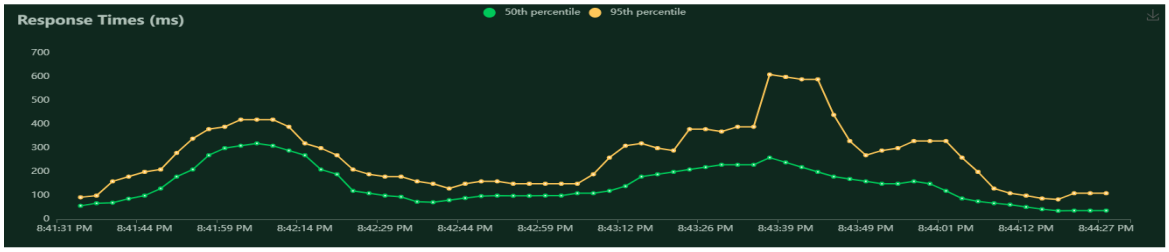
```
C:\Users\Viraj\OneDrive\Desktop\Cloud Project 2>kubectl apply -f deployment.yaml
deployment.apps/server-app unchanged
service/server-app created

C:\Users\Viraj\OneDrive\Desktop\Cloud Project 2>kubectl get deploy
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
server-app   1/1     1            1           94s

C:\Users\Viraj\OneDrive\Desktop\Cloud Project 2>kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
server-app-78bf5d76b4-gdl58   1/1     Running   0          103s

C:\Users\Viraj\OneDrive\Desktop\Cloud Project 2>kubectl apply -f hpa.yaml
horizontalpodautoscaler.autoscaling/cpu-autoscale created
```

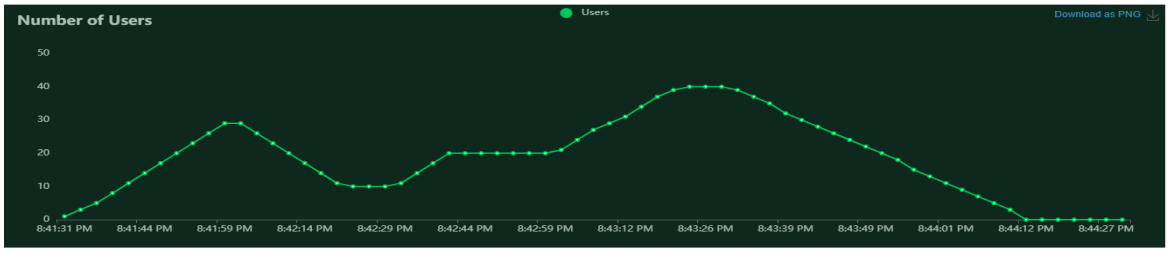
Managing a Kubernetes cluster via Minikube involves key steps to ensure effective resource orchestration. The 'minikube status' command provides a snapshot of the cluster's current status. Utilizing 'kubectl apply -f file,' Kubernetes resources are created or modified as per the specifications in the 'deployment.yaml' file, initiating the deployment process based on the file's directives. Information retrieval within the cluster is facilitated by 'kubectl get pods' and 'kubectl get deploy' commands, allowing detailed insights into individual running pods and deployment statuses, aiding in efficient monitoring and management.

Furthermore, scalability is enhanced using Horizontal Pod Autoscaler (HPA) functionality within Kubernetes. The HPA dynamically adjusts pod numbers based on CPU utilization, maintaining optimal performance. Docker containerization enables the deployment of the Flask application within Kubernetes, managed by Minikube. Load testing via Locust assesses the application's performance under varying workloads, validating its stability and scalability[17].
This orchestrated setup, spanning containerization, deployment, load testing, and adaptive scaling, showcases Kubernetes' robust resource management capabilities for optimal application performance and resource utilization.

Load Generation is shown on Locust




The provided graphs depict a series of load tests performed to gauge the application's performance under varying workloads. These tests serve to evaluate the scalability of the pods within the Kubernetes cluster.

By subjecting the system to diverse load levels, the aim is to assess how effectively the infrastructure can adapt and scale to meet changing demands.

```
C:\Users\Viraj\OneDrive\Desktop\Cloud Project 2>kubectl get hpa cpu-autoscale --watch
NAME            REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
cpu-autoscale   Deployment/server-app    0%/60%    1         10        1          23m
cpu-autoscale   Deployment/server-app    129%/60%   1         10        1          25m
cpu-autoscale   Deployment/server-app    129%/60%   1         10        3          25m
cpu-autoscale   Deployment/server-app    162%/60%   1         10        3          26m
cpu-autoscale   Deployment/server-app    127%/60%   1         10        3          27m
cpu-autoscale   Deployment/server-app    127%/60%   1         10        6          27m
cpu-autoscale   Deployment/server-app    127%/60%   1         10        7          27m
cpu-autoscale   Deployment/server-app    0%/60%     1         10        7          28m
cpu-autoscale   Deployment/server-app    0%/60%     1         10        7          33m
cpu-autoscale   Deployment/server-app    0%/60%     1         10        1          33m
```

These tests effectively scrutinize the system's ability to handle fluctuations in traffic and workload intensity. They shed light on the Kubernetes cluster's responsiveness in dynamically adjusting the number of pods through Horizontal Pod Autoscaler (HPA) mechanisms. These findings are pivotal in understanding the system's resilience and its capability to efficiently manage resources, ensuring optimal performance and stability across different usage scenarios.

The table shows the details of Horizontal Pod Autoscaler's behavior regarding CPU utilization fluctuations within the cluster. Notably, when CPU usage breaches the 60%

threshold (e.g., instances like 129%/60% or 162%/60%), the HPA promptly increases pod numbers to accommodate heightened loads, seen in the rise from 1 to 3, 6, and 7 replicas respectively. Conversely, with decreasing loads (e.g., from 127%/60% to 0%/60%), the HPA adeptly scales down pods, gradually reducing replicas back to 1. These dynamic adjustments underscore the HPA's agility in dynamically managing resources, ensuring optimal performance and stability amidst varying workload intensities.

## 6.3 Analysis of the results

The conducted experiment notably satisfies the **TR16.1** and **TR16.2** criteria by employing scalable cloud resources and demonstrating the effectiveness of autoscaling solutions in managing diverse workloads. Specifically, the successful demonstration of Kubernetes' Horizontal Pod Autoscaler showcases its ability to efficiently manage workload variations, ensuring optimal resource utilization and system stability. This aligns precisely with the prerequisites of a cloud-based smart city, emphasizing scalability, reliability, and performance for delivering seamless and responsive services to citizens.

# 7. Ansible playbooks

**SKIPPED**

# 8. Demonstration

**SKIPPED**

# 9. Comparisons

**SKIPPED**

# 10. Conclusion

## 10.1. The lessons learned

The journey through this project has been an enriching experience that has challenged conventional engineering approaches and instilled a deeper understanding of the architectural process. Here are the detailed reflections on the lessons learned:

**Understanding the Depth of Requirements Analysis**

The process began with an in-depth analysis of business requirements (BRs), a step often underestimated in its importance. Moving beyond the urge to dive into the solution, we learned the value of taking a step back to thoroughly understand what is truly needed. The challenge was not only in identifying the BRs but also in translating them into precise technical requirements (TRs). This exercise sharpened our analytical skills and emphasized the importance of alignment between business objectives and technical capabilities.

**Navigating Trade-Offs**

One of the most significant lessons was learning to navigate trade-offs. Each TR comes with its own set of benefits and drawbacks, and prioritizing them was a complex task. We learned to evaluate the impact of each decision, not just in isolation but how it shapes the overall architecture. This involved a continuous balance between reliability, performance, cost, and other critical factors.

**Selection of Services and Prioritization**

With a plethora of AWS services at our disposal, selecting the most suitable ones for our TRs required meticulous research and understanding of each service's offerings. It was interesting to match the capabilities of services like Amazon Kinesis, AWS Lambda, and Amazon Aurora to our specific needs. The process was like fitting pieces into a puzzle, where each service had to contribute to the coherent picture of our architecture.

**Iterative Design and Feedback Loops**

The project embraced an iterative design approach, using feedback loops to refine our solutions continuously. This iterative process was not only applied to the software development cycle but also to the architectural design phase. The feedback mechanism allowed us to hone in on the most efficient and effective

design, making what initially seemed like a daunting task much more manageable.

**Balancing Technical and Business Perspectives**

Engineers often gravitate towards technical solutions, but this project required us to think from a business perspective first. We learned to ask not just how we can build something, but why we should build it in a certain way. This mindset shift was both difficult and intriguing, as it opened up a new way of thinking about problem-solving.

**The Simplicity on the Far Side of Complexity**

Unexpectedly, after navigating through the complex web of BRs, TRs, and the myriad of AWS services, the final design of the solution turned out to be fairly straightforward. Once the top priorities were clear, and we made informed trade-offs, the pieces fell into place more easily than anticipated. It underscored the principle that simplicity often lies on the far side of complexity.

**A Practical Approach to Solution Architecture**

The project was far from boring; it provided a practical approach to solution architecture, teaching us valuable industry practices. It was fascinating to see how theoretical knowledge translates into real-world applications and how a structured approach to design can yield robust and scalable architectures.

## 10.2. Possible continuation of the project

**SKIPPED**

# 11. References


[1] Amazon Web Services. (n.d.). *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] Microsoft Azure Services. Azure Cloud Products, Microsoft. https://azure.microsoft.com/en-us/products/

[3] Google Cloud Platforms Services. Google Cloud Platforms Products, Google.https://cloud.google.com/products

[4] Amazon Kinesis. AWS.https://aws.amazon.com/kinesis/?did=ap_card&trk=ap_card

[5] Amazon Aurora. AWS. https://aws.amazon.com/rds/aurora/?did=ap_card&trk=ap_card

[6] Amazon Quicksight. AWS. https://aws.amazon.com/quicksight/?did=ap_card&trk=ap_card

[7] Amazon SImple Queue Service. AWS.https://aws.amazon.com/sqs/?did=ap_card&trk=ap_card

[8] Secure multi-tenant data ingestion pipelines with Amazon Kinesis Data Streams and Kinesis Data Analytics for Apache Flink. AWS. https://aws.amazon.com/blogs/big-data/secure-multi-tenant-data-ingestion-pipelines-with-amazon-kinesis-data-streams-and-kinesis-data-analytics-for-apache-flink/

[9] Event–Driven Serverless Architecture Using AWS Lambda. AWS. https://cuelogictech.medium.com/event-driven-serverless-architecture-using-aws-lambda-6aef8d52ba80#:~:text=Event-driven%20implies%20that%20a%20Lambda%20function%20is%20triggered,and%20they%20have%20the%20responsibility%20to%20process%20it.

[10] City planning analogy with cloud architecture
https://aws.amazon.com/blogs/architecture/use-a-city-planning-analogy-to-visualize-and-create-your-cloud-architecture/

[11] Lukasz Wieclaw, Volodymyr Pasichnyk, Natalija Kunanets, Oleksij Duda, Oleksandr Matsiuk,Pawel Falat - Cloud computing technologies in “smart city” projects
https://ieeexplore.ieee.org/document/8095101/authors

[12] S3 with Lambda https://docs.aws.amazon.com/lambda/latest/dg/with-s3.html

[13] Central storage: Amazon S3 as the data lake storage platform. AWS.
https://docs.aws.amazon.com/whitepapers/latest/building-data-lakes/amazon-s3-data-lake-storage-platform.html

[14] AWS SQS AND DECOUPLED ARCHITECTURES.
https://www.deviq.io/insights/aws-sqs-decoupled-architecture

[15] AWS Lambda with Kubernetes
https://docs.aws.amazon.com/lambda/latest/dg/with-kubernetes.html

[16] Kubernetes
https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/

[17] Locust.io experiments — running in Kubernetes
https://medium.com/locust-io-experiments/locust-io-experiments-running-in-kubernetes-95447571a550

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

[19] OpenAI and ChatGPT