MODELING A GLOBALLY SCALABLE AND COMPLIANT MULTI-TENANT SAAS ARCHITECTURE

Mohamed Jaward Uzama Zaid

239379U

MSc in Computer Science

Department of Computer Science and Engineering
Faculty of Engineering

University of Moratuwa
Sri Lanka

June 2025

MODELING A GLOBALLY SCALABLE AND COMPLIANT MULTI-TENANT SAAS ARCHITECTURE

Mohamed Jaward Uzama Zaid

239379U

Thesis submitted in partial fulfilment of the requirements for the degree
MSc in Computer Science

Department of Computer Science and Engineering
Faculty of Engineering

University of Moratuwa
Sri Lanka

June 2025

DECLARATION

I declare that this is my own work and this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other University or Institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: Date: 06/06/2025

The above candidate has carried out research for the Masters thesis under my supervision. I confirm that the declaration made above by the student is true and correct.

Name of Supervisor: Dr. Buddhika Karunarathne

Signature of the Supervisor: Date: 22/06/2025

ACKNOWLEDGMENT

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr. Buddika Karunaratne, for his valuable guidance, patience, and support throughout this research project. His expertise and constructive feedback were instrumental in shaping this work. I am equally grateful to Dr. Sapumal Ahangama for his insightful comments and suggestions during my PG Diploma viva examination. I would also like to extend my sincere thanks to my final viva examiners, Dr. Thanuja Ambegoda and Dr. Thilinat Thanthriwatta, for their valuable feedback on the final thesis.

My sincere appreciation goes to all the lecturers and administrative staff members of the Department of Computer Science and Engineering, University of Moratuwa, who contributed to organizing the MSc program and facilitating this research journey. Your dedication and support created an environment conducive to academic growth and research excellence. I would especially like to acknowledge Mrs. Nisansala Samarasooriya for supporting administrative works.

To my beloved wife, words cannot adequately express my appreciation for your unwavering support during both the challenging and rewarding moments of this research journey. You have been my pillar of strength throughout this endeavor. I am also profoundly grateful to all my family members for their constant encouragement and understanding during this intensive period of study.

Finally, I would like to acknowledge my friends and peer colleagues who provided moral support and stimulating discussions throughout this academic pursuit. Your companionship made this journey more meaningful and enjoyable.

ABSTRACT

The rapid growth of cloud computing has established Software as a Service (SaaS) as the dominant software delivery model. The increasing adoption of SaaS creates opportunities for SaaS providers to expand globally and capture emerging markets. However, global expansion introduces significant challenges in balancing global scalability, data residency compliance, and operational efficiency. Existing research often addresses these requirements in isolation, resulting in suboptimal solutions.

This research proposes and validates a novel SaaS architecture that reconciles these competing requirements. The architecture comprises two key components: (1) a logically centralized yet physically distributed Control Plane, which enables operational efficiency through unified management of Application Planes and tenants, and (2) regionally distributed Application Planes, supporting global scalability while complying with regional data residency regulations.

The proposed solution (fully distributed nCPxnAP model) is evaluated against two alternative deployment models (single region 1CPx1AP and hybrid 1CPxnAP). Through comprehensive comparative analysis of three architectural variants, we demonstrate that our fully distributed nCPxnAP model achieves optimal performance, ensures full compliance with data residency laws, and preserves operational efficiency. The key contributions of this research are: (1) an architectural model for a globally scalable, compliant, and operationally efficient multi-tenant SaaS; (2) empirical validation via real world implementation and testing; and (3) practical guidelines for SaaS providers at various growth stages.

The findings offer a clear pathway for SaaS providers transitioning from single region deployments to global distribution while maintaining data residency compliance and operational efficiency. The research establishes new design principles for compliant global SaaS deployments and provides actionable implementation blueprints and performance benchmarks. It not only advances academic research but also provides industry professionals with a validated architectural model for navigating the complexities of modern global SaaS delivery.

Keywords: Cloud Computing, SaaS, Global Scalability, Data Residency Compliance, Operational Efficiency

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Description
AP	Application Plane
API	Application Programming Interface
AWS	Amazon Web Services
CD	Continuous Deployment
CI	Continuous Integration
CP	Control Plane
DB	Database
DNS	Domain Name System
ECR	Elastic Container Registry
ERD	Entity Relationship Diagram
IaC	Infrastructure as Code
ID	Identification
IT	Information Technology
MTA	Multi Tenant Architecture
SaaS	Software as a Service
SLA	Service Level Agreement
S3	Simple Storage Service
TPMS	Third Party Management Systems
UI	User Interface
URL	Uniform Resource Locator

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

The rapid adoption of cloud computing, particularly the Software as a Service (SaaS) delivery model, has transformed how businesses operate and access software. SaaS provides significant advantages, including scalability, cost effectiveness, and accessibility, making it a strong choice for software providers across various industries. With this growing adoption, SaaS providers have the opportunity to expand their market to multiple regions and even globally. That being so, SaaS providers need to rethink their SaaS application deployments. Deploying SaaS applications globally introduces more challenges, particularly in balancing global scalability with increasingly stringent regional data compliance regulations, while maintaining operational efficiency. This research addresses these challenges by modeling an architecture for globally scalable and compliant SaaS applications.

A key problem in global SaaS deployments is reconciling the need for global scalability with the complex landscape of regional data residency regulations. Data residency laws mandate that specific data types [1] be stored and processed within defined geographical boundaries [2]. These regulations, designed to protect citizens' data, vary significantly across different countries and regions, creating substantial challenges for SaaS providers seeking international reach. Non compliance can lead to substantial fines, legal actions, and reputational damage [3]. Existing approaches often struggle to balance scalability and compliance effectively. Some prioritize performance related scalability without addressing global business scalability, while others focus on general compliance without tailoring solutions to SaaS or providing specific architectural recommendations for SaaS providers. This creates a need for architectural patterns that holistically address these competing requirements.

Operational efficiency is another common issue in SaaS. As described in the AWS SaaS whitepaper [4], operational efficiency is crucial for the success of the SaaS business model. Therefore, SaaS providers must prioritize operational efficiency when developing their solutions. Failure to do so can hinder the growth and scalability of a SaaS application. Providing a unified experience to manage and operate tenant environments is key to achieving operational efficiency [5]. This can be achieved through a two layer architecture, consisting of an Application Plane and a Control Plane [6]. The Application Plane hosts the SaaS solution, while the Control Plane orchestrates the Application Plane. This two layer architecture serves as the foundation for our research to achieve operational efficiency.

Our proposed architecture consists of a centralized Control Plane and regionally distributed Application Planes. The centralized Control Plane orchestrates Application Planes and manages tenants, including compliance driven tenant scheduling and routing. Meanwhile, the distributed Application Planes, deployed regionally, store and process tenant data within their designated regions. This design

minimizes operational overhead while ensuring adherence to regional data compliance requirements and enabling a globally scalable and efficient SaaS architecture.

In the following subsections, we will explore key concepts related to SaaS, multi-tenancy, and data residency compliance. The report will then present the research problem, objectives, and a review of related work, followed by the proposed solution and implementation details. Finally, we will provide an in depth analysis of the experimental results, research findings, and recommendations.

1.1 What is SaaS?

The term SaaS refers to a business and delivery model. The problem is that not everyone understands what it means to be SaaS. While there is some agreement on some of the core values of SaaS, there is still significant confusion regarding what it means to be SaaS. It's obvious that the team's views on SaaS might vary. At the same time, a lack of clarity around SaaS principles and concepts may lead to some confusion for those considering a SaaS method of delivery.

As mentioned in [4] SaaS Architecture Fundamentals, defining what it means to be SaaS starts with accepting on a key principle, thus SaaS is a business model. This means that adoption of a SaaS delivery model is primarily motivated by a set of business goals. Though technology will be utilized to achieve some of these goals, SaaS is about establishing a mindset and model that focuses on a specific set of business objectives such as,

- Agility: In order to stay successful, SaaS companies have to adopt new pricing models, market niches, and consumer demands in order to adjust to market, customer, and competitor changes.
- Operational efficiency: SaaS companies target operational efficiency for scale and agility, focusing on regular feature releases and a unified experience for managing customer environments. This strategy eliminates the need of one off versions and customizations, allowing successful growth and scaling.
- Frictionless onboarding: To embrace growth, focus on allowing frictionless tenant onboarding, regardless of customer type. Transformation to a service centric model, focusing on repeatability and efficiency of the tenant onboarding lifecycle, ensuring time to value for all customers.
- Innovation: Transitioning to SaaS involves not only satisfying current customer needs but also establishing foundational elements for innovation. Also use the agility to drive future innovations that allows unlocking new markets, opportunities, and efficiencies for customers.
- Market response: SaaS provides agility to enable businesses to react instantly to changes in the market. Investing in organizational, technical, and cultural

elements enables business strategy to be adjusted in response to changing market and consumer conditions.

- Growth: SaaS is a growth focused business model that aligns moving parts of the organization around agility and efficiency, allowing organizations to target growth models by supporting fast adoption of their SaaS offerings.

Each of the above items are focused on business outcomes. There are many technical best practices and strategies that can be utilized to build a SaaS solution, but they don't change the set of objectives of the business. These objectives are crucial to take into account when defining organizational goals for SaaS adoption.

By considering above objectives and principles, the paper [4] has formalized a definition for SaaS as,

“SaaS is a business and software delivery model that gives organizations the ability to offer their solutions in a low friction, service centric model that maximizes value for customers and providers. It relies on agility and operational efficiency as pillars of a business strategy that promotes growth, reach, and innovation.”

1.2 What is Multi-Tenancy

The notion of multi-tenancy is surrounded by lots of misconceptions. A common misconception is that multi-tenancy only involves resource sharing among multiple customers or tenants. However it's important to make clear that multi-tenancy spans beyond resource sharing. It involves the comprehensive management and operation of every tenant within a SaaS system through a unified interface.

The paper [7] has come up with a new definition or redefines the concept of multi-tenancy to avoid misunderstanding and ensure accurate interpretation in the SaaS landscape. In the context of SaaS, multi-tenancy is a key notion that refers to the ability of a single software instance to serve multiple tenants or customers. Unlike traditional setups where each customer would have a dedicated software instance (with versioning), multi-tenancy optimizes resource utilization by allowing shared access to computing and storage infrastructure. This model goes beyond simple resource sharing to include comprehensive tenant management and operation through a unified interface within the SaaS system. This method not only enhances efficiency but also facilitates scalability, offering a cost effective solution for serving diverse customer needs.

In practical terms, there could be differences in the way resources are isolated or shared in a multi tenant SaaS environment. This could range from shared computing and storage for all tenants to dedicated resources based on different types of tenancy. Overall, the essence of multi-tenancy in the SaaS landscape lies in its ability to collectively manage and operate a whole suite of services, ensuring a seamless and unified experience for all tenants.

1.3 Data Residency Compliance

Data residency laws, which mandate that specific data types [1] be stored and processed within defined geographical boundaries [2], further complicate global SaaS deployment. Enacted to protect citizens' data, these regulations present significant obstacles for SaaS providers seeking international reach. These laws are designed to protect the privacy and security of individuals' data by ensuring that it is subject to the legal jurisdiction of their country or region. The specific requirements of data residency laws vary significantly across different jurisdictions, creating a complex compliance landscape for global SaaS providers.

Non compliance with data residency laws can lead to severe consequences, including significant fines, legal actions, and reputational damage [3]. Therefore, it is essential for SaaS providers to carefully consider data residency requirements when designing and deploying their applications globally. Scaling a SaaS application across multiple regions presents challenges such as resource provisioning, management, and tenant scheduling. Data residency introduces an additional layer of complexity, as tenant data must be stored and processed within the designated region. Ensuring operational efficiency while adhering to data residency regulations often results in increased infrastructure costs, management overhead, and the risk of errors. Effective application and tenant management, including tenant scheduling and compliance driven tenant routing, requires a well defined architectural pattern.

CHAPTER 2

RESEARCH PROBLEM

With the emergence of cloud computing, the software delivery model has evolved, and SaaS has quickly become a popular choice among software vendors. SaaS has changed how businesses access and utilize software, offering significant benefits such as scalability, cost effectiveness, and accessibility. The success of the SaaS delivery model presents providers with the opportunity to expand their market globally. To achieve this, SaaS providers need to deploy multiple components across different regions to address latency and data residency complaints issues.

Deploying SaaS applications globally introduces significant challenges, particularly in balancing global reach and scalability with adherence to regional data residency laws and maintaining operational efficiency. The data residency regulations, designed to protect citizens' data, mandate that specific data types be stored and processed within defined geographical boundaries. For SaaS providers operating internationally, non compliance can lead to severe consequences, including substantial financial penalties, legal actions, and reputational damage. Therefore, SaaS providers must rethink their architectural strategies to comply with these laws.

Most approaches to global SaaS deployment often involve deploying a single instance globally, accessible from anywhere. This approach leads to performance issues as SaaS consumers from different regions experience higher latency when accessing a single instance over the internet and data residency law is another concern. Deploying multiple components across regions can address latency problems, but it introduces new challenges that determine how and where tenant data should be stored to comply with data residency laws. Additionally, managing multiple deployed components and tenants becomes increasingly complex and costly, potentially threatening the success of the SaaS offering.

Existing research often focuses on performance oriented scalability without addressing global business growth. Some studies discuss global scalability for SaaS but lack proper architectural patterns. Others address data residency laws, but their discussions are not accompanied by actionable architectural recommendations. Similarly, most works on operational efficiency focus on building SaaS solutions with efficiency but fail to address the operational challenges of global scalability.

As a result, there are no comprehensive architectural solutions available for globally scalable SaaS that effectively balance regional data residency compliance and operational efficiency. This research aims to bridge this gap by proposing an architectural model. The proposed model seeks to balance these competing requirements, enabling SaaS providers to achieve global reach while meeting data residency requirements and maintaining operational efficiency.

CHAPTER 3

RESEARCH OBJECTIVES

- **Define a Globally Scalable Multi-Tenant SaaS Architecture:** Develop an architectural framework that supports multi tenant SaaS deployment across multiple regions, ensuring global scalability.

- **Ensure Compliance with Data Residency Laws:** Propose and integrate mechanisms to adhere to regional data residency regulations within the architectural design.

- **Preserve Operational Efficiency:** Establish strategies to maintain operational efficiency while minimizing overhead in managing globally distributed Application Planes and tenants.

- **Evaluate the Architecture's Effectiveness:** Assess the proposed architecture within the context of retail management, demonstrating its scalability, compliance adherence, and operational efficiency in real world scenarios.

CHAPTER 4

LITERATURE REVIEW

SaaS has become a dominant software delivery model, providing many advantages for software vendors such as scalability, cost effectiveness, and accessibility, transforming how businesses operate and access software. A fundamental architectural principle of SaaS is multi-tenancy, which allows multiple tenants (customers) to share the same underlying infrastructure to maximize resource utilization and reduce costs. This has boosted its rapid adoption across diverse industries. With this growing adoption of SaaS, SaaS providers need to scale their businesses globally to expand their market reach. While much attention is given to performance scaling, such as horizontal and vertical scaling of application instances to meet demand, the challenges of business scaling are often overlooked. Offering SaaS solutions globally introduces a significant challenge, balancing the need for global reach and scalability with adhering regional data residency regulations and maintaining operational efficiency. These regulations, designed to protect personal data by mandate data storage and processing within defined geographical boundaries. This creates a significant compliance challenge for SaaS providers operating internationally, requiring them to navigate a complex legal landscape. Non compliance can result in penalties, legal action, and reputational damage. Additionally, while finding solutions to scale SaaS applications to multiple regions with residency compliance, it is important to maintain operational efficiency, which is essential for the success of the SaaS model. This research addresses this critical challenge by modeling a globally scalable and compliant multi-tenant SaaS architecture. In this section, we will review the related works in SaaS architecture, multi-tenancy, data residency compliance, and scaling strategies to identify the key research gaps and motivate the need for our proposed solution.

4.1. SaaS and Multi-Tenancy

SaaS is a dominant software delivery model, offering on demand access to applications over the internet and eliminating the need for local installation and maintenance. This model provides many advantages, including cost effectiveness, scalability, and accessibility from any internet connected device. The AWS whitepaper [4], emphasizes that SaaS is not only an architectural pattern but also a set of business objectives focused on delivering software as a service. These objectives include maximizing customer value, minimizing customer cost, and achieving operational efficiency through economies of scale. These business goals are naturally linked to the underlying architecture, specially multi-tenancy which is key to achieving economies of scale and operational efficiency. This change from traditional software ownership to a subscription based model permites businesses to use enterprise grade software without much upfront investments in hardware and infrastructure. This has reduced the barrier to entry for many businesses, particularly

small and medium sized enterprises, and has motivated the rapid adoption of SaaS across various industries. This paper gives valuable guidance on the fundamental principles and considerations for building SaaS applications and setting the groundwork for understanding the architectural choices involved in implementing efficient and scalable SaaS offerings. This guidance is important for understanding the transition to cloud based software delivery and the involvements for businesses of all sizes.

A core architectural principle supporting the efficiency and scalability of SaaS is multi-tenancy. Multi-tenancy allows multiple tenants (customers) to share the same underlying infrastructure, including servers, databases, and application code. This shared model maximizes resource utilization and reduces costs for both the SaaS providers and the customers. However, this sharing introduces complexities in ensuring data isolation and security. The AWS whitepaper [8] discusses different isolation strategies, including data partitioning [9] to address these complexities. Different multi-tenancy models, such as database per tenant, schema per tenant, and shared database, provide varying degrees of isolation, performance, and cost effectiveness. The paper [5] also spotlights the importance of unified experience within the multi-tenant environment to increase operational efficiency. This can be attained by splitting the SaaS into two halves as Control Plane and Application Plane [6]. The Application Plane handles the core application logic and tenant interactions, processing requests and managing tenant data. On the other hand, the Control Plane is responsible for managing tenant provisioning, configuration, monitoring, metering, and other administrative tasks. This separation improves operational efficiency by enabling the shared Application Plane to be managed and operated through a unified interface which reduces maintenance overhead and ensures consistency across all tenants. This separation of concerns is important for achieving scalability, maintainability, and operational efficiency in a multi-tenant SaaS environment.

Other researches give more context and details on multi-tenancy. [10] presents a comprehensive framework for designing and implementing multi-tenant applications. This framework covers a wide range of important architectural decisions and design patterns such as strategies for data partitioning (database per tenant, schema per tenant, shared database), tenant isolation techniques (application level isolation, virtualization), and security considerations (authentication, authorization, access control). The framework also addresses non-functional requirements such as scalability, performance, and resource efficiency which provide valuable guidance for architects in building robust and scalable multi-tenant SaaS applications. [11] performed a comparative analysis of different multitenant architecture (MTA) patterns. Their work gives a detailed comparison of various MTA patterns, such as shared databases with tenant identifiers, separate schemas, and separate databases, evaluating them based on key factors such as performance, scalability, flexibility, security, and cost effectiveness. This comparison provides valuable insights into the trade offs associated with each pattern, helping architects to choose the most relevant

approach based on the specific requirements of their SaaS application. These works give valuable guidance on the principles and practical implementation of multi-tenancy in SaaS environments which provide a strong foundation for our research.

4.2. SaaS and Architectural Patterns

The paper [12] presents a valuable performance comparison of monolithic and microservice architectures in a multi-tenant SaaS environment. The study examines key performance factors such as response time, throughput, and resource utilization under varying tenant loads and customization levels. Their findings provide insights into the performance properties of each approach which helps architects make informed decisions based on desired application requirements and expected tenant behavior. This evaluation gives an understanding of how performance related scalability is achieved in each architecture and how they perform in a multi-tenant environment. However, while this study gives valuable insights into performance related scalability, it primarily focuses on scaling within a single deployment region. It does not address the complexities of business scalability to a global audience or the architectural suggestions of data residency regulations in different geographical locations. Our research addresses this gap by considering the additional challenges posed by data residency compliance in globally distributed SaaS deployments.

Beyond performance, reliability and efficient resource management are important non-functional requirements for multi-tenant SaaS applications. [13] address the challenge of ensuring high availability and fault tolerance. They present a method for creating and dynamically managing application instances based on application load and availability zones. Their approach focuses on minimizing reliability risks by considering factors such as server energy consumption, capacity, and network capacity. By dynamically adjusting the number and placement of application instances, they target to optimize resource utilization while maintaining high availability. This method suggests deploying multiple instances to enhance reliability by distributing the load and providing redundancy. However, while this work gives valuable insights into improving reliability and resource efficiency through multiple instance deployments, it does not address the implications of global scale or the complexities posed by data residency regulations.

Other architectural considerations also play a significant role in SaaS deployments. [14] give an empirical analysis of key factors that influence SaaS architecture dimensions such as scalability, performance, and security. Their work identifies and examines factors that impact these dimensions which provide valuable insights for architects in making informed architectural choices. While this work addresses important aspects of general architectural considerations in multi-tenant SaaS and gives a broad overview of factors influencing SaaS architecture, it does not explicitly address the complexities of global scalability and data residency compliance.

4.3. SaaS and Scalability

SaaS enables significant business scalability and global reach. A recent research [15] has explicitly discussed this connection, indicating how the SaaS model enables rapid entry into new markets and supports the scaling of business operations. This study highlights the role of SaaS in eliminating traditional barriers to business growth, such as high upfront IT infrastructure costs and complex operational overhead. By using subscription based models and cloud infrastructure, SaaS allows businesses of all sizes, from small to enterprises, to scale their resources and operations more efficiently. The study also emphasizes the facilitation of global reach through SaaS which enables businesses to serve customers in different regions without the logistical challenges of managing local data centers. The paper also addresses operational efficiency enhancement through automation and analytics. However, while this paper addresses the importance of global reach and the role of SaaS in facilitating it, it does not dive into the specific architectural challenges of adhering to regional data residency regulations. This is an important gap that our research tries to address, by focusing on architectural patterns and strategies that allow global reach while ensuring compliance.

Effective infrastructure management is crucial for achieving scalability and agility in SaaS deployments, particularly in multi cloud environments. [16] discuss the use of infrastructure as code (IaC) on multi cloud environments for deploying enterprise SaaS applications. They explore how IaC enables automation of infrastructure provisioning, configuration, and management, allowing improved scalability, agility, and cost efficiency. By using IaC, SaaS providers can rapidly and simply deploy and scale their applications across multiple cloud environments, adapting to changing demand and ensuring business growth. This study emphasizes the importance of multi cloud deployment for SaaS providers. Building upon this, [17] further examines the use of new generation IaC tools and techniques for managing enterprise SaaS workloads on multi cloud environments. They discuss how these advanced IaC techniques can be used to automate complex deployment options, including hybrid cloud deployments and disaster recovery. Their work highlights the importance of using declarative IaC to ensure consistency and repeatability in infrastructure provisioning. Both these papers give a strong foundation for understanding how IaC can automate infrastructure management and enable scalability in multi cloud SaaS environments. Although these studies do not directly consider the complexities of global scalability and data residency regulations, they give valuable insights into the use of IaC for managing complex deployments. These insights mainly focus on automation, consistency, and multi cloud management, and will be a valuable reference for using IaC to deploy our proposed solution.

4.4. SaaS and Data Compliance

A serious challenge for SaaS providers serving globally is navigating the complex landscape of data privacy and residency regulations. These regulations, designed to

protect personal data by instruct where data must be stored and processed, creating significant compliance difficulties for global SaaS deployments. [18] describe the comprehensive overview of the concept of data sovereignty, which states that data is subject to the laws and regulations of the country in which it is located. This concept is fundamental to understanding data residency regulations and their involvements for cloud computing. [19] further describes the challenges introduced by data residency regulations to cloud deployments, emphasizing the possible conflicts between the global nature of cloud services and the geographically specific requirements of data regulation laws. These works give an important legal and conceptual foundation for understanding the complexities of data compliance in a global deployments context. They highlight the need for SaaS providers to carefully address the legal implications of their architectural choices and deployment strategies.

The practical use of data privacy laws varies significantly across different regions which require SaaS providers to adopt region specific compliance strategies. The paper [20] highlights the U.S. regulatory landscape, particularly investigating the role of Third Party Management Systems (TPMS) in navigating compliance. It discusses how TPMS can support SaaS providers to manage data privacy and security risks introduced by using third party services, which is specifically relevant in the context of data residency, where data is processed or stored by different entities in different locations. The paper [21] gives insights into the specific data protection regulations in China, which are known for their strict regulations regarding data localization and cross border data transfer. Knowing these regional variations is important for SaaS providers willing to operate in these markets. From these papers, we can understand that when it comes to scaling SaaS globally, region related data regulations are a significant challenge, and we need to seriously address these challenges in our architectural design.

4.5. SaaS and Operational Efficiency

[22] examines how pricing models influence the development and operation of SaaS applications, emphasising the important operational efficiency. They state that pricing is not just a revenue generation method but a key factor that impacts many aspects of SaaS operations. The paper explores how different pricing strategies (subscription based, usage based, tiered pricing) influence decisions related to feature prioritization, resource allocation, and deployment strategies. For example, usage based pricing models motivates efficient resource utilization and can lead to architectural decisions that optimize resource usage. They also explore how pricing can influence the use of automation and other operational efficiency measures. [23] further discuss the relationship between pricing and operational aspects in SaaS operation, strengthening the link between pricing and operational efficiency. They investigate how pricing models impact operational decisions related to capacity planning, resource provisioning, and service level agreements (SLAs). The paper

highlights the need for a comprehensive approach that addresses both pricing and operational costs to make sure profitability and customer satisfaction. They emphasize how efficient operational management, including automation, monitoring, and efficient resource allocation, is important for supporting different pricing models and achieving business goals. Both papers highlight that operational efficiency is not just a technical concern but a business necessity that is directly impacted by pricing choices. However, while these papers provide valuable guidance into the relationship between pricing and operational efficiency, they do not explicitly address the added complexities of achieving global scalability while adhering to data residency regulations and maintaining operational efficiency.

4.6. Research Gap

While existing research heavily explores individual aspects of SaaS, such as multi-tenancy, architectural patterns, scalability strategies, data compliance regulations, and operational efficiency, a critical gap remains in addressing their interrelation within globally distributed SaaS deployments. Studies on business scalability often ignore the challenges introduced by geographically diverse data residency laws. In the same way, research on reliability, resource management (including IaC and multi cloud), and data compliance lacks specific architectural guidance for multi-tenant SaaS solutions that effectively balances compliance, scalability and operational efficiency in a global context. Moreover, studies on operational efficiency often overlook the additional overhead and complexities posed by data residency laws in a globally distributed environment. The intersection of global business scalability, compliance with diverse data residency regulations, and the maintenance of operational efficiency within a multi-tenant SaaS architecture presents a significant, yet largely unexplored, research area. This highlights the need for a novel architectural model to address these interconnected challenges.

CHAPTER 5

PROPOSED METHOD

5.1 Proposed Architecture

Building upon the challenges identified in the previous sections, this section presents an innovative architecture to address the challenges. Our literature review shows a significant gap in effectively balancing global scalability with regional data residency regulations while maintaining operational efficiency. To identify this gap, we propose a novel architectural solution designed to, (1) enable global expansion, (2) ensure compliance with data residency regulations, and (3) maintain operational efficiency. The following discussion details the architecture's core components and their specific roles in addressing these requirements. Finally, we demonstrate the solution's effectiveness through a mock implementation using a retail management case study.

From the definition of SaaS and multi-tenancy defined in [4, 7], SaaS solutions provide a unified experience where all tenants share the same application instance, enabling collective management and operations. As described in the AWS whitepaper [6], the SaaS environment can split into two distinct planes such as the Control Plane and the Application Plane (Fig. 5.1). The Control Plane orchestrates the Application Plane components and tenant operations, while the Application Plane hosts the actual SaaS application instances. This separation of concerns creates clear architectural divisions that enable efficient system management.

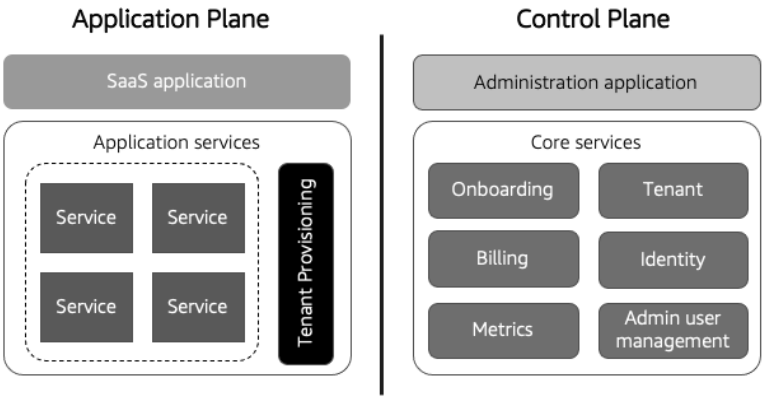


Fig. 5.1: Application Plane vs Control Plane

This two layer architecture provides the foundation for operational efficiency. We extend this model by implementing, (1) a logically centralized yet physically distributed Control Plane for unified management of Application Planes and tenant operations, and (2) regionally distributed Application Planes to achieve global scalability while complying with regional data residency requirements. By

intelligently deploying Application Planes across multiple geographic regions, tenant data can be stored within the defined boundary.

The logically centralized Control Plane orchestrates all Application Plane components and tenant operations, preserving operational efficiency through unified management. This centralized governance streamlines cross regional deployment and administration of Application Planes. Additionally, distributing Control Plane components across regions reduces latency for Control Plane related operations.

During the tenant onboarding process, tenants are dynamically (based on the configurations) assigned to specific Application Planes based on their business location. This ensures global scalability, data compliance and operational efficiency.

The proposed architecture is illustrated in Fig. 5.2.

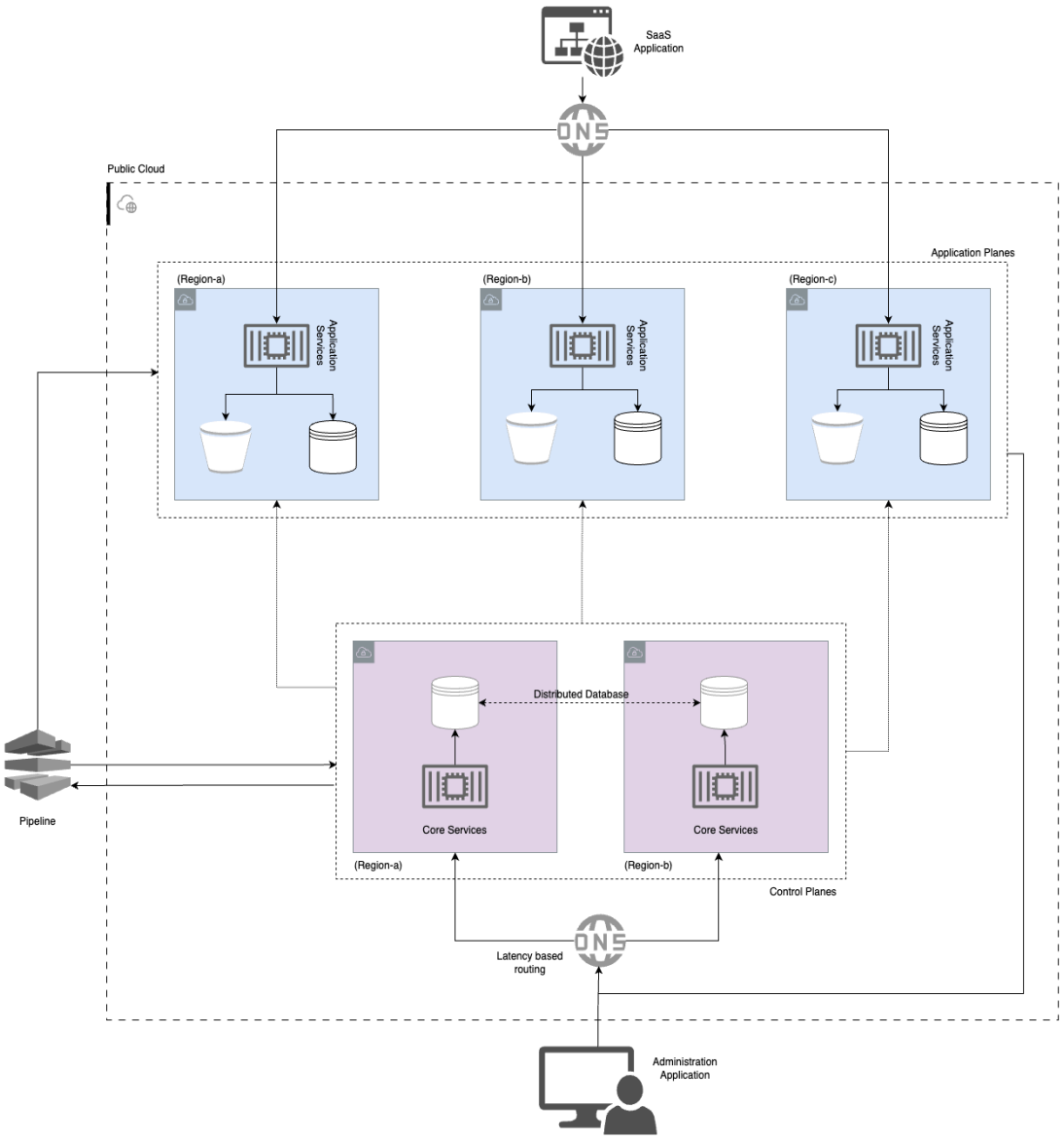


Fig. 5.2: Proposed Architecture

5.1.1 The Logically Centralized, Physically Distributed Control Plane

The Control Plane is globally distributed to improve performance, while maintaining a logically centralized architecture responsible for managing the entire SaaS application. It does not store any tenant specific application data subject to data residency regulations. Instead, it stores only configuration information for the SaaS application in a distributed database, enabling centralized management with physical distribution. Its core functionalities are:

- Tenant Metadata Management: This module manages non sensitive tenant metadata, such as billing information, administrative user information, subscription details, and non sensitive configuration settings. This data is stored in a globally accessible and highly available distributed data storage.

- Tenant Configuration Management: This module handles tenant specific configurations, including determining which Application Plane hosts tenant's application data. These centrally stored configurations ensure efficient routing of tenant users to the appropriate Application Planes.

- Compliance Driven Tenant Scheduling and Routing: This is the key component for achieving data residency. When a new tenant onboards, the Control Plane decides the appropriate Application Plane region based on the tenant's geographical country and applicable data residency regulations. This mapping is stored in a highly available routing table. All user requests from that tenant are then routed by the Control Plane to the desired regional Application Plane instance at user login time. This ensures that tenant data lies within the required geographical boundary.

- Application Plane Deployment and Management: The Control Plane manages the deployment, updates, and monitoring of all Application Plane components across different regions. This includes version management, configuration updates, and health checks. This centralized management simplifies operational overhead and ensures consistency across the distributed Application Planes.

- Configurations: The Control Plane also manages configurations such as regions, countries, and the mapping between them based on regional data residency laws. It maintains the association between regions and Application Planes to enable compliance driven tenant scheduling and request routing. Additionally, it handles country-specific configurations such as tiers, pricing, time zones, languages, and currencies to support country specific customization while preserving operational efficiency.

5.1.2 The Distributed Application Plane

The Application Plane is composed of multiple, geographically distributed components of the SaaS application. Each instance is deployed in a specific region and is responsible for serving tenants whose data must reside in that region.

- Regional Data Storage: Each Application Plane instance manages its own isolated data storage and static file storage, ensuring that tenant data remains within the designated region. Appropriate data isolation techniques (database per tenant, schema per tenant, or row level security) are employed to prevent data leakage between tenants.

- Regional Application Logic and Processing: Each Application Plane instance runs the SaaS application logic and processes user requests using the tenant data stored locally within that region.

- Independent Scalability: Each Application Plane instance can be scaled independently based on regional demand, allowing for efficient resource utilization and optimal performance.

5.1.3 Pipeline

To ensure consistent, repeatable, and automated deployments and management of the Application Plane components across different regions, we implement a robust deployment pipeline leveraging IaC. These pipelines are managed and orchestrated by the centralized Control Plane.

- IaC Tools: We use IaC tools to define and manage the infrastructure of each Application Plane instance. This includes application containers, databases and other required modules. Using IaC allows us to version control our infrastructure configurations, enabling easy rollback and management.

- Control Plane Orchestration: The Control Plane orchestrates the entire deployment pipeline. When a new region is added or an existing one requires an update, it triggers the appropriate deployment pipelines. This centralized orchestration reduces the management overheads.

5.2 Tenant Scheduling and Routing

Tenant scheduling and routing are key concepts in our proposed architecture, facilitating the storing and retrieval of tenant specific SaaS application data from the specified Application Plane. To enable this, a centrally managed routing table is maintained within the Control Plane. This table makes sure seamless mapping and efficient access to tenant data across Application Planes. The diagram in Fig. 5.3, shows the relationship between regions, countries, Application Planes, tenants and admin users.

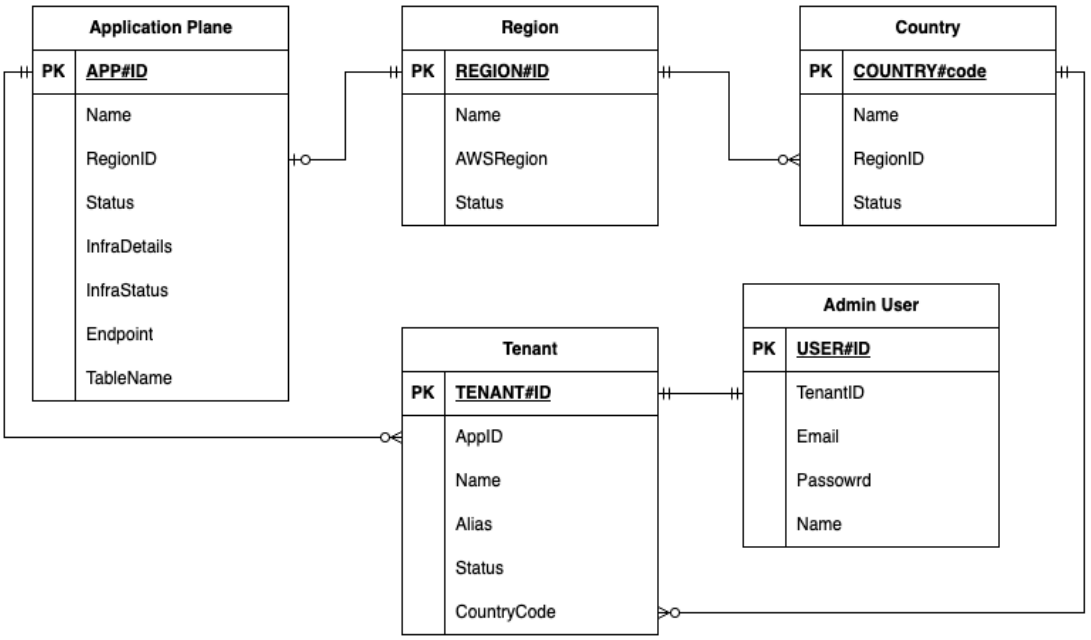


Fig. 5.3: Control Plane Data Model (ERD)

We can store pre configured values for regions which link public cloud data center regions such as AWS regions. A new Application Plane can then be deployed in each configured region. Additionally, countries can also be pre mapped based on regional residency regulations with regions. For example, countries within the European Union can be mapped to specific data center regions in EMEA. Similarly, each country can be associated with a data center region according to its residency laws. These pre configured values serve as a routing table which enables efficient tenant scheduling and routing.

5.2.1 Tenant Scheduling

Tenant scheduling is executed during the onboarding process. When a new tenant registration request is received, the tenant's country is used to get the Application Plane details from the pre configured routing table. If a country does not map to any region or Application Plane, the system responds with a "Service Not Available" message. If a match is found, the tenant is created in the Control Plane database by storing the tenant and billing information. Corresponding details are also stored in the particular Application Plane database based on the routing table. Since the routing table is configured according to data residency regulations, the tenants are assigned to the Application Planes that comply with regional residency rules.

The diagram in Fig. 5.4 shows a simple flowchart of the tenant onboarding process.

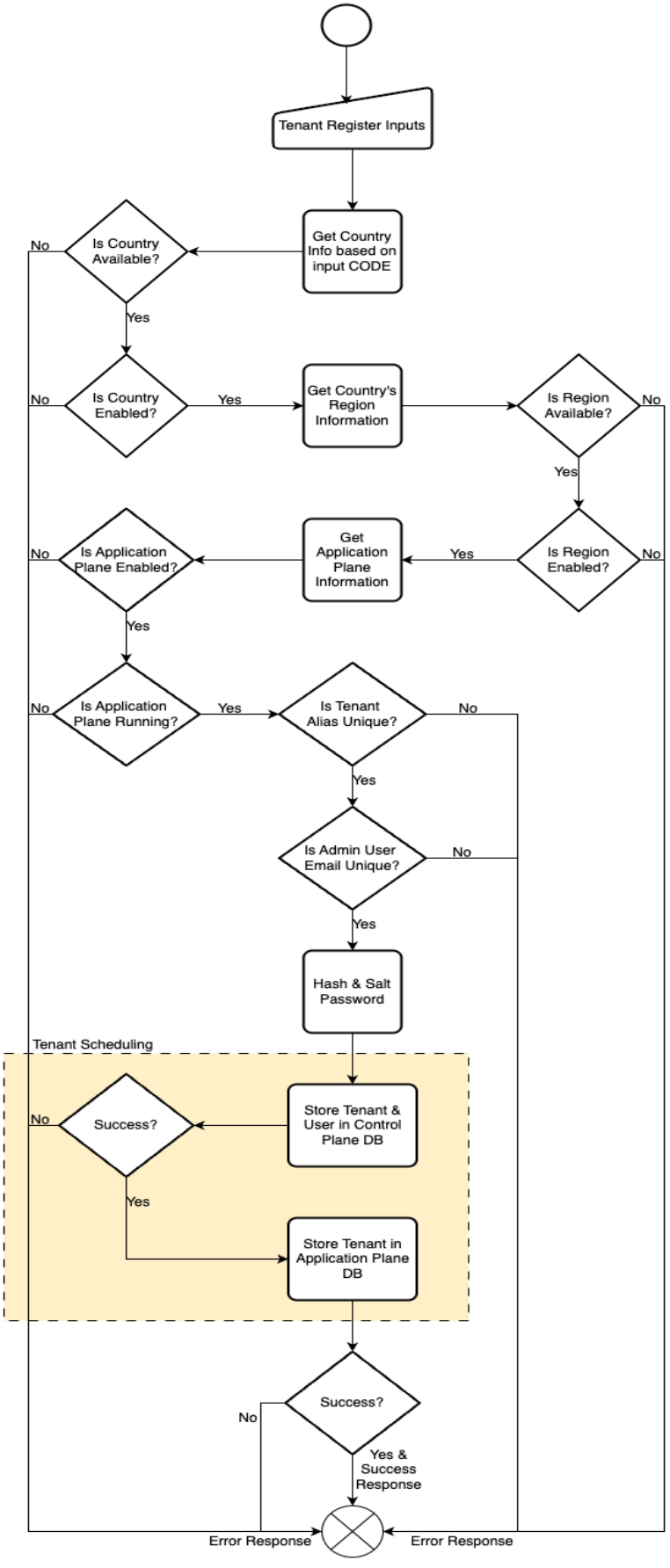


Fig. 5.4: Tenant Registration Flowchart

5.2.2 Tenant Routing

Tenant routing occurs in two scenarios. First, during the tenant admin logs in, the Control Plane responds with an access token and the endpoint of the assigned Application Plane. The admin then uses this endpoint to access the home page and other resources. The second scenario happens when a tenant user attempts to log in, the user has a unique tenant alias. This alias is used to query the Control Plane for tenant specific details, including the corresponding Application Plane information. Based on the retrieved data, along with the provided email and password, the login request is forwarded to the appropriate Application Plane.

This centralized routing mechanism ensures that tenant requests are efficiently and accurately directed to their respective Application Planes. The diagram in Fig. 5.5 illustrates the sequence diagram for both tenant admin and user login.

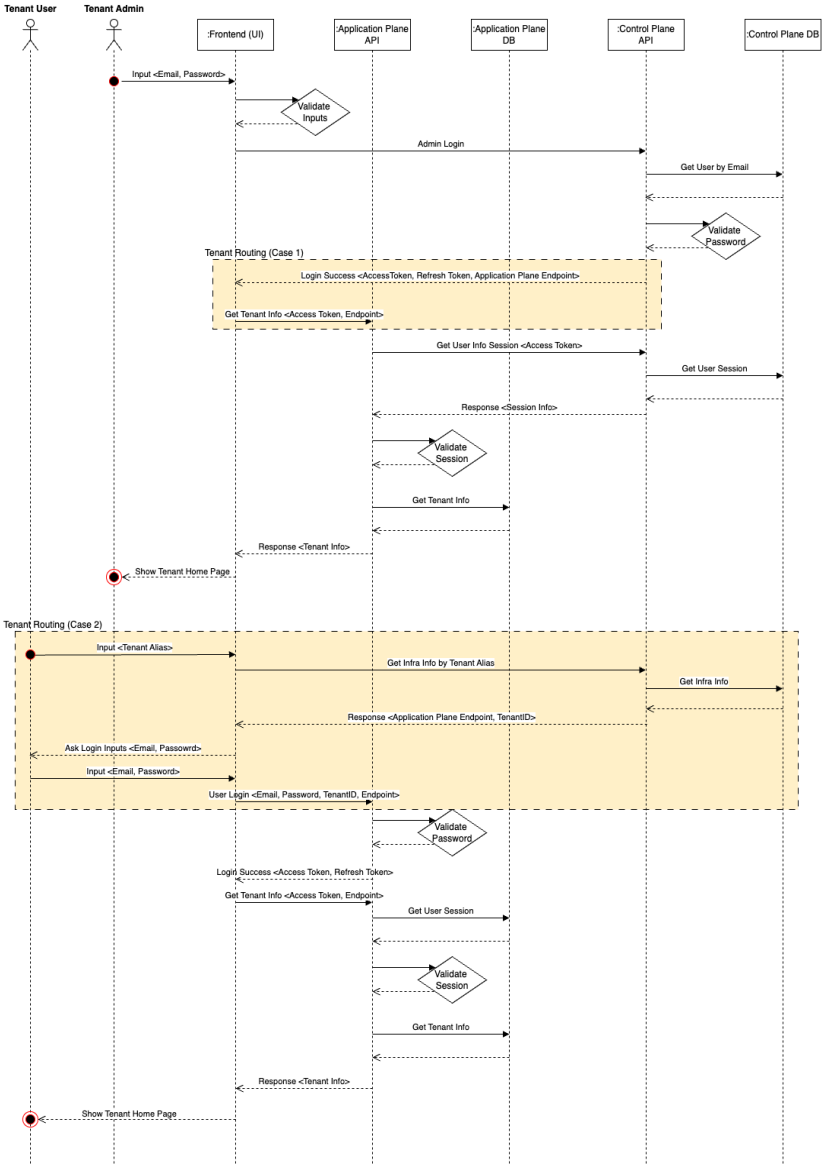


Fig. 5.5: Tenant User Login Sequence Diagram

5.3 Addressing the Research problem

This proposed architecture addresses the defined research problems by,

- Enabling Global Scalability: By distributing the Application Plane across multiple regions, the architecture can handle a global user base and adapt to different regional demands. The centralized Control Plane simplifies the management of these distributed components.
- Ensuring Data Residency Compliance: The compliance driven tenant scheduling and routing techniques in the Control Plane guarantees that tenant data is stored and processed within the required geographical boundaries, adhering to regional data residency regulations.
- Maintaining Operational Efficiency: The centralized Control Plane simplifies tenants management and Application Planes management which reduces operational overhead and ensures consistency across the distributed Application Planes.

5.4 Evaluation

To assess the feasibility and effectiveness of the proposed globally scalable and compliant multi-tenant SaaS architecture, we will conduct a practical evaluation across various architectural setups (detailed in upcoming sections). This evaluation will be based on a mock implementation using a retail management scenario, providing a relevant and realistic context for analysis.

The mock implementation enables a comprehensive evaluation of critical architectural aspects, including performance (measured by latency), global scalability (inferred from performance across regions), compliance with data residency requirements (assessed through correct tenant placement in Application Planes based on configuration), and operational overhead (measured by the time required to deploy changes across distributed Application Plane instances).

These dimensions, latency, global scalability, data residency compliance, and operational overheads, serve as our key evaluation parameters for validating the proposed architecture.

CHAPTER 6

IMPLEMENTATION DETAILS

In this chapter, we take a deep dive into the implementation details of each component within our proposed architecture for experimentation. We begin by examining the technology stack used and how each tool was utilized to build the components. Following that, we explore the implementation specifics of our two main components: the Control Plane and the Application Plane.

6.1 Tech Stack

To build our components, we used a combination of various technologies to implement the two main parts of our system: the Control Plane and the Application Plane.

- Golang – It is a popular programming language well suited for building cloud native applications. In our architecture, we used Golang to develop backend APIs such as the Tenant Manager API (part of the Control Plane) and the Retail Manager API (part of the Application Plane). Each of these APIs is responsible for handling user data, executing business logic as defined by our proposed architecture, interacting with the database for data retrieval and storage, and integrating with third-party services such as email providers and CI/CD pipelines.

- Docker – We use Docker to containerize our backend APIs, allowing them to run consistently across different environments and machines. This helps eliminate platform and operating system dependencies, ensuring smoother deployment and scalability in both development and production environments.

- AWS – We use AWS as our only cloud provider, utilizing a range of managed services to design, implement, and deploy the two core components of our architecture: the Control Plane and the Application Plane.

- AWS ECR (Elastic Container Registry) – ECR is a fully managed AWS service used to store, manage, and version Docker container images. In our architecture, we use ECR to store the Docker images of our backend APIs, enabling seamless deployment and version control.

- AWS App Runner – App Runner is a managed AWS service that allows us to run containerized applications without managing servers. We use it to deploy and run our backend APIs directly from the Docker images stored in ECR. App Runner also provides several builtin features, including automatic scaling, HTTPS support, access to application logs and metrics, an auto generated domain name, and support for custom domain mapping.

- AWS DynamoDB – DynamoDB is a fully managed NoSQL database service by AWS, optimized for key value and document based data storage. We use DynamoDB across both backend APIs to efficiently store and manage application data. Additionally, it is integrated into our Terraform provisioning setup as a state lock mechanism, ensuring safe and consistent infrastructure updates.

  DynamoDB supports Global Tables, which allow us to replicate data across multiple AWS regions. This capability enables us to deploy multiple Control Plane components in different regions while maintaining a consistent and centrally managed dataset, supporting high availability and scalability.

  Furthermore, DynamoDB offers builtin monitoring features such as logs and performance metrics, which improve observability and simplify operational troubleshooting.

- AWS S3 (Simple Storage Service) – S3 is a scalable, fully managed storage service provided by AWS. In our Application Plane, we use S3 to allow SaaS users to store and manage static files. We also leverage S3 to store Terraform state files, supporting the automated provisioning of infrastructure for both the Control Plane and the Application Plane.

- Amazon Route 53 – Route 53 is a fully managed DNS (Domain Name System) service by AWS that enables domain registration, DNS routing, and health checking. We use Route 53 to manage our domain and configure custom domain routing for our App Runner services. This is especially important in our architecture, where we may have multiple Control Planes and Application Planes running simultaneously, each requiring custom domain mappings and flexible routing configurations.

- Terraform – We use Terraform as an IaC tool to automate the infrastructure provisioning of the main two components: the Control Plane and the Application Plane. Since each component uses AWS-managed services, it becomes easy and fast to automate resource provisioning in the cloud. Terraform also supports simple infrastructure updates and destruction, allowing us to efficiently automate the entire lifecycle of our component infrastructure management.

- GitLab CI – We use GitLab CI as our CI/CD pipeline to automate various tasks and operations. In our architecture, GitLab pipelines are used to automatically build and push backend API Docker images to Amazon ECR whenever a new tag is created. Additionally, we use GitLab pipelines to automate Terraform provisioning, enabling infrastructure management for both the Control Plane and the Application Plane.

To manage the Control Plane infrastructure, we trigger the pipeline manually through the GitLab web interface. For the Application Plane, we trigger the pipeline via GitLab APIs, which are integrated with the Tenant Manager API. This integration allows seamless and automated management of the Application Plane through the Control Plane, ensuring efficient and scalable operations for our SaaS platform.

6.2 Control Plane

Let’s look at a deep dive into the Control Plane architecture, exploring scenarios with a single Control Plane as well as multiple Control Planes. We will also cover how to set up the Control Plane infrastructure effectively

6.2.1 Architecture

In the Control Plane, we use AWS App Runner to host and run the Tenant Manager API. This API is responsible for managing tenants and interacts with external services such as email servers to send emails and the GitLab API to trigger Application Plane pipelines.

The Control Plane also includes AWS DynamoDB, which is used by the Tenant Manager API to store and retrieve tenant and Application Plane related data. We configure DynamoDB as a Global Table to enable multi region replication:

- When there is a single Control Plane component, we use a primary DynamoDB instance in one region.

- When multiple Control Plane components are deployed across different regions, we add replica DynamoDB instances in those regions to ensure data availability and consistency.

AWS Route 53 helps to assign a custom domain to the App Runner instance running the Tenant Manager API. This is achieved by adding a CNAME record pointing to the App Runner endpoint. Additionally, we configure a latency based routing policy, which ensures that when multiple Control Plane components are available across regions, traffic to the Tenant Manager API is routed to the nearest instance based on the user's location. This setup provides improved performance and availability for users accessing the Control Plane services.

Fig. 6.1 and Fig. 6.2 illustrate the Control Plane architecture for:

1. A single Control Plane component

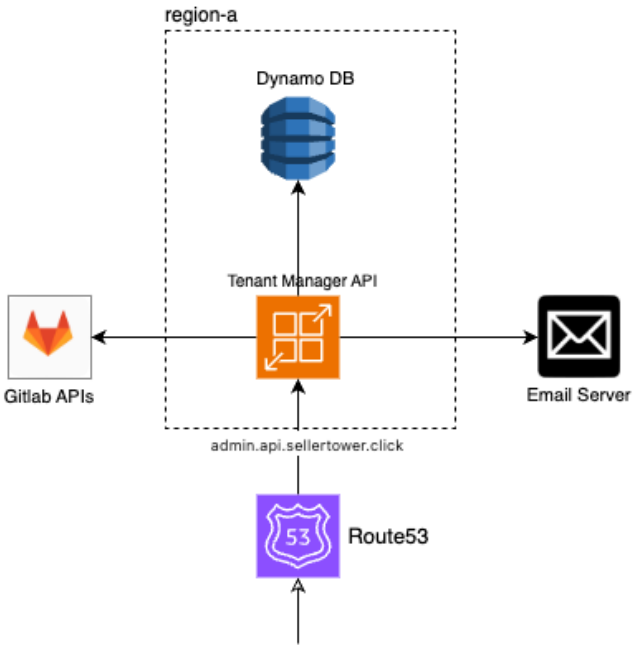


Fig. 6.1: Single Control Plane Architecture

2. Multiple Control Plane components with region-specific replicas and latency-based routing

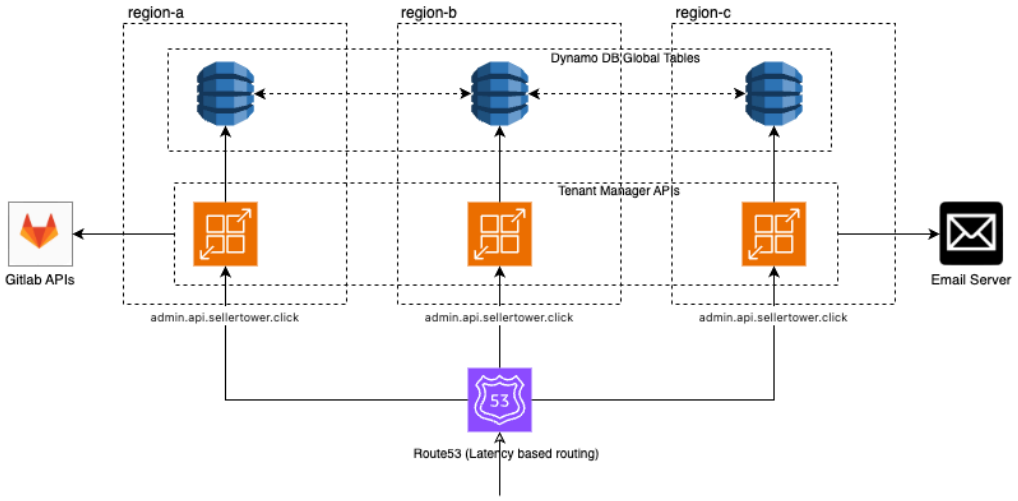


Fig. 6.2: Multiple Control Plane Architecture

6.2.2 Infra setup

In the Control Plane architecture uses a wide range of AWS serverless services to provision and manage resources. To ensure consistent and repeatable deployments, we employ Terraform as our IaC tool. Prior to infrastructure provisioning, the Tenant Manager API is containerized using Docker, and the resulting image is pushed to Amazon ECR. This process is automated through a GitLab CI/CD pipeline, which is

triggered upon the creation of a new Git tag in the Tenant Manager API repository. The pipeline handles both the Docker image build and push to ECR.

Once the image is available, Terraform modules, designed for modular and reusable infrastructure definitions, are used to provision Control Plane components. These modules support deployment across multiple AWS regions, allowing for scalability and redundancy. These modules also handle tasks like deploying new versions of the Tenant Manager API, as well as starting and stopping service instances, and destroying a Control Plane component.

To streamline and standardize these operations, a dedicated GitLab pipeline is implemented to manage the entire lifecycle of the Control Plane infrastructure. This pipeline accepts a set of input parameters including:

- The current infrastructure version
- The version of the Tenant Manager API
- The intended action (e.g., deploy, start, stop, destroy)
- The target AWS region

Based on these parameters, the pipeline automates the necessary tasks to configure and manage Control Plane components. This mechanism enables efficient deployment and maintenance of multiple Control Plane instances across geographically distributed regions, ensuring high availability, scalability, and operational consistency and efficiency.

6.3 Application Plane

This section presents a detailed discussion of the Application Plane architecture and outlines the process involved in provisioning and configuring Application Plane components.

6.3.1 Architecture

In the Application Plane, AWS App Runner is utilized to host the Retail Manager API, which serves as a core service for managing retail business operations. This API integrates with several AWS managed services to fulfill its functionality. Specifically, AWS DynamoDB is used to store and retrieve retail related data efficiently, while AWS S3 is employed as a static file storage solution to manage business related images and media files.

Additionally, AWS Route 53 is used to configure custom domains for each App Runner service instance. Each Application Plane component is assigned a unique subdomain, enabling distinct identification and access to individual components via domain based routing.

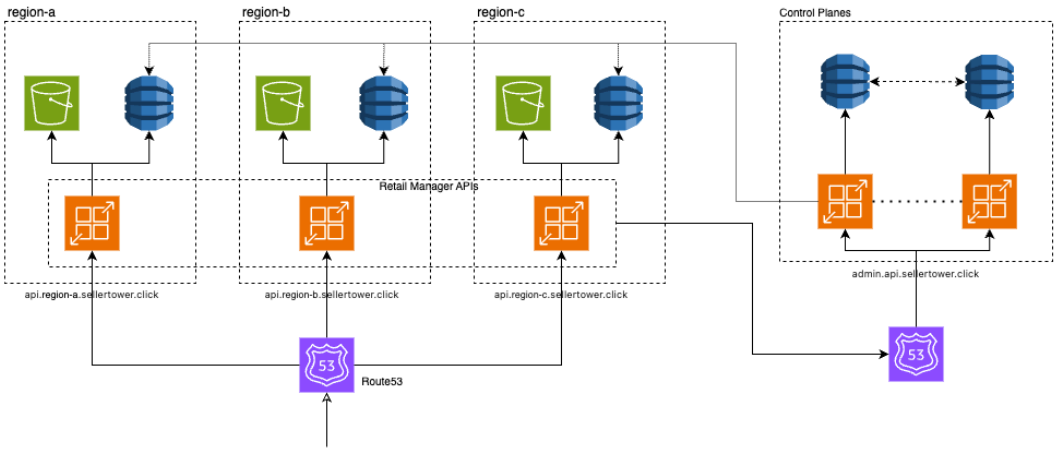


Fig. 6.3: Application Plane Architecture

6.3.2 Infra setup

A GitLab pipeline is employed to automate the Docker image build and push process for the Retail Manager API whenever a new GitLab tag is created in the corresponding repository. This automation ensures that the application components are consistently containerized and stored in Amazon ECR.

To enable consistent and repeatable deployments of AWS-managed infrastructure for the Application Plane components, Terraform is utilized. Reusable Terraform modules are developed to abstract the infrastructure definitions, enabling seamless deployment of Application Plane components across multiple AWS regions. These modules also facilitate common operational tasks such as deploying new versions of the Retail Manager API, starting and stopping service instances, and destroying Application Plane environments when necessary.

To orchestrate these processes, a dedicated GitLab CI/CD pipeline manages the complete lifecycle of Application Plane components. This pipeline accepts a set of parameters, including:

- The infrastructure version
- The Retail Manager API version
- The desired operation (e.g., deploy, start, stop, destroy)
- The target AWS region
- Associated infrastructure metadata (e.g., DynamoDB table name, S3 bucket name, and the unique subdomain)

Furthermore, these pipelines are programmatically triggered by the Tenant Manager API residing in the Control Plane component via GitLab APIs. This setup allows for centralized and efficient management of Application Plane deployments, ensuring consistency, scalability, and operational efficiency across the SaaS architecture.

CHAPTER 7

EXPERIMENTAL RESULTS

This section presents a comprehensive evaluation of our proposed architecture through a structured experimentation process. The primary objective is to assess its global scalability, performance, data residency compliance, and overall operational efficiency. To this end, we evaluate three distinct architectural configurations, each comprising different combinations of Control Plane and Application Plane components. These configurations are selected based on their alignment with the optimization of our defined evaluation parameters.

We begin with an overview of the experimental setup, detailing the infrastructure configurations used to simulate and compare the selected architectural models. This includes both single and multi region deployments of control and Application Plane components. Each configuration is designed to surface trade-offs in latency, scalability, and operational complexity, enabling a balanced and fair comparison.

Subsequently, we outline the test cases developed to validate the experimental environment. These test cases emulate realistic operational scenarios. Each case is mapped to measurable performance indicators that support our research objectives. Next, we describe the test data used in these scenarios. The dataset replicates real world retail business activity. We then detail the testing infrastructure utilized for executing the experiments. This includes a suite of automation tools. This environment ensures a repeatable and controlled evaluation process across all test cases.

Finally, we visualize and analyze the results obtained from each experimental setup. Comparative analysis across different architectures is performed based on the collected metrics, highlighting the performance trade-offs and regional impact on system behavior. The insights derived from this analysis guide our understanding of the optimal architecture for global SaaS deployment. We conclude the section with a summary of the findings and key recommendations based on empirical evidence from the experimentation.

7.1 Experimental Setup

To evaluate the effectiveness of our proposed architecture, we consider three distinct architectural setups (including the proposed architecture), each featuring a unique combination of Control Plane and Application Plane configurations. These combinations are selected based on specific assumptions intended to validate our evaluation parameters across all setups. For testing purposes, we use ten different countries as request sources, simulating real world scenarios. Each architecture is configured accordingly to handle requests from these countries. In this section, we

provide a detailed analysis of each architectural setup, including its configuration and infrastructure management.

7.1.1 Single Region (1CPx1AP)

7.1.1.1 Architecture

This setup consists of a single Control Plane managing a single Application Plane. Both components are deployed in the same AWS region (ap-south-1). The Fig. 7.1 shows the overview architecture of this setup.

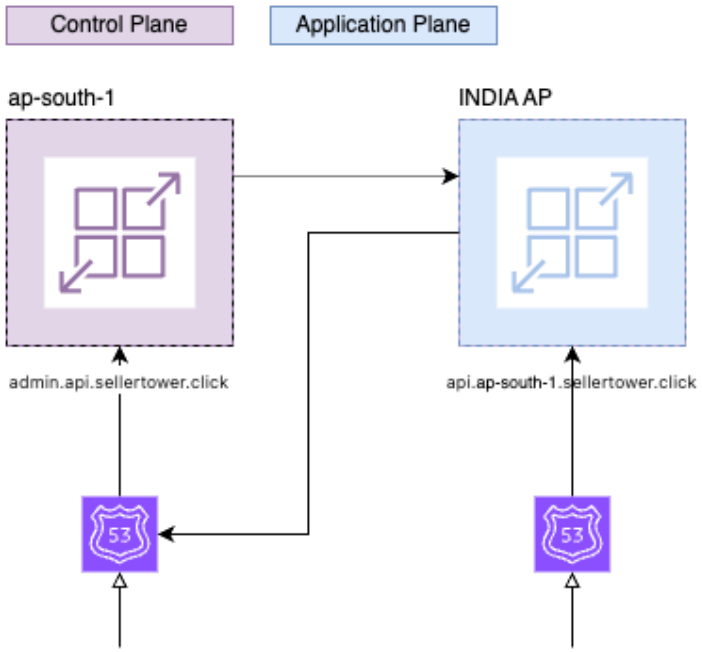


Fig. 7.1: 1CPx1AP Architecture

7.1.1.2 Pre Configuration

In the Control Plane component, we have configured the region to Application Plane mapping, as detailed in the tables Table. 7.1 and Table. 7.2.

Table. 7.1: 1CPx1AP Application plane

Application Plane Name	AWS Region	Endpoint
INDIA AP	ap-south-1	https://api.ap-south-1.sellertower.click

Table. 7.2: 1CPx1AP Region

Region ID	Region Name	AWS Region	Application Plane
5ce58926-eaf9-4081-9243-737b7b930a5a	INDIA	ap-south-1	INDIA AP

The Table. 7.3. shows the configuration of mapping between countries and regions.

Table. 7.3: 1CPx1AP Countries

Country Code	Country Name	Region ID
USA	America	5ce58926-eaf9-4081-9243-737b7b930a5a
IND	India	5ce58926-eaf9-4081-9243-737b7b930a5a
MY	Malaysia	5ce58926-eaf9-4081-9243-737b7b930a5a
ID	Indonesia	5ce58926-eaf9-4081-9243-737b7b930a5a
ES	Spain	5ce58926-eaf9-4081-9243-737b7b930a5a
UAE	UAE	5ce58926-eaf9-4081-9243-737b7b930a5a
CA	Canada	5ce58926-eaf9-4081-9243-737b7b930a5a
FR	France	5ce58926-eaf9-4081-9243-737b7b930a5a
AUS	Australia	5ce58926-eaf9-4081-9243-737b7b930a5a
LK	Sri Lanka	5ce58926-eaf9-4081-9243-737b7b930a5a

Based on the above configuration, whenever a business is created from any of the specified countries, its information is scheduled to be created in the INDIA AP Application Plane.

7.1.1.3 Components Management

In this architectural setup, a GitLab pipeline is utilized to create a single Control Plane component. Through this Control Plane, an Application Plane is provisioned by triggering another GitLab pipeline. The images in Fig. 7.2 and Fig. 7.3 illustrates the completed GitLab pipelines for the creation of both the Control Plane and the Application Plane.

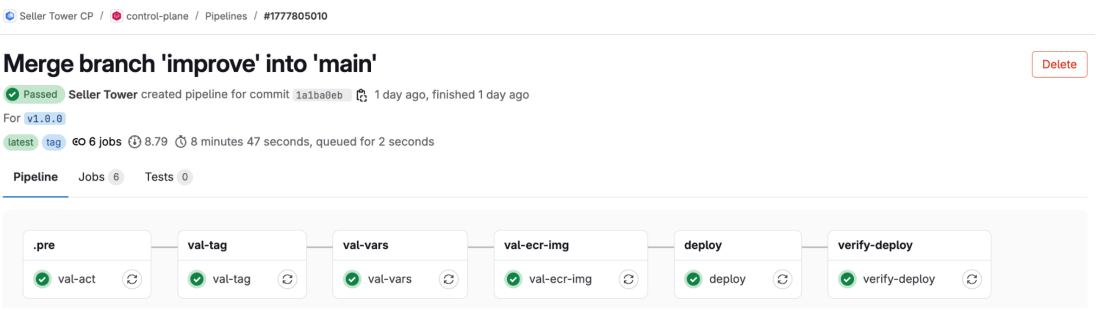


Fig. 7.2: Control Plane Deploy Gitlab Pipeline (1CPx1AP)

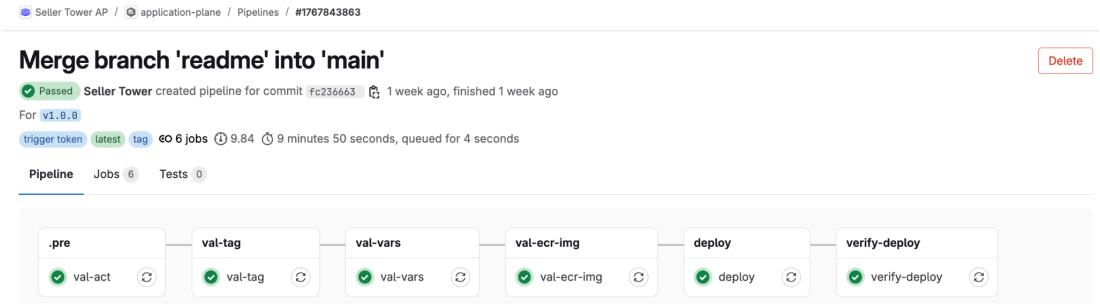


Fig. 7.3: Application Plane Deploy Gitlab Pipeline (1CPx1AP)

After the experimentation phase, a pipeline was triggered from the Control Plane to initiate the destruction of the Application Plane. Subsequently, the Control Plane itself was destroyed using a separate pipeline. The Fig. 7.4 and Fig. 7.5 shows the completed GitLab pipelines for the teardown of both the planes.

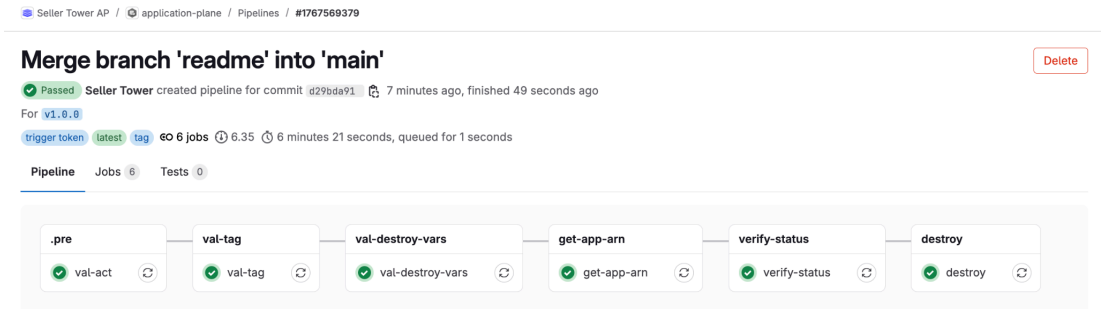


Fig. 7.4: Application Plane Destroy Gitlab Pipeline (1CPx1AP)

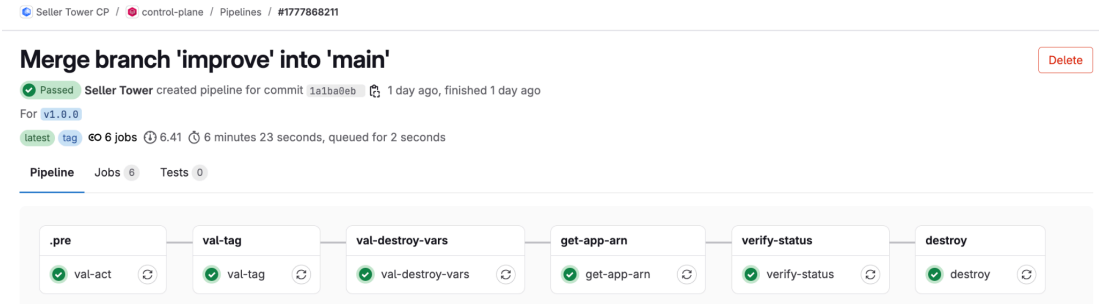


Fig. 7.5: Control Plane Destroy Gitlab Pipeline (1CPx1AP)

7.1.2 Hybrid (1CPxnAP)

7.1.2.1 Architecture

This setup features a single Control Plane that manages multiple Application Planes. The Control Plane (ap-south-1) and each Application Plane is deployed across different AWS regions to address data residency compliance, as per our defined assumptions.

The architecture diagram in Fig. 7.6 provides an overview of this setup.

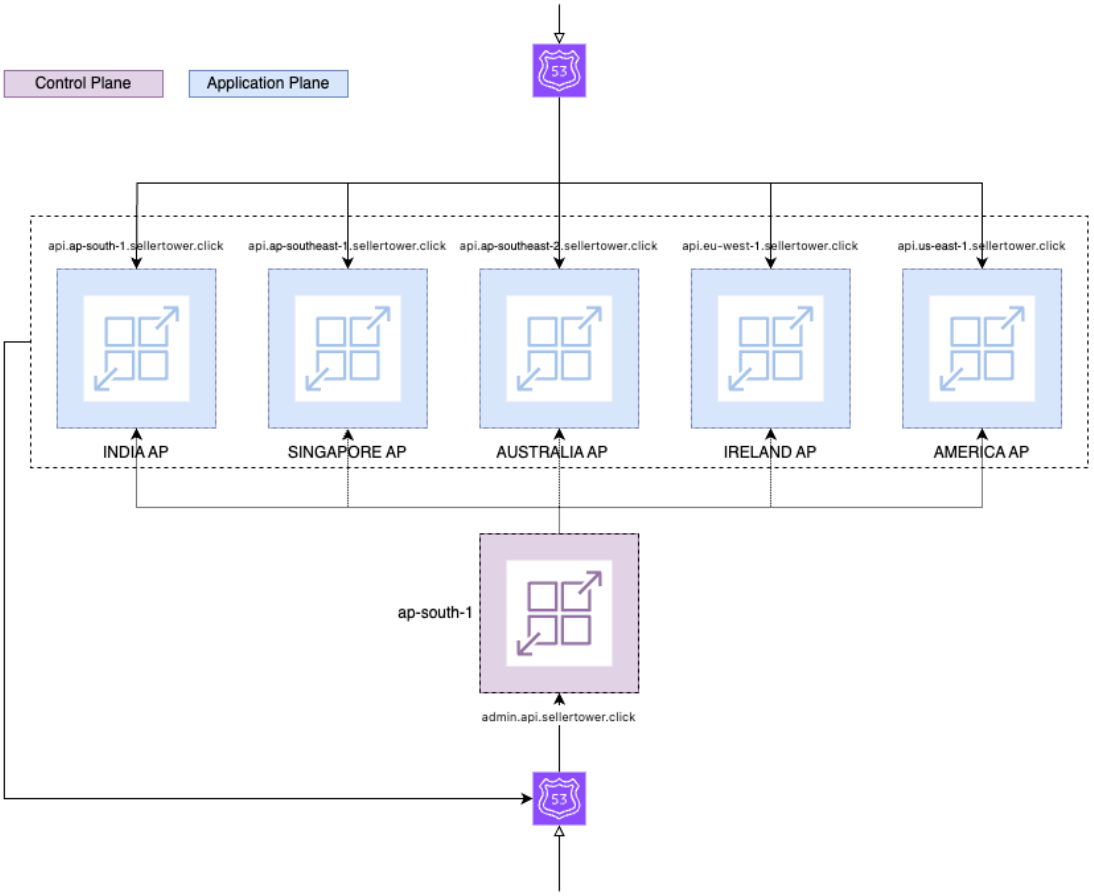


Fig. 7.6: 1CPxnAP Architecture

7.1.2.2 Pre Configuration

Inside the Control Plane component, we have configured the region to Application Plane mapping, as detailed in the Table. 7.4 and Table. 7.5.

Table. 7.4: 1CPxnAP Application planes

Application Plane Name	**AWS Region**	**Endpoint**
INDIA AP	ap-south-1	https://api.ap-south-1.sellertower.click
SINGAPORE AP	ap-southeast-1	https://api.ap-southeast-1.sellertower.click
AUSTRALIA AP	ap-southeast-2	https://api.ap-southeast-2.sellertower.click
IRELAND AP	eu-west-1	https://api.eu-west-1.sellertower.click
AMERICA AP	us-east-1	https://api.us-east-1.sellertower.click

Table. 7.5: 1CPxnAP Regions

Region ID	Region Name	AWS Region	Application Plane
5ce58926-eaf9-4081-9243-737b7b930a5a	INDIA	ap-south-1	INDIA AP
6ff682fe-cbb0-4f14-86ab-3f1f59b7ff93	SINGAPORE	ap-southeast-1	SINGAPORE AP
82fde3a2-8e37-44be-9988-1f86e2b4b957	AUSTRALIA	ap-southeast-2	AUSTRALIA AP
2d504246-095b-4575-be2f-88d972db6855	IRELAND	eu-west-1	IRELAND AP
855af49f-815e-41d5-8bfd-8b48741dbea8	AMERICA	us-east-1	AMERICA AP

The Table. 7.6 shows the configuration of mapping between countries and regions.

Table. 7.6: 1CPxnAP Countries

Country Code	Country Name	Region ID
USA	America	855af49f-815e-41d5-8bfd-8b48741dbea8
IND	India	5ce58926-eaf9-4081-9243-737b7b930a5a
MY	Malaysia	6ff682fe-cbb0-4f14-86ab-3f1f59b7ff93
ID	Indonesia	6ff682fe-cbb0-4f14-86ab-3f1f59b7ff93
ES	Spain	2d504246-095b-4575-be2f-88d972db6855
UAE	UAE	5ce58926-eaf9-4081-9243-737b7b930a5a
CA	Canada	855af49f-815e-41d5-8bfd-8b48741dbea8
FR	France	2d504246-095b-4575-be2f-88d972db6855
AUS	Australia	82fde3a2-8e37-44be-9988-1f86e2b4b957
LK	Sri Lanka	6ff682fe-cbb0-4f14-86ab-3f1f59b7ff93

Based on the above configuration, whenever a business is created from any of the specified countries, its information is directed to the appropriate Application Plane. These mappings are defined according to our assumed data residency requirements. As a result, business data is stored in compliance with regional data residency laws.

7.1.2.3 Components Management

In this architectural setup, a GitLab pipeline is used to create a single Control Plane component. Through this Control Plane, multiple Application Planes are provisioned by triggering separate GitLab pipelines. Fig. 7.7 and Fig. 7.8 illustrate the completed pipelines for the creation of both the Control Plane and the Application Planes.

Fig. 7.7: Control Plane Deploy Gitlab Pipeline (1CPxnAP)

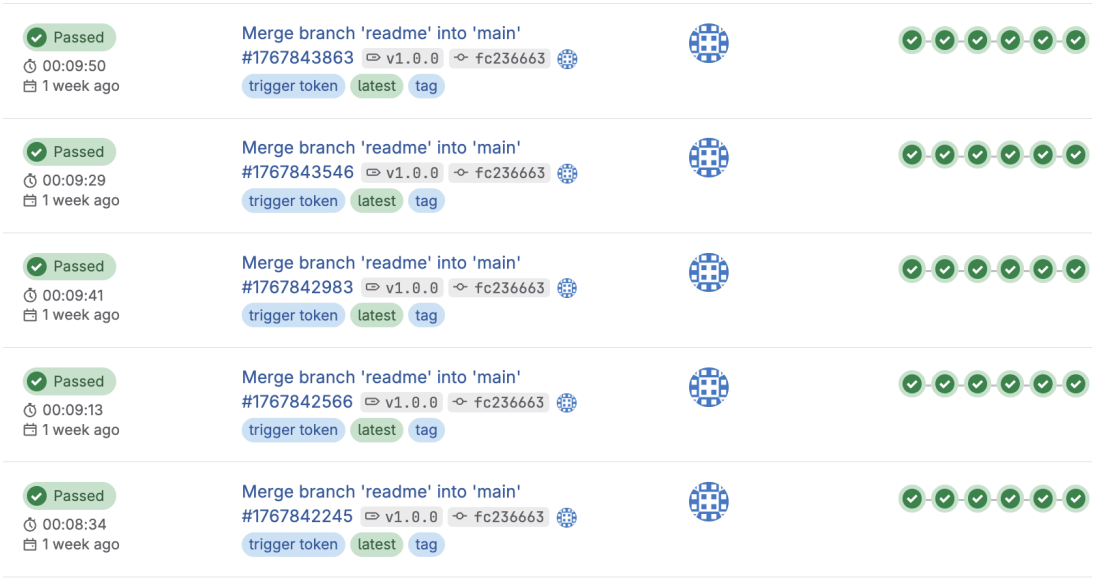


Fig. 7.8: Application Plane Deploy Gitlab Pipelines (1CPxnAP)

After the experimentation phase, the Control Plane triggered multiple pipelines to initiate the destruction of the Application Planes. Subsequently, the Control Plane itself was destroyed using a separate pipeline. Fig. 7.9 and Fig. 7.10 shows the completed GitLab pipelines for tearing down both the control and Application Planes.

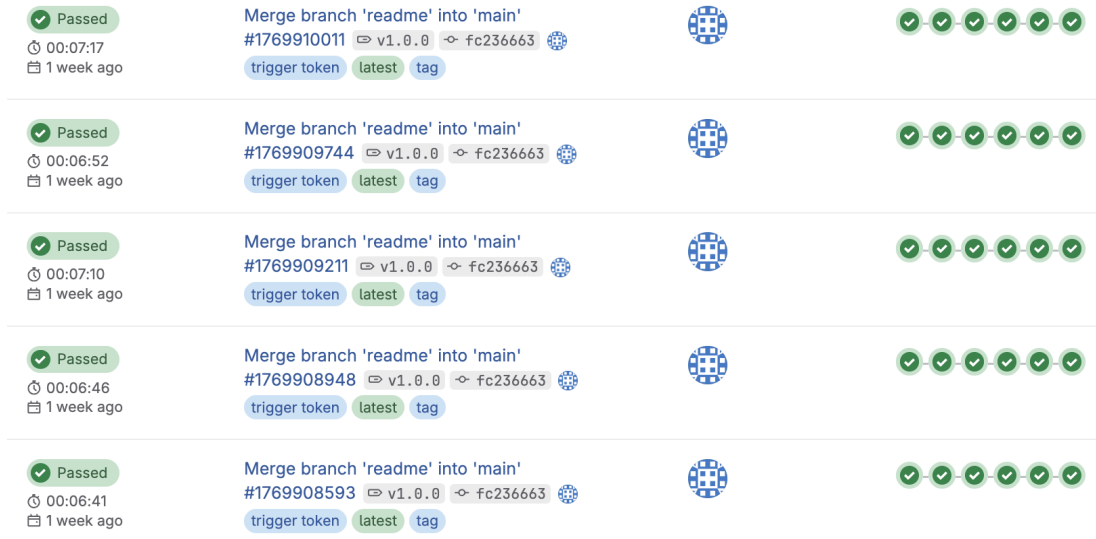


Fig. 7.9: Application Plane Destroy Gitlab Pipelines (1CPxnAP)

Fig. 7.10: Control Plane Destroy Gitlab Pipeline (1CPxnAP)

7.1.3 Fully Distributed (nCPxnAP)

7.1.3.1 Architecture

This setup features multiple Control Planes that manage multiple Application Planes. The Control Planes are deployed across various AWS regions to reduce latency for certain test cases. Each Application Plane is also deployed in different AWS regions for similar objectives, as described in the previous architecture. The architecture diagram in Fig. 7.11 provides an overview of this setup.

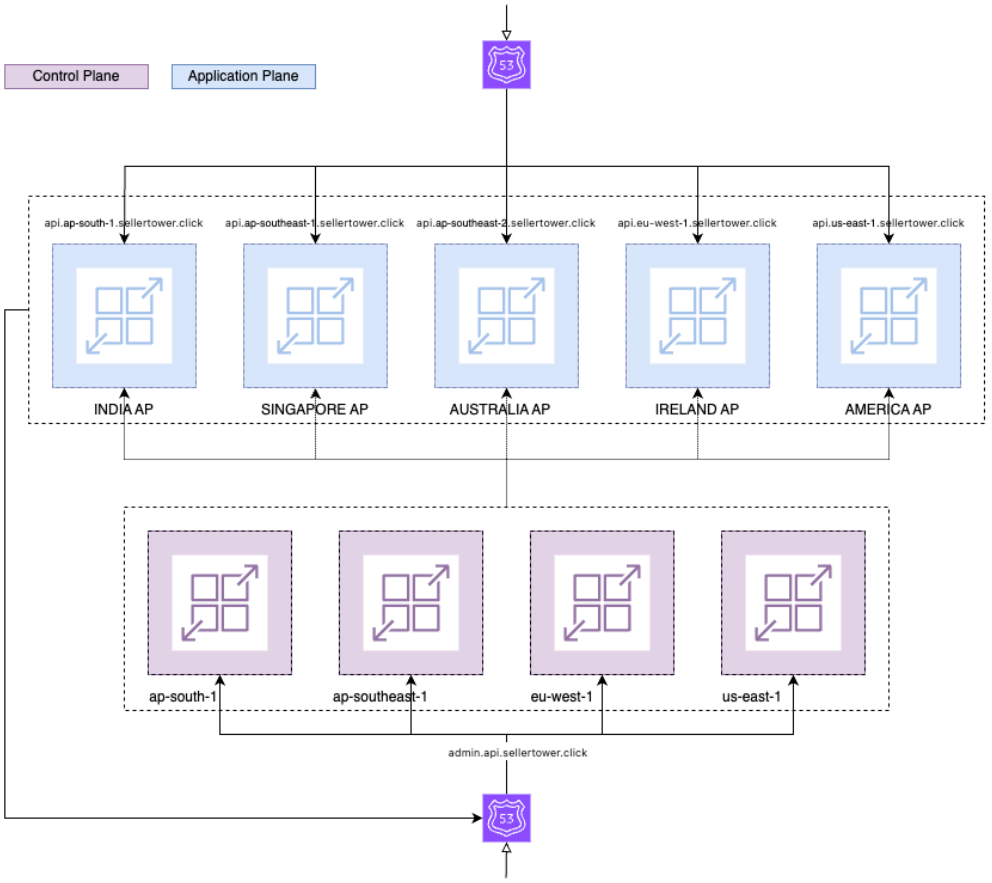


Fig. 7.11: nCPxnAP Architecture

7.1.3.2 Pre Configuration

The Table. 7.7 shows the deployment of Control Planes across different regions.

Table. 7.7: nCPxnAP Control planes

AWS Region	**Component Type**
ap-south-1	Primary
ap-southeast-1	Secondary
eu-west-1	Secondary
us-east-1	Secondary

The configuration of Application Planes, regions and countries are similar to previous architecture (1CPxnAP). Based on the configuration business information is stored according to data residency law as previous architecture. Additionally, this architecture offers reduced latency for certain test cases due to the presence of multiple Control Planes distributed across regions.

7.1.3.3 Components Management

In this architectural setup, multiple GitLab pipelines are used to create several Control Plane components. Through these Control Planes, multiple Application Planes are provisioned by triggering separate GitLab pipelines similar to previous architecture (1CPxnAP). The images in Fig. 7.12 and Fig. 7.13 illustrates the completed pipelines for the creation of the Control Planes.

Fig. 7.12: Control Plane Deploy Gitlab Pipelines (nCPxnAP)

After the experimentation phase, the Control Planes triggered multiple pipelines to initiate the destruction of the Application Planes similar to previous architecture (1CPxnAP). Subsequently, the Control Planes themselves were destroyed using separate pipelines. The images below show the completed GitLab pipelines for tearing down the Control Planes.

Fig. 7.13: Control Plane Destroy Gitlab Pipelines (nCPxnAP)

7.2 Test Cases

To evaluate our experimental setups, we selected specific test cases designed to measure the defined evaluation parameters. In this section, we provide a detailed analysis of each test case.

7.2.1 Business Register

The *Business Register* test case registers a new business using input parameters such as the business name, alias, country code, and admin user details (email and password). This test case helps evaluate compliance with data residency laws by storing business information in the Tenant Manager database and, based on the country code, storing corresponding data in the Retail Manager database. Latency is measured to assess performance. This test case is a rare occurrence in our system. The sequence diagram in Fig. 7.14 illustrates the business registration process.

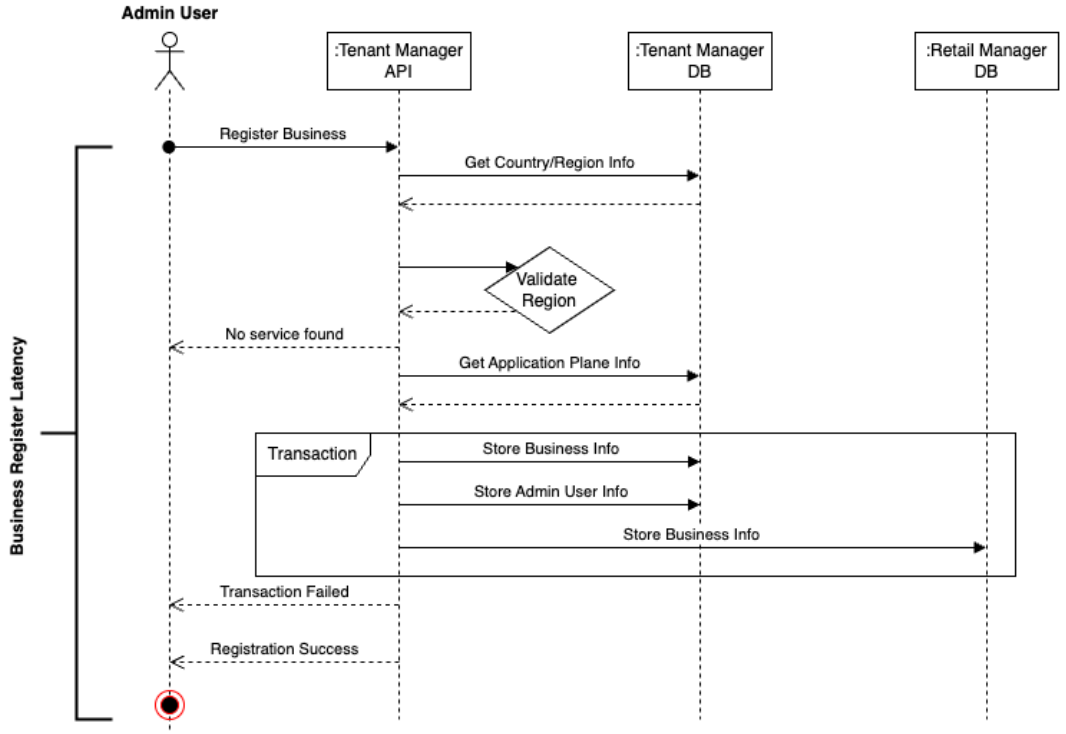


Fig. 7.14: Business Register Sequence Diagram

7.2.2 Admin Login

The *Admin Login* test case interacts only with the Control Plane components. It takes email and password as input parameters and is used to measure system performance through latency. It also serves as a prerequisite for executing other test cases. This is a somewhat rare occurrence in the system. The sequence diagram in Fig. 7.15 illustrates the admin login process.

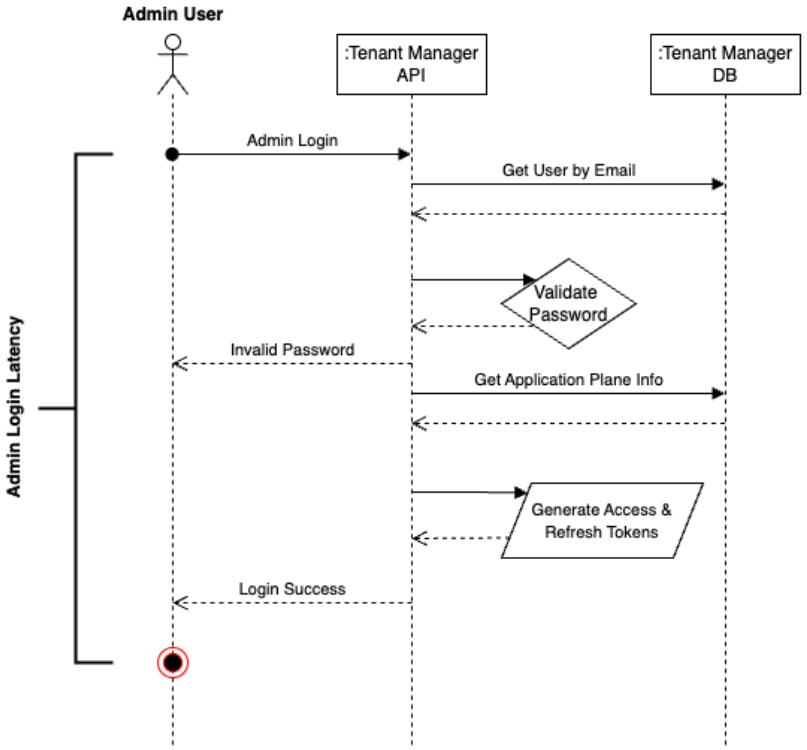


Fig. 7.15: Admin Login Sequence Diagram

7.2.3 Admin Get Business

The *Admin Get Business* test case uses the access token returned by the admin login to retrieve business information. It also uses the Application Plane endpoint provided during login to validate tenant routing, ensuring compliance with data residency laws. This is a frequent operation for business admin users interacting with the system. The sequence diagram in Fig. 7.16 illustrates this process.

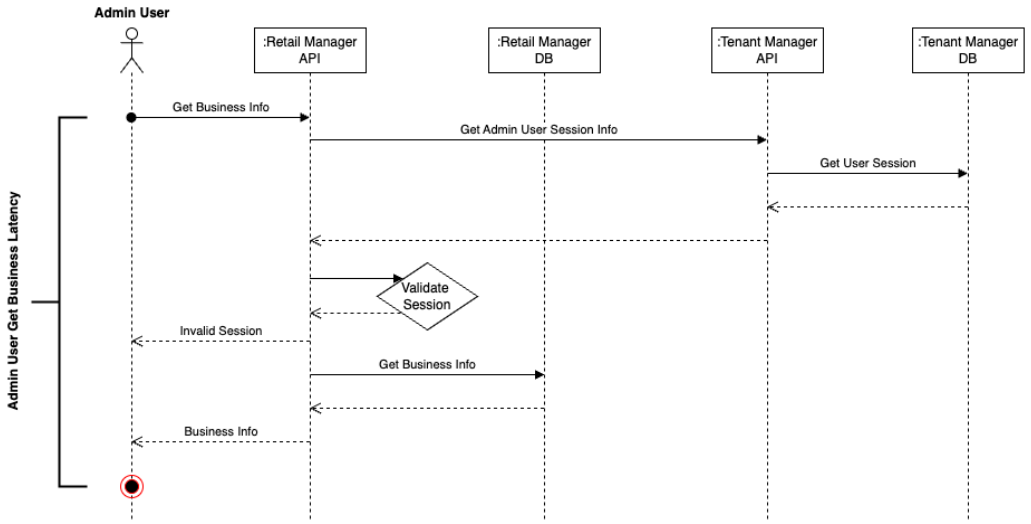


Fig. 7.16: Admin Get Business Sequence Diagram

7.2.4 Create Business User

The *Create Business User* test case accepts user information as input parameters and is primarily used to support the testing of other test cases. It helps measure system performance through latency and ensures compliance with data residency laws by storing user data in the appropriate Retail Manager database. This is a somewhat rare operation for business admin users. The sequence diagram in Fig. 7.17 shows the process of creating a business user.

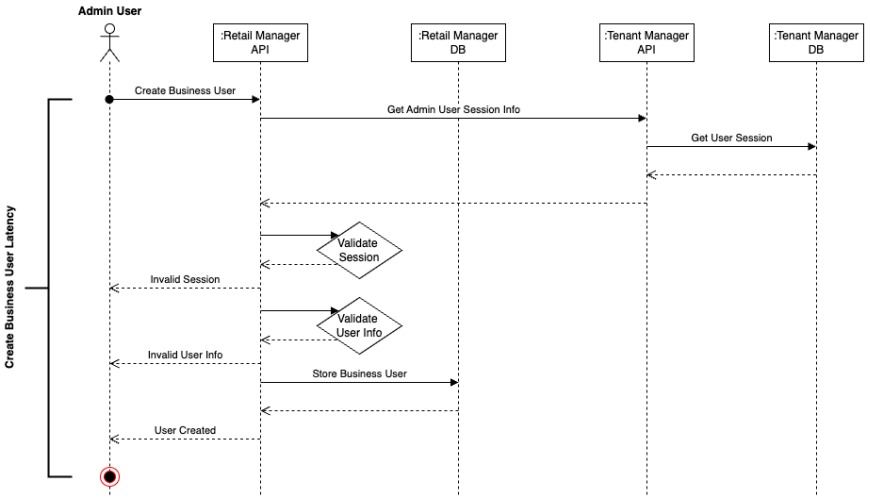


Fig. 7.17: Create Business User Sequence Diagram

7.2.5 Business User Login

This test case has two main actions. First, it retrieves the Application Plane details of a business using the business alias. Then, based on the returned business ID, along with the user's email and password, it sends login credentials to the Application Plane endpoint obtained from the Application Plane details. This test case plays a key role in validating tenant routing and ensuring adherence to data residency laws. Latency is also measured to evaluate performance. This is a somewhat rare occurrence in the system. The sequence diagram in Fig. 7.18 illustrates the business user login process.

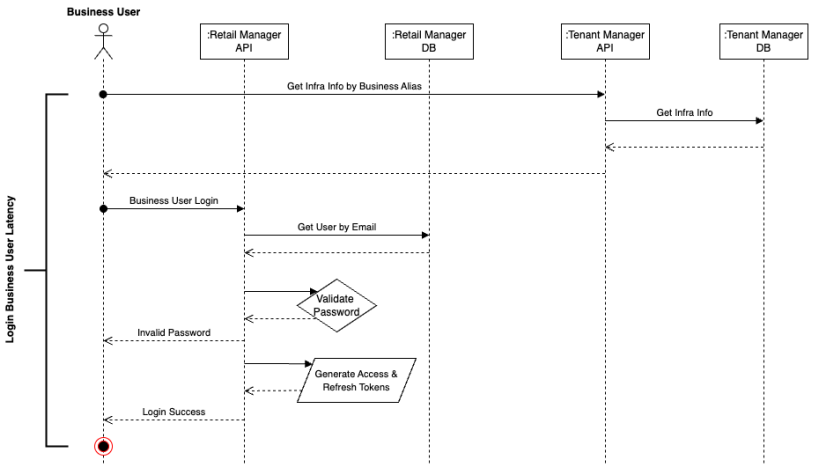


Fig. 7.18: Business User Login Sequence Diagram

7.2.6 Business User Get Business

The *Business User Get Business* test case uses the access token returned from the business user login. It is one of the most frequent operations in our system. The latency performance of this test case is critical for assessing the overall system performance. It interacts only with the Application Plane component. The sequence diagram in Fig. 7.19 illustrates this test case.

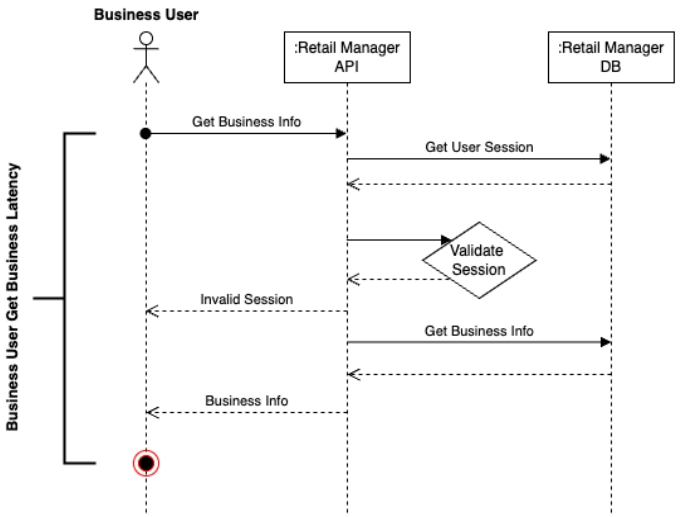


Fig. 7.19: Business User Get Business Sequence Diagram

7.3 Test Dataset

A Golang script was used to generate simulated test data for executing the test cases described in the previous section. As defined in the experimental setup, test datasets were generated for 10 countries, with 500 records per country, resulting in a total of 5,000 records. Each dataset includes simulated business and user information.

The script accepts a country code as an input parameter and generates the corresponding dataset while ensuring the uniqueness of key values such as business aliases and user email addresses. For each country code, the script produces a separate CSV file containing the generated data. The Fig. 7.20 shows a screenshot of the data generation process.

```
uzama.zaid $ go run main.go -code USA
✅ CSV file 'businesses_USA.csv' created successfully with 500 entries!
uzama.zaid $ go run main.go -code IND
✅ CSV file 'businesses_IND.csv' created successfully with 500 entries!
uzama.zaid $ go run main.go -code MY
✅ CSV file 'businesses_MY.csv' created successfully with 500 entries!
uzama.zaid $ go run main.go -code ID
✅ CSV file 'businesses_ID.csv' created successfully with 500 entries!
uzama.zaid $ go run main.go -code ES
✅ CSV file 'businesses_ES.csv' created successfully with 500 entries!
uzama.zaid $ go run main.go -code UAE
✅ CSV file 'businesses_UAE.csv' created successfully with 500 entries!
uzama.zaid $ go run main.go -code CA
✅ CSV file 'businesses_CA.csv' created successfully with 500 entries!
uzama.zaid $ go run main.go -code FR
✅ CSV file 'businesses_FR.csv' created successfully with 500 entries!
uzama.zaid $ go run main.go -code AUS
✅ CSV file 'businesses_AUS.csv' created successfully with 500 entries!
uzama.zaid $ go run main.go -code LK
✅ CSV file 'businesses_LK.csv' created successfully with 500 entries!
```

Fig. 7.20: Screenshot of Test Dataset Generation

7.4 Testing Setup

After setting up the experimental architectures, defining the test cases, and preparing the datasets, the next challenge was executing these test cases using the generated data. A key difficulty was automating the test cases due to their interdependencies, some test cases require the output of previous ones as input. To handle this a Golang script has been used to automate the test execution. The script takes a country code as input and runs the test cases in the following sequence:

- Read Dataset - Load 500 records from the CSV file corresponding to the provided country code.
- Business Register - Run the business registration test sequentially for each dataset entry, recording the latency for each request.
- Admin Login - Once registration is complete, perform admin login for each business and measure latency. The responses are stored in a structured format for use in subsequent test cases.
- Admin Get Business - Using the access token and Application Plane endpoint returned from the admin login, run the admin get business test and record latency.
- Create Business User - Using the same token and endpoint, along with user data from the dataset, run the create business user test case and measure latency.
- Business User Login - Execute the business user login test using business (alias) and user information from the dataset. Latency is recorded, and responses are stored for the next test case.
- Business User Get Business - Using the access token and endpoint from the previous step, run the business user get business test and record latency.

After all test cases are executed, the script calculates the average, p90, and p95 latencies for each test case. Latency results per dataset and the summary statistics are saved in separate CSV files for each test case. These CSVs are then uploaded to an AWS S3 bucket for storage and analysis.

To simulate a realistic, country specific environment, tests are executed on behalf of each country using AWS EC2 instances deployed in regions that closely match the selected countries. This regional deployment ensures accurate latency measurements.

The Table. 7.8 shows the mapping between countries and their corresponding EC2 regions.

Table. 7.8: Country and EC2 Region Mapping

Country	Code	Region
America	USA	us-west-1
India	IND	ap-south-1
Malaysia	MY	ap-southeast-5
Indonesia	ID	ap-southeast-3
Spain	ES	eu-south-2
UAE	UAE	me-central-1
Canada	CA	ca-central-1
France	FR	eu-west-3
Australia	AUS	ap-southeast-4
Sri Lanka	LK	local laptop

Since setting up EC2 instances and running the test scripts is a repetitive (for each test case and experimental architecture) and time-consuming task, we used Terraform to automate the entire process. This automation includes provisioning EC2 instances in the appropriate AWS regions, uploading the test scripts to each instance, executing the test scripts with the required parameters, downloading the resulting CSV files from S3 to the local machine and destroying all AWS resources created specifically for the testing setup once execution is complete. This approach significantly reduced manual effort and ensured consistent, repeatable test environments for each country. The Fig. 7.21 presents a high-level architecture of the automated testing setup.

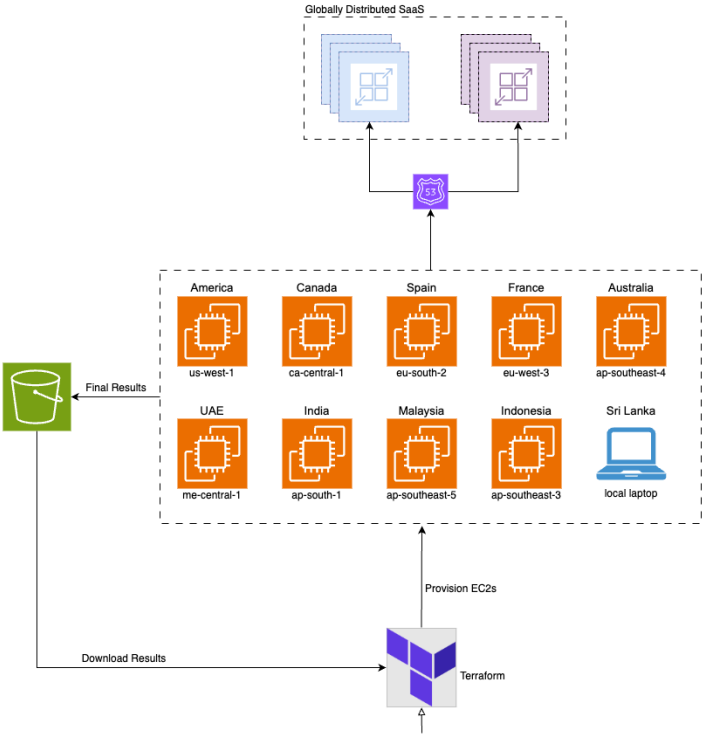


Fig. 7.21: High Level Diagram of Automated Testing Setup

7.5 Final Results

This section presents a comprehensive analysis of the results obtained from the experimentation conducted on each architectural setup. For every architecture, we begin by examining key metrics from system components and databases. This helps us understand how different parts of the system behaved during the automated test runs. We then analyze the latency for each test case in detail. For every test case, we calculate and evaluate summary statistics, including the average latency, the 90th percentile (P90), and the 95th percentile (P95). These metrics provide insight into system responsiveness and performance under various conditions.

Following the individual analysis, we compare the summary results across all test cases between the experimental architectures. This comparative study allows us to identify which architectural setup performs most efficiently under the defined scenarios. Finally, we discuss the findings and draw conclusions based on the observed data. Recommendations are provided to guide future improvements and inform architectural decisions going forward.

7.5.1 1CP x 1AP

7.5.1.1 Component Metrics

The diagrams in Fig. 7.22 shows the metrics and resource utilization for the Control Plane and Application Plane components after running automated tests. They include CPU and memory usage, the number of requests handled, and the latency of each processed request.

Fig. 7.22: Application Metrics (1CPx1AP)

The Fig. 7.23 shows the metrics for the Tenant Manager DB and Retail Manager DB. It includes the number of read and write queries, along with their processing latency.

Fig. 7.23: Database Metrics (1CPx1AP)

7.5.1.2 Data Residency Law

Since only one Application Plane is deployed in a single region, and all countries are mapped to this single Application Plane, data residency requirements cannot be met. Business data from all countries would have to be stored in one region, violating regional data residency laws. This clearly demonstrates that the 1CP×1AP architecture is not suitable when compliance with data residency regulations is required.

7.5.1.3 Operational Efficiency

In the 1CPx1AP architecture, only one Control Plane and one Application Plane are required, resulting in minimal operational overhead. One key aspect of operational efficiency considered is the time taken to deploy or roll out an Application Plane instance. Since the architecture relies solely on serverless components, resource provisioning is fully automated using Terraform and GitLab CI.

Even in cases where manual intervention is needed, the time and effort required remain low due to the minimal number of components involved. Additionally, tenant

management is streamlined, as all tenants are served through a single Application Plane, offering a unified experience with reduced complexity. Therefore, the 1CPx1AP architecture provides strong operational efficiency by minimizing the time required for Application Plane deployment or rollout.

7.5.1.4 Final Latencies

The diagrams in Fig. 7.24 shows latency comparisons for each of the six defined test cases. Each graph represents one test case and contains 10 lines, with each line corresponding to a different country. The x-axis represents 500 sequential data points (requests), and the y-axis indicates latency, allowing for a clear visualization of performance trends across countries.

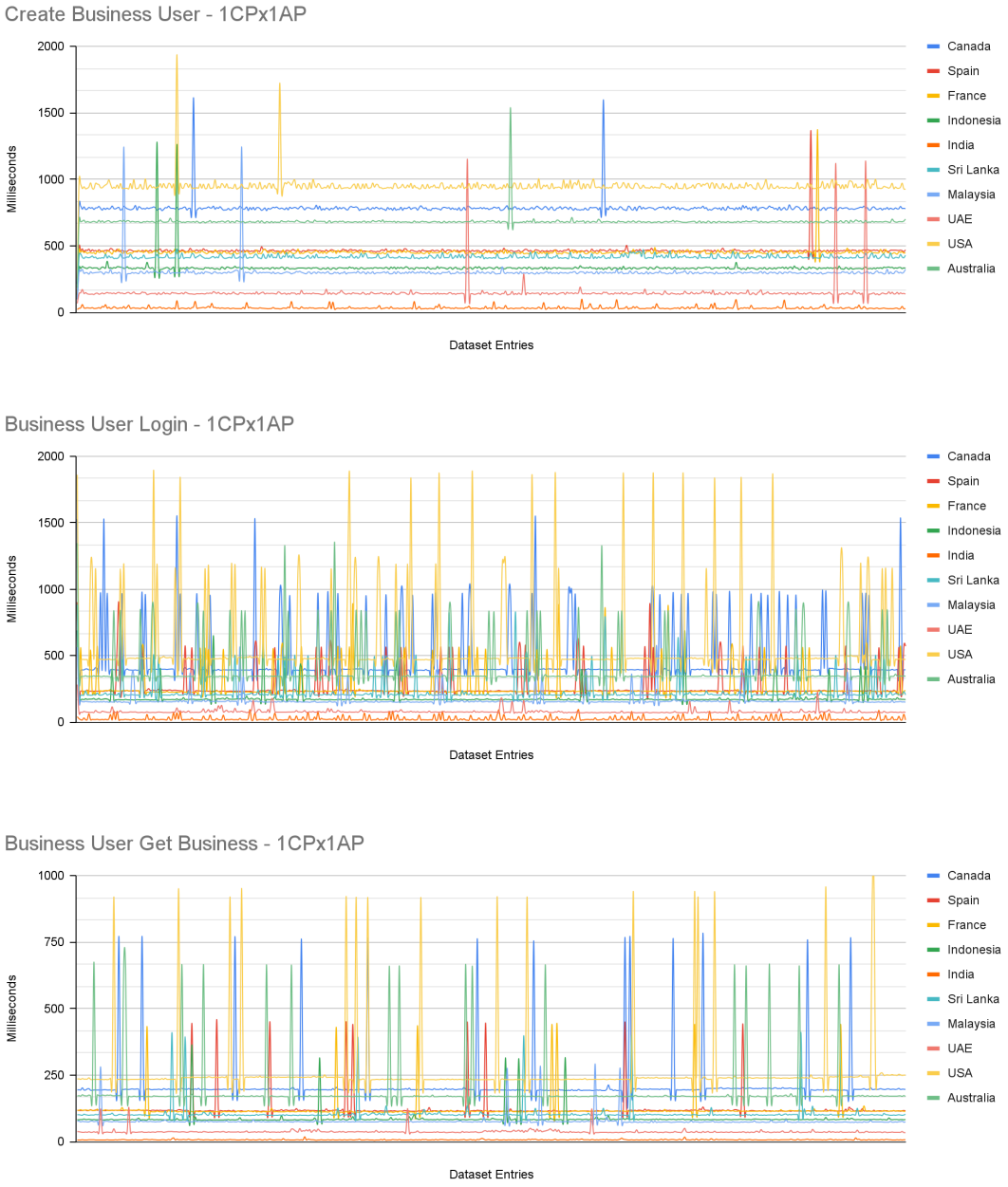


Fig. 7.24: Final Latencies Analysis (1CPx1AP)

The diagrams in Fig. 7.24 clearly shows the latency differences across countries, primarily influenced by the geographical distance between each country and the regions where the Control Plane and Application Plane are deployed. Since the SaaS is hosted in India, latency is significantly lower for India and neighboring countries across all test cases.

In contrast, countries farther from India experience higher latency, especially in test cases that require communication between both the Control Plane and Application Plane, such as Business Register, Create Business User, and Business User Login. An exception is the Admin Get Business test case, which performs relatively well because it involves only read operations from DynamoDBs. Similarly, the Admin

Login and Business User Get Business test cases also show better performance, as they involve communication with either the Control Plane or Application Plane.

7.5.1.5 Final Summary

To better analyze the latency results, a summary was created using key statistical metrics such as average, p90, and p95 latencies. The diagrams in Fig. 7.25 presents a bar chart for each test case, where each chart compares latency across all countries using these summary statistics. This provides a clearer and more comprehensive view of performance variations by region.

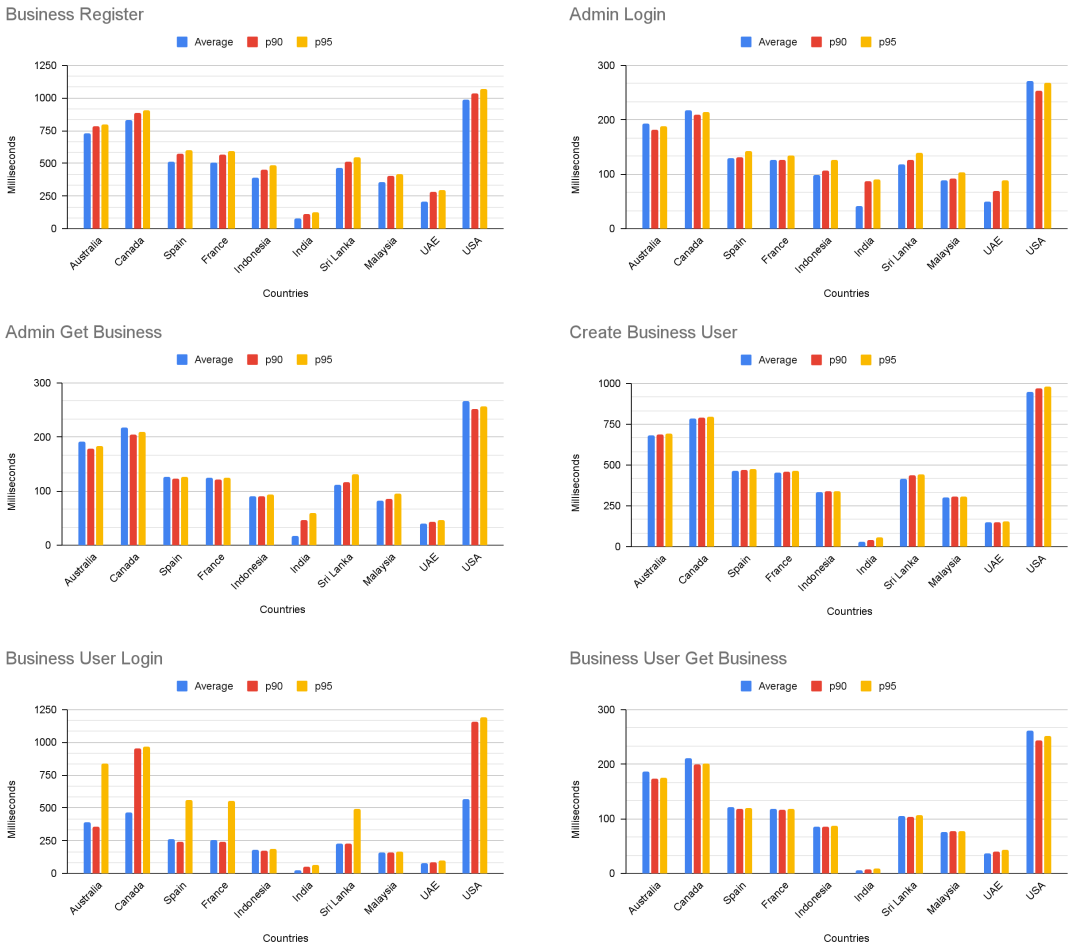


Fig. 7.25: Final Summary Latencies Analysis (1CPx1AP)

From the above bar charts in Fig. 7.25, a clear insight emerges that India and its neighboring countries consistently show better performance across all test cases. One particularly notable observation is in the Business User Login test case. For countries farther from India including Sri Lanka (likely due to network variability), the p95 latency is significantly higher compared to the average and p90 values. For more distant countries like the USA and Canada, even the p90 values are noticeably elevated relative to the average. This suggests that multiple round trips to the Control

Plane and the Application Plane from these remote countries introduce added latency, making this test case potentially slower for many users in those countries.

In contrast, the other test cases show relatively similar values across average, p90, and p95. This is likely because they involve a single request sent to the India region from the origin, with all subsequent operations occurring locally within the region, reducing overall latency variance.

7.5.2 1CP x nAP

7.5.2.1 Component Metrics

The diagrams in Fig. 7.26 presents the performance metrics and resources utilizations of an architecture with a single Control Plane and multiple Application Plane components, captured after executing automated tests. These metrics include CPU and memory usage, request throughput, and request latency. To provide clearer insights, the metrics from multiple Application Planes are aggregated and visualized together in the same graph.

Fig. 7.26: Application Metrics (1CPxnAP)

The Fig. 7.27 displays the performance metrics for the Tenant Manager Database (in Control Plane) and the Retail Manager Databases across all Application Planes. It highlights the number of read and write queries, along with their corresponding processing latencies. To enhance clarity and provide a consolidated view, the metrics

from multiple Retail Manager Databases are aggregated and presented in a single graph.

Fig. 7.27: Database Metrics (1CPxnAP)

7.5.2.2 Data Residency Law

In the 1CP×nAP architecture, the Application Plane is deployed across multiple regions to comply with data residency laws. Each country is mapped to a specific Application Plane based on its data residency requirements, ensuring that business data is stored in the appropriate regional location as configured by the Control Plane. This setup effectively satisfies data residency regulations, clearly demonstrating that the 1CPxnAP architecture is the most suitable choice when compliance is a critical requirement.

This can be manually verified by checking that business data is stored in the correct regional DynamoDB databases during the Business Register test case. The images in Fig. 7.28 shows screenshots of the databases in eu-west-1 (Ireland), ap-southeast-2 (Australia), and ap-southeast-1 (Singapore). According to our configuration, each of these databases should contain 1,000, 500, and 1,500 records respectively.

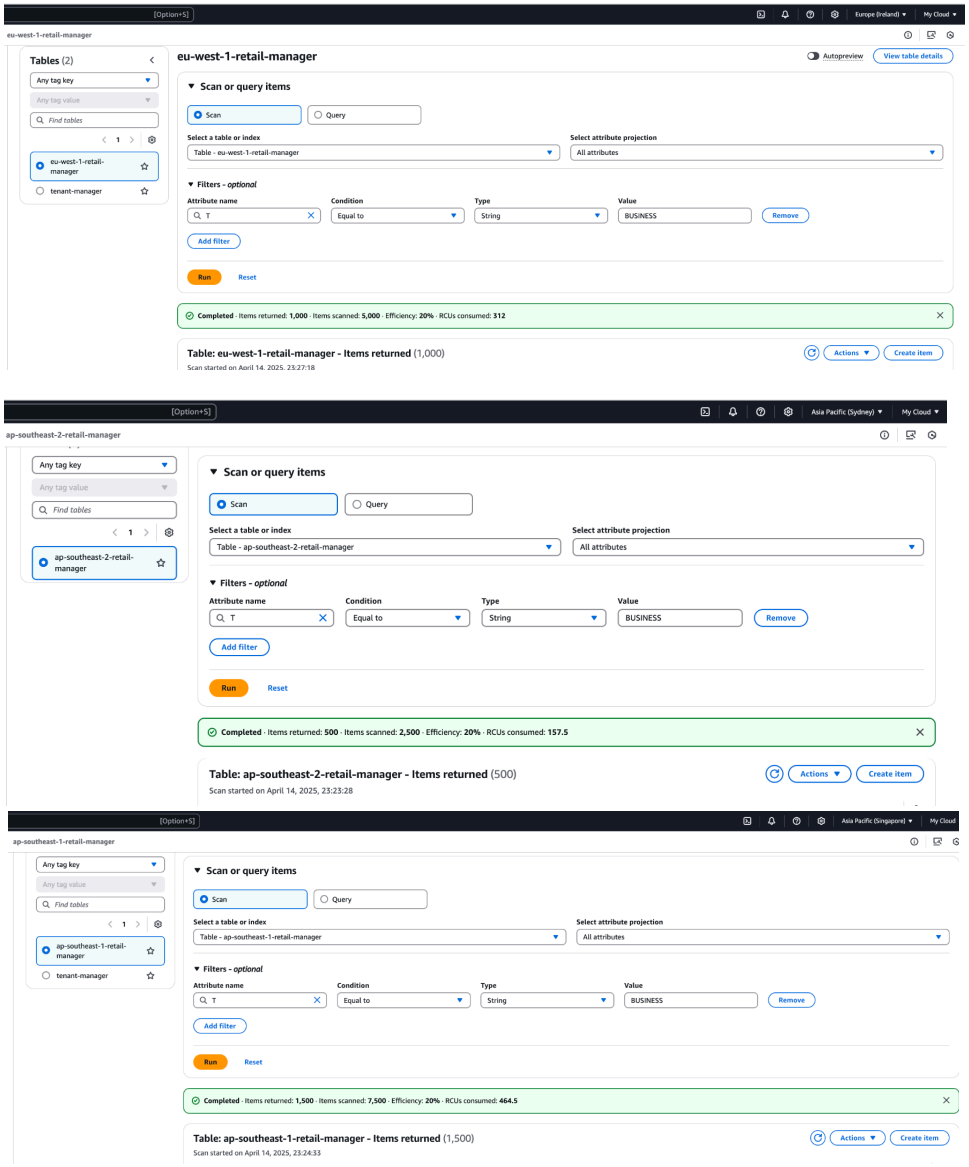


Fig. 7.28: Screenshot of Databases (1CPxnAP)

Based on the screenshots in Fig. 7.28, it can be confirmed that the Business Registration test case functions correctly according to the region-country mapping configured for data residency compliance.

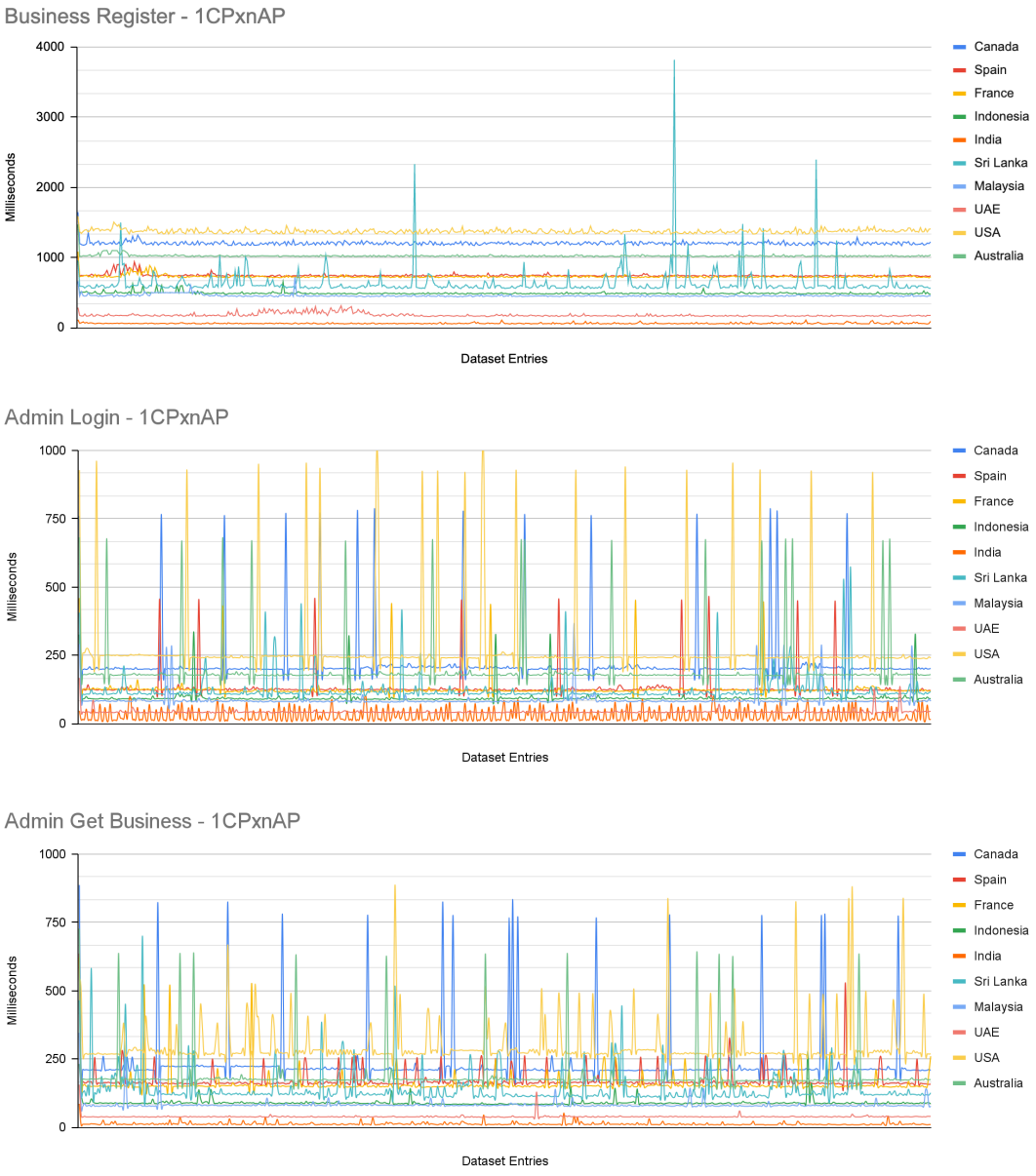
7.5.2.3 Operational Efficiency

In the 1CPxnAP architecture, a single Control Plane manages multiple Application Planes. Operational efficiency measured via the time taken to roll out updates to every Application Plane instance. While this setup introduces some operational overhead due to the need to provision multiple Application Planes, implementation minimizes this through full automation using tools like Terraform and GitLab CI. Since the architecture is built entirely with serverless components, the overhead remains significantly reduced, even with multiple Application Planes.

However, in scenarios where manual provisioning is required, operational efficiency may increase due to the need to provision each component individually. That said, this primarily affects the early setup stage. Despite this, automation streamlines the version rollout process for Application Plane instances, reducing operational burden and ensuring a consistent experience across all tenants. Additionally, having multiple Application Planes enables flexibility in rolling out features incrementally to specific regions. Overall, the 1CPxnAP architecture maintains a good balance between scalability and operational efficiency, making it well suited for globally distributed SaaS applications.

7.5.2.4 Final Latencies

The diagrams in Fig. 7.29 illustrates latency comparisons for the six defined test cases. Each graph represents a single test case and includes 10 lines, with each line corresponding to a different country.



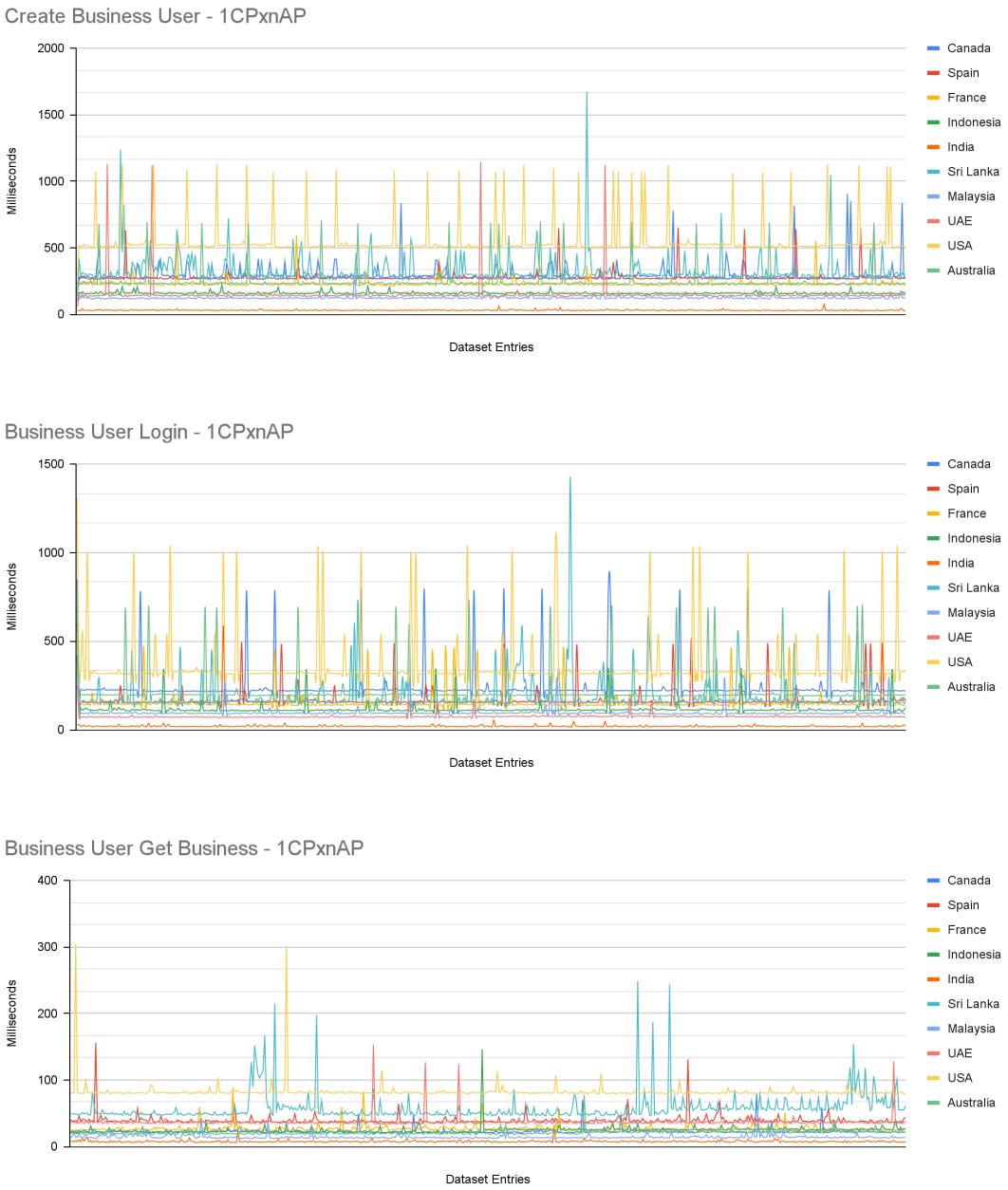


Fig. 7.29: Final Latencies Analysis (1CPxnAP)

The Business User Get Business test case demonstrates clear performance improvements across all countries, as it communicates only with the Application Planes which are deployed close to each country. In contrast, other test cases show slightly reduced performance due to the need for interaction with the Control Plane, which is deployed in the India region.

Countries like India and its neighbors perform consistently well across all test cases, due to their proximity to the Control Plane. The Business Register test case shows the poorest performance for countries far from India, as it involves multiple write operations to different databases in two different regions. Other test cases perform comparatively better, since Application Planes are regionally deployed which provides low latency.

7.5.2.5 Final Summary

The Fig. 7.30 displays a bar chart for each test case, with each chart comparing latency across all countries using summary statistics such as average, p90, and p95.

Fig. 7.30: Final Summary Latencies Analysis (1CPxnAP)

The Business Register test case shows noticeable performance degradation for countries other than India and the UAE. This is due to the need for multiple write operations across two databases located in different regions. Since both India and the UAE are mapped to the Indian region for their Application Plane, and the Control Plane is also located in India, these two countries experience significantly better performance in this test case. Notably, they also perform well across all other test cases.

The Business User Get Business test case demonstrates strong performance across all countries, as it only involves the Application Planes, which are deployed close to each country. However, Sri Lanka and the USA show slightly lower performance in comparison, indicating that the Application Planes are still not close enough. This trend is reflected across most test cases, where these two countries consistently report higher average, p90, and p95 latencies.

Overall, test cases that involve Application Plane interactions generally show good performance across all regions (except in certain cases for the USA and Sri Lanka) due to the proximity of the Application Planes to each respective country.

7.5.3 nCP x nAP

7.5.3.1 Component Metrics

The images in Fig. 7.31 displays the aggregated metrics and resource utilization of the Control Planes and Application Planes.

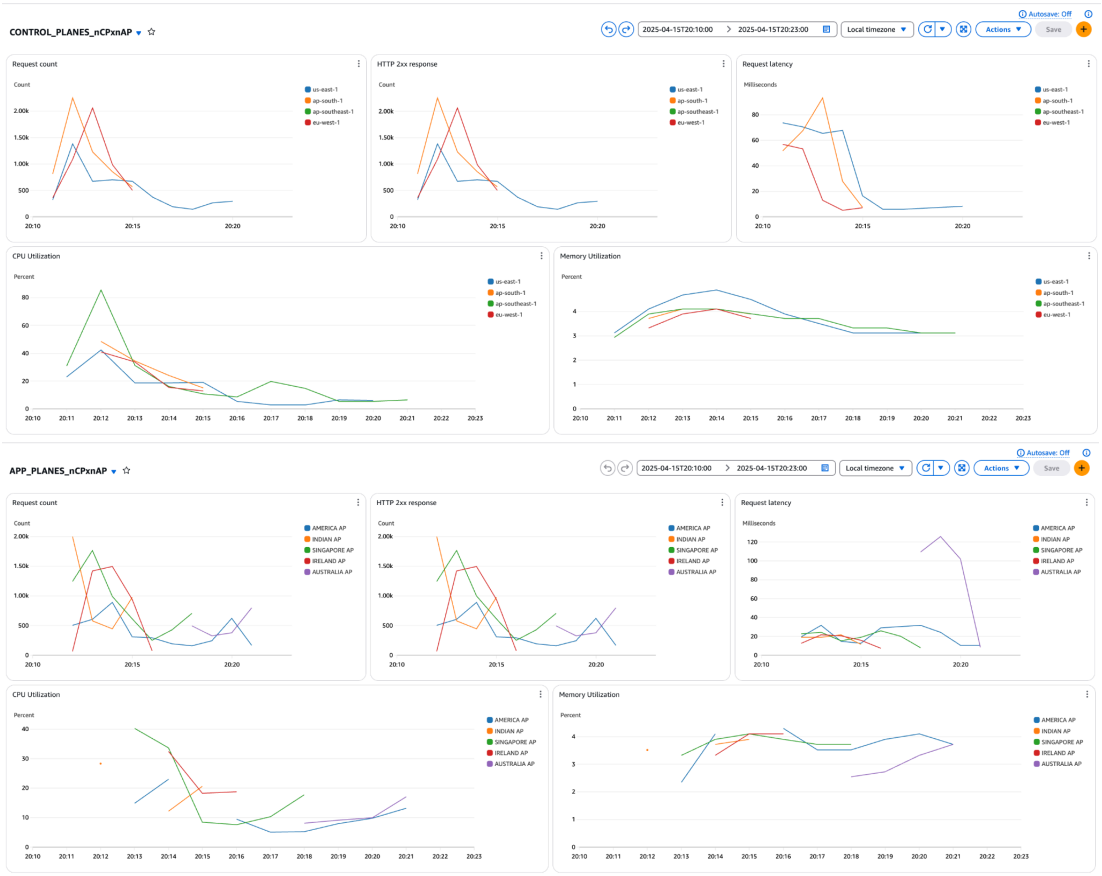
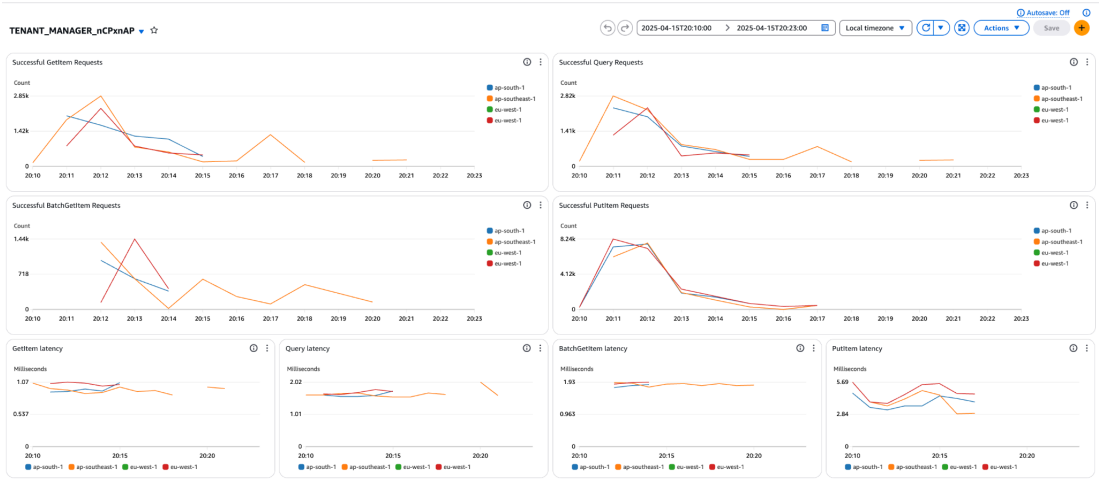


Fig. 7.31: Application Metrics (nCPxnAP)

Fig. 7.32 shows aggregated metrics for both Tenant Managers and Retail Managers.



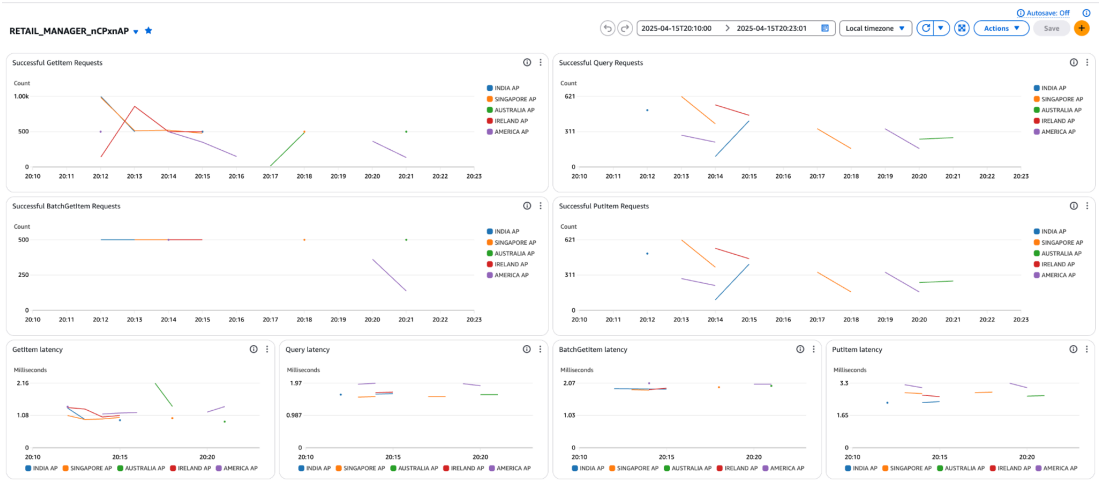


Fig. 7.32: Database Metrics (nCPxnAP)

7.5.3.2 Data Residency Law

The nCP×nAP architecture retains the same multi-region deployment of Application Planes as the 1CP×nAP setup to ensure compliance with data residency laws. However, it introduces additional Control Plane components to reduce latency and improve performance for certain test cases. Despite these additions, data residency requirements remain fully preserved through the configured mappings between countries, regions, and Application Planes. Verification of this data residency compliance can be carried out in the same manner as demonstrated for the 1CP×nAP architecture.

7.5.3.3 Operational Efficiency

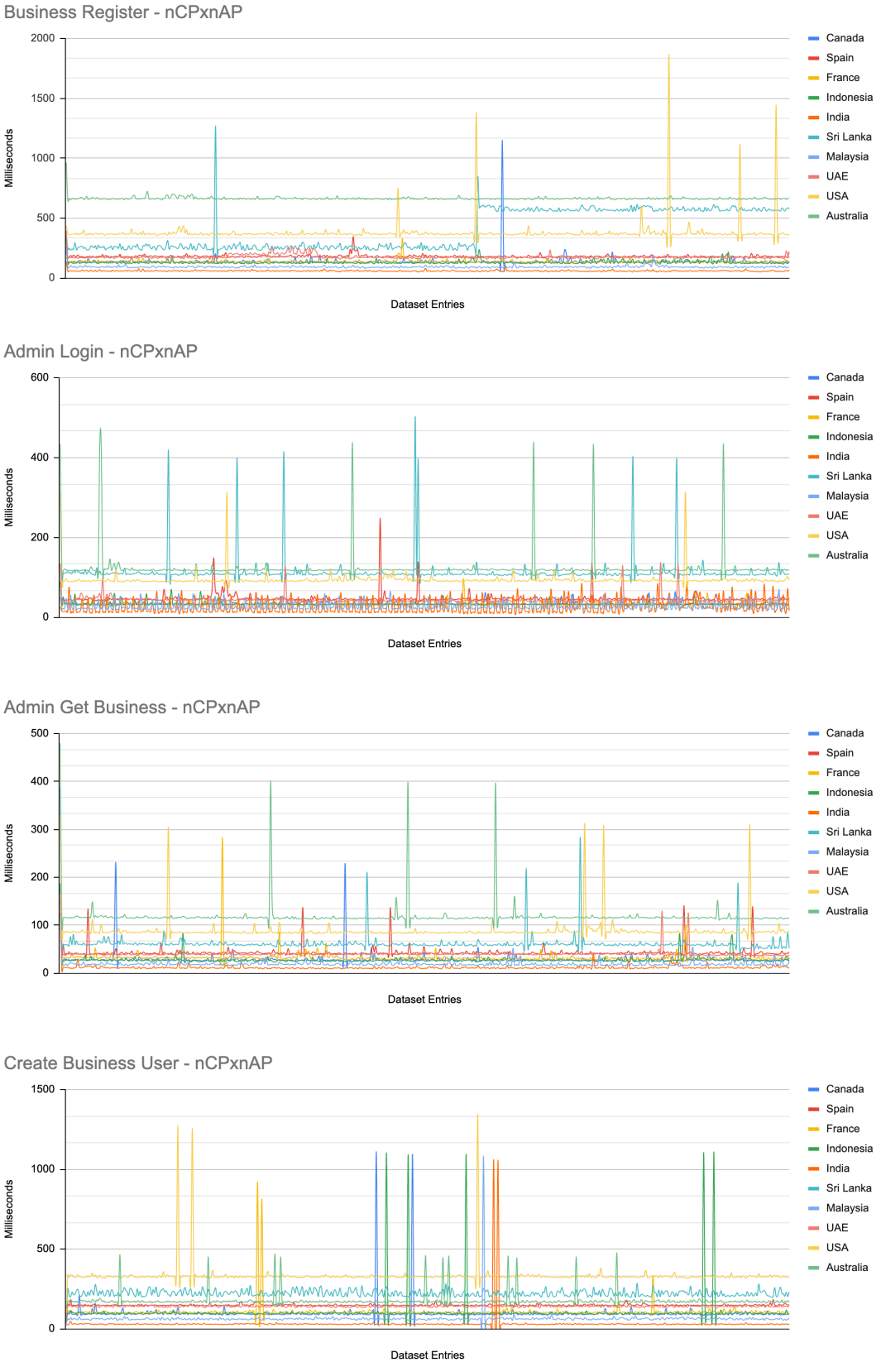
In the nCPxnAP architecture, multiple Control Planes manage multiple Application Planes distributed across different regions. While this setup introduces more infrastructure components due to distribution of Control Plane, operational efficiency is primarily measured by the time taken to roll out updates to every Application Plane instance, not by the setup effort for Control Plane components.

Like the 1CPxnAP architecture, nCPxnAP also benefits from automation using tools such as Terraform and GitLab CI, which streamline the rollout of new versions and features to Application Plane instances. Although the presence of multiple Application Planes may require more coordination, automation significantly reduces the operational overhead. This architecture also supports flexible and region specific rollouts, helping maintain a consistent and unified experience across tenants globally.

Overall, despite its fully distributed nature, the nCPxnAP architecture preserves operational efficiency through automation and controlled deployment processes focused on Application Plane management.

7.5.3.4 Final Latencies

The diagrams in Fig. 7.33 illustrates latency comparisons across six defined test cases and ten countries for the nCP×nAP architecture.



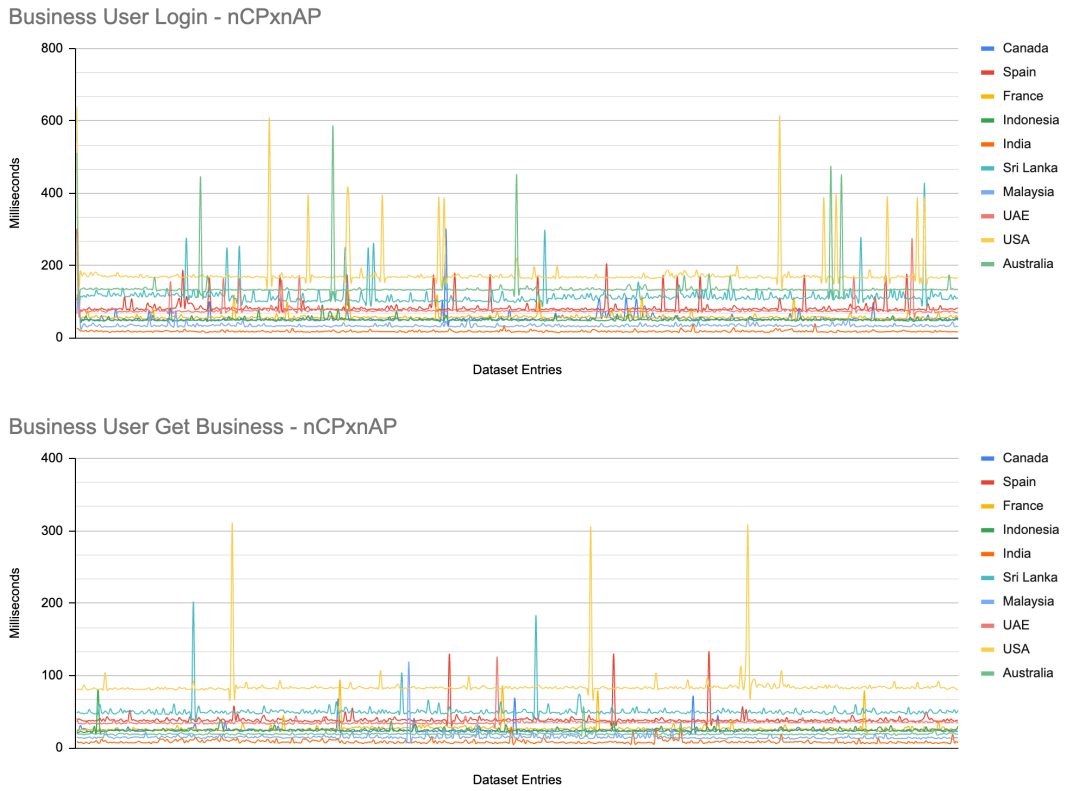
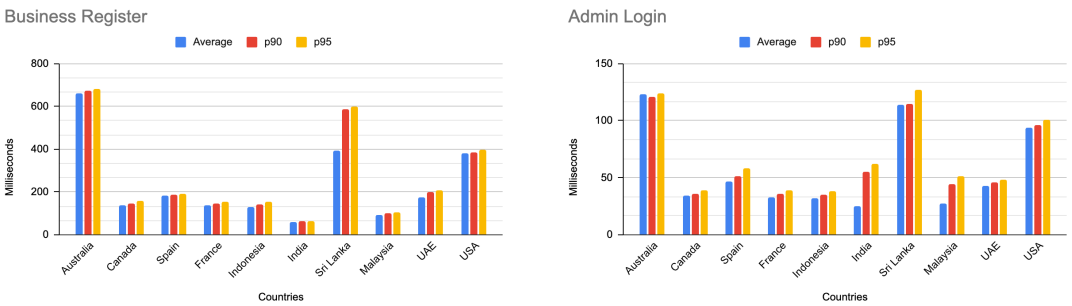


Fig. 7.33: Final Latencies Analysis (nCPxnAP)

Based on the images in Fig. 7.33, an overall view shows that every country performs well across all test cases. A deeper look at the Business User Get Business test case reveals that each country is interacting with its designated Application Plane, resulting in strong performance for all countries. However, on closer inspection, countries like the USA and Sri Lanka show slightly lower performance, likely due to the greater distance from their assigned Application Plane. Similarly, in the Business Register test case, Australia experiences a slight performance drop, likely due to its nearest Control Plane being located in Singapore. Despite these minor variations, the nCP×nAP architecture delivers consistently strong performance overall.

7.5.3.5 Final Summary

The images in Fig. 7.34 presents a bar chart for each test case, comparing summary statistics such as average, p90, and p95, across ten countries.



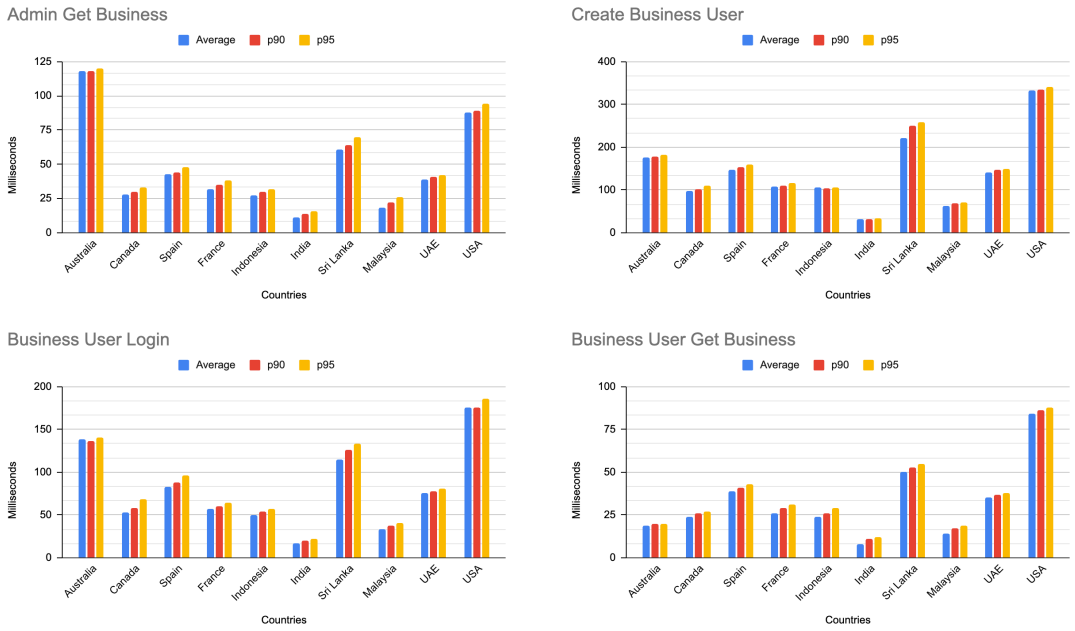


Fig. 7.34: Final Summary Latencies Analysis (nCPxnAP)

The bar charts in Fig. 7.34 provides a clear view of each country's performance across all test cases, making it easy to identify patterns and differences. Overall, all countries perform well, however, Australia, Sri Lanka, and USA show comparatively slightly higher latency in most test cases. Sri Lanka and the USA exhibit slightly lower performance across all test cases, likely due to the greater distance from both their nearest Control Plane and assigned Application Plane. Australia, on the other hand, performs well in the Business User Get Business test case, which interacts only with the Application Plane and benefits from closer proximity to its Application Plane. In other test cases, Australia's performance drops slightly due to the relative distance from its Control Plane, which is based in Singapore. While these differences are noticeable when compared to other countries, the overall performance remains higher across all countries.

7.5.4 Comparison

In this section, we will compare each configured architecture and evaluate key parameters such as performance latency, scalability, data residency laws, and operational efficiency. Additionally, we will assess cost and availability based on the configurations and the results obtained from automated tests.

7.5.4.1 Performance Latencies

Based on the summarized latencies from automated testing of each test case, the following bar charts in Fig. 7.35 to Fig. 7.40, illustrates the performance comparison of each architecture. For clearer insights, a separate bar chart has been created for each summary metrics such as average latency, P90, and P95, under each test case.

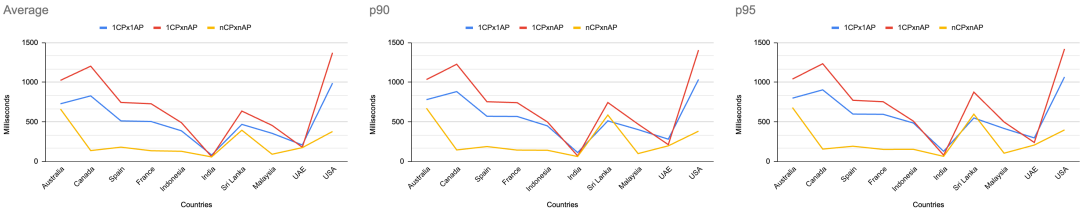


Fig. 7.35: Business Register - Summary Latencies Comparison

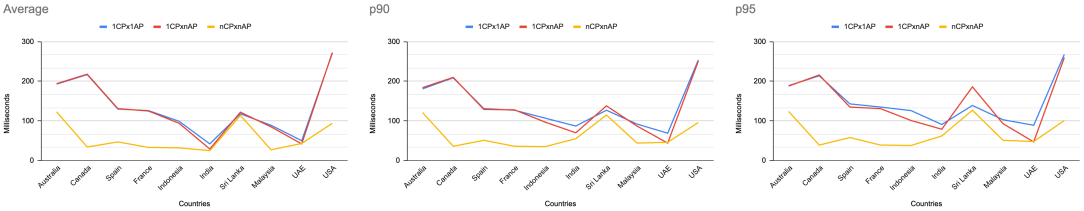


Fig. 7.36: Admin Login - Summary Latencies Comparison

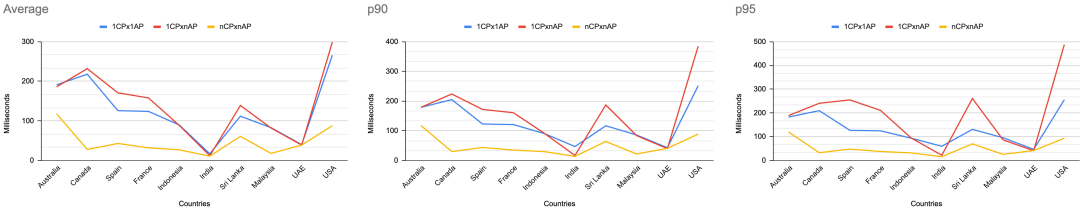


Fig. 7.37: Admin Get Business - Summary Latencies Comparison

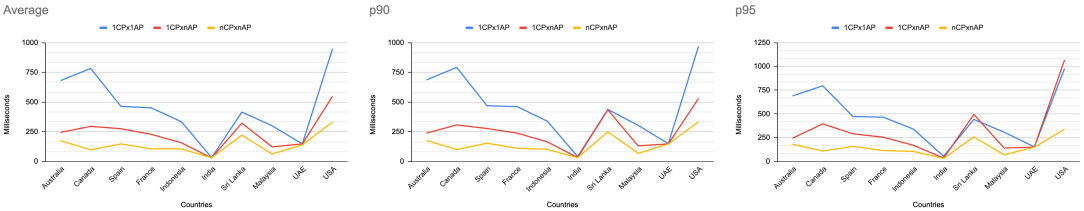


Fig. 7.38: Create Business User - Summary Latencies Comparison

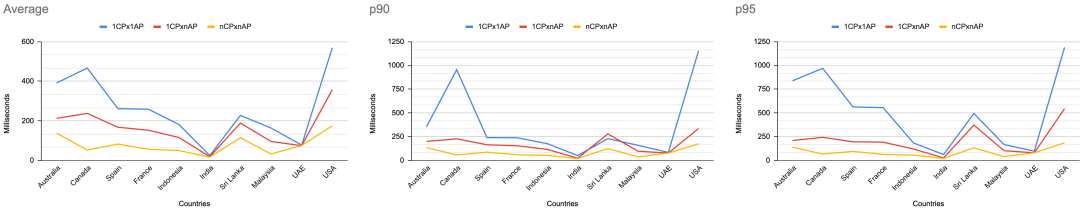


Fig. 7.39: Business User Login - Summary Latencies Comparison

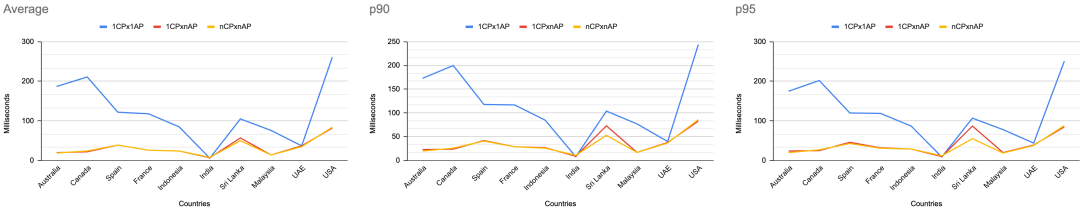


Fig. 7.40: Business User Get Business - Summary Latencies Comparison

Based on the bar chart in Fig. 7.35 to 7.40, the nCPxnAP architecture clearly outperforms the other two architectures, across almost all countries and test cases.

Determining the second best architecture requires a closer look, as performance varies depending on the test case. The 1CPx1AP architecture performs better for business admin user scenarios, while 1CPxnAP shows better performance for business user scenarios. Since business user interactions are more frequent in the proposed architecture, we can conclude that 1CPxnAP is the second best performer. In comparison, 1CPx1AP consistently delivers lower performance.

In the same diagrams (Fig. 7.35 to 7.40), we can observe the patterns across average, P90, and P95 latency graphs for each test case. When these latency summaries show similar trends, it indicates consistency in latency results, which is an important performance indicator. Upon close examination, nCPxnAP demonstrates strong consistency across all test cases. 1CPxnAP also shows consistent performance, based on the alignment of its latency patterns. However, for 1CPx1AP, particularly in the Business User Login test case, there is a noticeable increase in P90 and P95 latency for certain countries. This highlights that 1CPx1AP has some limitations in handling different environments reliably.

Therefore, the performance ranking is as follows,

nCPxnAP >> 1CPxnAP > 1CPx1AP

7.5.4.2 Data Residency Law

As discussed in the results section for each architecture about data residency law, both nCPxnAP and 1CPxnAP comply with data residency requirements due to their configurations, which align countries, regions, and Application Planes mapping based on assumed data residency laws. On the other hand, the 1CPx1AP architecture uses a single Application Plane to store data of businesses from worldwide in one region, thereby violating the data residency laws.

Therefore, the ranking in terms of compliance with data residency laws is as follows,

nCPxnAP = 1CPxnAP >> 1CPx1AP

7.5.4.3 Operational Efficiency

As we consider operational efficiency in this context, it is primarily measured by the time taken to roll out updates to every Application Plane instance. Based on this measure, the 1CPx1AP architecture offers the highest efficiency due to a single Application Plane needing to be updated, resulting in low deployment time and overhead. On the other hand, 1CPxnAP and nCPxnAP involve multiple Application Planes, which could introduce more operational overheads. However, by leveraging automation tools such as Terraform and GitLab CI, deployments to multiple Application Planes can be executed in parallel which significantly reduces the time required and closes the gap in operational efficiency.

Therefore, with automation in place, all three architectures demonstrate similar operational efficiency, regardless of the number of components involved.

The ranking for operational efficiency is as follows,

$$\textbf{nCPxnAP} = \textbf{1CPxnAP} \cong \textbf{1CPx1AP}$$

7.5.4.4 Global Scalability

Global scalability refers to the ability of SaaS applications to expand and operate across multiple regions, delivering optimal performance to global users while maintaining compliance with data residency laws and ensuring operational efficiency. Based on the configurations, latency results, and earlier comparisons, we can conclude that the nCPxnAP architecture offers the highest global reach by distributing both application and Control Planes while effectively balancing other key factors. The 1CPxnAP architecture ranks slightly lower, as it distributes only the Application Plane. While it still balances most factors, its single Control Plane causes performance limitations in certain test cases. On the other hand, the 1CPx1AP architecture is clearly not suitable for global scalability, as it neither balances critical factors nor delivers competitive performance.

Therefore, the ranking for global scalability is as follows,

$$\textbf{nCPxnAP} > \textbf{1CPxnAP} >> \textbf{1CPx1AP}$$

7.5.4.5 Cost

To compare the cost of each architecture, we assume a unit cost for provisioning each component. Since both the Application Plane and Control Plane use similar resources, we can consider their costs to be approximately equal. Under this assumption, nCPxnAP and 1CPxnAP require n units of cost based on the number of regions, based on the configurations both architectures require totals of 9 units and 6 units respectively. In contrast, 1CPx1AP requires only 2 units. This demonstrates that the overall cost increases linearly with the number of control or Application Planes deployed.

Therefore, the cost ranking is as follows,

$$\textbf{nCPxnAP} > \textbf{1CPxnAP} >> \textbf{1CPx1AP}$$

7.5.4.6 Availability

Availability refers to an architecture's ability to remain responsive and resilient, particularly by minimizing single points of failure. At a high level, the 1CPx1AP architecture offers the lowest availability, as it relies on a single Control Plane and a single Application Plane, meaning that the failure of either component results in a complete system outage.

The 1CPxnAP architecture improves slightly on this, as it distributes the Application Plane across regions. While a single Application Plane failure would only affect a subset of users or businesses, the presence of a single Control Plane still represents a significant risk for a single point of failure. The nCPxnAP architecture offers the highest availability by distributing both Control Planes and Application Planes across regions. Although there remains a small risk of localized outages due to individual Application Plane failures, the overall system is far more resilient compared to the other two architectures. Therefore, the availability ranking is as follows,

nCPxnAP >> 1CPxnAP >= 1CPx1AP

7.5.4.7 Summary

Table 7.9 presents a comparison of the three architectures based on the factors discussed above

Table. 7.9: Architecture Comparison Summary

Parameters	1CPx1AP	1CPxnAP	nCPxnAP
Performance	LOW	MEDIUM	HIGH
Data Residency Law	LOW	HIGH	HIGH
Operational Efficiency	HIGH	HIGH	HIGH
Global Scalability	LOW	MEDIUM	HIGH
Cost	LOW	MEDIUM	HIGH
Availability	LOW	MEDIUM	HIGH

7.5.5 Findings and Recommendations

The results showed that the 1CPx1AP architecture is not suitable for global scale and data residency compliance. Our assumption was validated and it shows that this setup lacks the necessary support for global scalability and data residency compliance. While the 1CPxnAP architecture improved compliance and scalability by adding more Application Planes, it still faced performance issues for some test cases due to the centralized Control Plane. Our final proposal, the nCPxnAP architecture, distributed both Control Planes and Application Planes, and the tests confirmed that it successfully supports global scalability without performance bottleneck and ensures data residency compliance, and preserve operational efficiency.

Based on these findings and costs, we recommend different architectures depending on the size and reach of the SaaS providers. Startups or providers operating in one or two countries can begin with the cost effective 1CPx1AP architecture by deploying the Application Plane in their target region to achieve compliance requirements. Medium sized businesses can adopt the 1CPxnAP model, gradually expanding their Application Planes based on their service regions and compliance needs. Large enterprises can implement the full nCPxnAP architecture, scaling both control and Application Planes to achieve global scalability, meet compliance requirements, and maintain operational efficiency. Control plane components can be strategically deployed closer to their corresponding Application Plane regions to enhance performance.

7.5.6 Limitations

In this research, we limited the scope of operational efficiency to a specific and measurable factor, the time taken to deploy or roll out new versions of the SaaS application to the Application Plane. This approach provided a practical and consistent basis for comparison across architectures. However, in real world systems, operational efficiency is influenced by a broader set of factors beyond rollout time. These include DevOps effort, automation complexity, resource utilization, error recovery mechanisms, monitoring capabilities, scalability under varying loads, and collaboration overhead during updates. Since these aspects were outside our research scope, they were not fully evaluated in this research.

Another limitation is the assumption of a monolithic architecture for the SaaS application implementation. Monolithic systems simplify deployment and versioning, as all components are packaged and deployed together. This architectural choice contributed to lower operational overhead in our experiments. However, modern SaaS platforms adopt microservices architectures, which are inherently more complex to manage. Deployments in such environments may involve rolling out multiple services independently, maintaining backward compatibility, and coordinating updates, all of which increase operational effort.

Future research should explore these dimensions in greater depth and also experiment with different application design patterns to provide a more complete picture of the trade offs involved in the proposed architecture.

CHAPTER 8

CONCLUSION

The emergence of cloud computing has revolutionized software delivery, establishing SaaS as a dominant model due to its scalability, cost efficiency, and accessibility. However, as SaaS providers expand their global reach, they encounter significant challenges in simultaneously achieving global scalability, data residency compliance, and operational efficiency. Existing solutions typically address these requirements in isolation, focusing either on performance optimization or regulatory adherence without offering a comprehensive architectural approach. This research bridges this gap by proposing and validating an innovative architecture that effectively balances these competing requirements.

Our solution features a logically centralized yet physically distributed Control Plane that manages regionally distributed Application Planes. The Control Plane handles critical functions including tenant onboarding, tenant routing, and Application Plane orchestration, ensuring tenant data remains within designated geographical boundaries. The Application Planes, deployed across multiple regions according to Control Plane configurations aligned with data residency laws, host the actual SaaS application components.

To validate our proposed architecture, we implemented three distinct configurations such as 1CPx1AP, 1CPxnAP, and our proposed nCPxnAP model. Through rigorous experimentation comparing these architectures, we obtained valuable insights into their performance, compliance, and operational characteristics. The single region 1CPx1AP architecture, while operationally simpler and cost effective, is not suitable for global deployments due to poor performance for distant users and non compliance with data residency regulations. The 1CPxnAP model maintained operational efficiency while demonstrating strong compliance capabilities and reasonable performance, even though latency issues persisted for users located far from the Control Plane region. Our fully distributed nCPxnAP architecture emerged as the optimal solution for global SaaS providers, combining operational efficiency with superior performance across all test cases while guaranteeing full compliance with data residency requirements. Although this configuration incurs higher infrastructure costs, the resulting improvements in user experience, compliance assurance, and system resilience justify the investment for large scale operations.

These findings provide clear guidance for SaaS providers at various growth stages. Startup providers operating within a single jurisdiction can begin with the simpler 1CPx1AP model. Mid-sized companies expanding regionally may adopt the 1CPxnAP approach to balance compliance and cost through strategically placed regional Application Planes. Large enterprises serving global markets will derive maximum benefit from the nCPxnAP architecture, as its superior performance,

operational efficacy, data residency compliance, and reliability outweigh the higher costs.

Future research directions offer exciting opportunities to enhance the proposed architecture. While this study provides valuable insights, it is important to note that the findings are based on specific architectural assumptions. Future work should broaden these boundaries by exploring microservice based SaaS applications and incorporating a wider range of operational efficiency factors, such as DevOps effort, resource utilization, and more. Additionally, extending support for multi cloud deployments will reduce vendor lock in and increase flexibility, addressing scenarios where public cloud availability is limited or private cloud support is required.

This thesis makes three key contributions: (1) an architectural model for a globally scalable, compliant, and operationally efficient multi-tenant SaaS; (2) practical validation through real world implementation with quantifiable results across performance, compliance, and operational efficiency; and (3) actionable guidelines helping SaaS providers select and optimize architectures based on their scale and regulatory requirements. By successfully addressing the critical requirements such as global scalability, data residency compliance, and operational efficiency, this research empowers SaaS providers to expand globally with confidence. The research findings not only advance academic research but also deliver immediate value to industry professionals navigating the complexities of modern global SaaS delivery.

REFERENCES

[1] "Data protection under GDPR - Your Europe," Your Europe, Jan. 1, 2022. [Online]. Available: https://europa.eu/youreurope/business/dealing-with-customers/data-protection/data-protection-gdpr/index_en.htm

[2] N. Cory and L. Dascoli, "How barriers to cross-border data flows are spreading globally, what they cost, and how to address them," ITIF, Dec. 4, 2021. [Online]. Available: https://itif.org/publications/2021/07/19/how-barriers-cross-border-data-flows-are-spreading-globally-what-they-cost/

[3] E. Limchayseng, "Data residency concerns for global applications," Heroku Blog, Aug. 22, 2023. [Online]. Available: https://blog.heroku.com/data-residency-concerns-global-applications

[4] "SaaS is a business model - SaaS architecture fundamentals," AWS Whitepapers, 2023. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/saas-architecture-fundamentals/saas-is-a-business-model.html

[5] "Moving to a unified experience - SaaS architecture fundamentals," AWS Whitepapers, 2023. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/saas-architecture-fundamentals/moving-to-a-unified-experience.html

[6] "Control plane vs. application plane - SaaS architecture fundamentals," AWS Whitepapers, 2023. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/saas-architecture-fundamentals/control-plane-vs-application-plane.html

[7] "Re-defining multi-tenancy - SaaS architecture fundamentals," AWS Whitepapers, 2023. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/saas-architecture-fundamentals/re-defining-multi-tenancy.html

[8] "Tenant isolation - SaaS architecture fundamentals," AWS Whitepapers, 2023. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/saas-architecture-fundamentals/tenant-isolation.html

[9] "Data partitioning - SaaS architecture fundamentals," AWS Whitepapers, 2023. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/saas-architecture-fundamentals/data-partitioning.html

[10] S. Pushpan, "Multi-tenant architecture: A comprehensive framework for building scalable SaaS applications," Int. J. Sci. Res. Comput. Sci. Eng. Inf. Technol., 2024.

[11] J. Kabbedijk, M. Pors, S. Jansen, and S. Brinkkemper, "Multi-tenant architecture comparison," in Proc. Eur. Conf. Softw. Archit., 2014.

[12] P. Mangwani, N. Mangwani, and S. Motwani, "Evaluation of a multitenant SaaS using monolithic and microservice architectures," SN Comput. Sci., vol. 4, 2023, doi: 10.1007/s42979-022-01610-2.

[13] M. D. Samrajesh and K. Viswanathan, "Reliable component instance for multi-tenant software as a service application," in Proc. IEEE Int. Conf. Comput. Sci. Eng. (CSE) Embedded Ubiquitous Comput. (EUC), 2019, pp. 159-164.

[14] S. Aleem, F. Ahmed, R. Batool, and A. Khattak, "Empirical investigation of key factors for SaaS architecture dimension," IEEE Trans. Cloud Comput., 2019, doi: 10.1109/TCC.2019.2906299.

[15] A. Santhosh et al., "SaaS (Software as a Service) and its impact on business scalability," Int. Res. J. Adv. Eng. Manag. (IRJAEM), 2024.

[16] S. Achar and N. L. Tisuela, "A review of hosting enterprise SaaS with IaC on multi-cloud platforms," Int. J. Reciprocal Symmetry Phys. Sci., vol. 7, pp. 14-23, 2020.

[17] S. Achar, "Enterprise SaaS workloads on new-generation infrastructure-as-code (IaC) on multi-cloud platforms," Glob. Disclosure Econ. Bus., 2021.

[18] M. Lukings and A. H. Lashkari, "Data sovereignty," in Understanding cybersecurity law in data sovereignty and digital governance. Cham: Springer, 2022, pp. 1-12, doi: 10.1007/978-3-031-14264-2_1.

[19] C. R. Baudoin, "The impact of data residency on cloud computing," in Proc. 32nd Int. Conf. Adv. Inf. Netw. Appl. Workshops (WAINA), Krakow, Poland, 2018, pp. 430-435, doi: 10.1109/WAINA.2018.00124.

[20] E. G. Chukwurah, "Leading SaaS innovation within U.S. regulatory boundaries: The role of TPMS in navigating compliance," Eng. Sci. Technol. J., vol. 9, no. 2, pp. 45-60, 2024.

[21] L. Yang, Y. Lin, and B. Chen, "Practice and prospect of regulating personal data protection in China," Laws, vol. 13, no. 1, p. 5, 2024.

[22] A. García-Fernández, J. A. Parejo, and A. Ruiz-Cortés, "Pricing-driven development and operation of SaaS: Challenges and opportunities," arXiv preprint, arXiv:2403.14007, 2024. [Online]. Available: https://arxiv.org/abs/2403.14007

[23] B. Li and S. Kumar, "Managing software-as-a-service: Pricing and operations," Prod. Oper. Manag., vol. 31, no. 7, pp. 2588-2608, 2022.

APPENDICES

Appendix - A - Part of Tenant Manager API Code

```
// Create Business
func (svc business) Creat(ctx context.Context, buss businesses.CreateBusiness) (string, error) {

  // Get country details
  country, err := svc.country.Get(ctx, buss.CountryCode)
  if err != nil {
     logger.Error("failed to fetch country", logger.String("country", buss.CountryCode), logger.String("err", err.Error()))
     return "", entities.NewError("failed to fetch country", http.StatusUnprocessableEntity)
  }

  if !country.Status {
     logger.Error("country is not active", logger.String("country", buss.CountryCode))
     return "", entities.NewError("country is not active", http.StatusUnprocessableEntity)
  }

  // Get Region details
  region, err := svc.region.Get(ctx, country.RegionID)
  if err != nil {
     logger.Error("failed to fetch region", logger.String("region", country.RegionID), logger.String("err", err.Error()))
     return "", entities.NewError("failed to fetch region", http.StatusUnprocessableEntity)
  }

  if !region.Status {
     logger.Error("region is not active", logger.String("region", country.RegionID))
     return "", entities.NewError("region is not active", http.StatusUnprocessableEntity)
  }

  // Get AppPlane details
  appPlane, err := svc.appPlane.GetByRegionID(ctx, region.ID)
  if err != nil {
     logger.Error("failed to fetch app plane", logger.String("region", country.RegionID), logger.String("err", err.Error()))
     return "", entities.NewError("failed to fetch app plane", http.StatusUnprocessableEntity)
  }

  if !appPlane.Status {
     logger.Error("app plane is not active", logger.String("region", country.RegionID))
     return "", entities.NewError("app plane is not active", http.StatusUnprocessableEntity)
  }

  if appPlane.InfraStatus == app_planes.STOPPED {
     logger.Error("app plane is unavailable", logger.String("region", country.RegionID))
     return "", entities.NewError("app plane isunavailable", http.StatusUnprocessableEntity)

  }

  // check alias exists
  _, err = svc.repo.GetByAlias(ctx, buss.Alias)
  if err != nil && !errors.Is(err, entities.ErrBusinessNotFound) {
     logger.Error("failed to fetch business", logger.String("alias", buss.Alias), logger.String("err", err.Error()))
     return "", entities.NewError("failed to fetch business", http.StatusUnprocessableEntity)
  }

  if err == nil {
     logger.Error("alias already exists", logger.String("alias", buss.Alias))
     return "", entities.NewError("alias already exists", http.StatusConflict)
  }

  // check email doesn't exists
  req := users.UserRequest{
     Email: buss.User.Email,
     Type: users.BUSINESS_ADMIN,
  }

  _, err = svc.user.GetUserByEmail(ctx, req)
  if err != nil && !errors.Is(err, entities.ErrUserNotFound) {
     logger.Error("failed to fetch user", logger.String("email", buss.User.Email), logger.String("err", err.Error()))
     return "", entities.NewError("failed to fetch user", http.StatusUnprocessableEntity)
  }
```

```
  if err == nil {
    logger.Error("user email already exists", logger.String("email", buss.User.Email))
    return "", entities.NewError("user with email already exists", http.StatusConflict)
  }
  businessID := libs.GenerateUniqueID()
  userID := libs.GenerateUniqueID()
  now := time.Now()

  // Hash and salt password
  passwd, err := svc.auth.HashAndSalt(entity.User.Password)
  if err != nil {
    logger.Error("failed to hash password", logger.String("err", err.Error()))
    return "", entities.NewError("failed to hash password", http.StatusUnprocessableEntity)
  }

  buss := businesses.CreateBusiness{
    ID: businessID,
    Name: entity.Name,
    Alias: entity.Alias,
    CountryCode: entity.CountryCode,
    AppPlaneID: entity.AppPlaneID,
    Description: entity.Description,
    Type: entity.Type,
    Tier: tiers.FREE,
    AppPlane: entity.AppPlane,
    Status: true,
    Restriction: 0,
    User: users.CreateUser{
      ID: userID,
      BusinessID: businessID,
      Email: entity.User.Email,
      FirstName: entity.User.FirstName,
      LastName: entity.User.LastName,
      Password: passwd,
      Type: users.BUSINESS_ADMIN,
      Status: true,
      Verified: true,
      CreatedAt: now.Format(time.RFC3339),
      UpdatedAt: now.Format(time.RFC3339),
      CreatedBy: userID,
      UpdatedBy: userID,
    },
    CreatedAt: now.Format(time.RFC3339),
    UpdatedAt: now.Format(time.RFC3339),
    CreatedBy: userID,
    UpdatedBy: userID,
  }

  buss.TrimSpaces()

  err = buss.Validate()
  if err != nil {
    logger.Error("failed to validate business", logger.String("err", err.Error()))
    return "", entities.NewError("failed to validate business", http.StatusUnprocessableEntity)
  }

  // Save to repository
  id, err := svc.repo.Create(ctx, buss)
  if err != nil {
    logger.Error("failed to save business", logger.String("err", err.Error()))
    return "", entities.NewError("failed to save business", http.StatusUnprocessableEntity)
  }

  return id, nil
}


// Admin Login
func (svc auth) Login(ctx context.Context, login authEntities.Login) (authEntities.LoginResponse, error) {

  response := authEntities.LoginResponse{}

  request := users.UserRequest{
    Email: login.Email,
    Type: login.UserType,
  }
```

```
// get user entity by email
user, err := svc.user.GetUserByEmail(ctx, request)
if err != nil {
   logger.Error("error get user by email", logger.String("email", login.Email), logger.String("err", err.Error()))
   return response, err
}

// check user is enable
if !user.Status {
   logger.Error("user is not enable", logger.String("email", login.Email), logger.String("user_id", user.ID))
   return response, entities.NewError("access denied", 403)
}

// validate passwd match
matched, err :=  svc.adapter.ComparePasswd(user.Password, login.Password)
if err != nil {
   logger.Error("error compare password", logger.String("email", login.Email), logger.String("err", err.Error()))
   return response, errors.New("error compare password")
}

if !matched {
   logger.Error("wrong password", logger.String("email", login.Email))
   return response, errors.New("wrong password")
}

currentTime := time.Now()

// create access token
accessClaim := &authEntities.JWTClaims{
   BusinessID: user.BusinessID,
   UserID: user.ID,
   UserType: user.Type,
   DeviceID: login.DeviceID,
   DeviceType: login.DeviceType,
   IssuedAt: currentTime.Unix(),
   ExpiredAt: currentTime.Unix()+ svc.ttl,
}

// validate minimal access token information
err = accessClaim.Valid()
if err != nil {
   logger.Error("error validate access token", logger.String("email", login.Email), logger.String("err", err.Error()))
   return response, err
}

// generate access token
response.AccessToken, err = svc.adapter.GenerateJWT(ctx, accessClaim)
if err != nil {
   logger.Error("error generate access token", logger.String("email", login.Email), logger.String("err", err.Error()))
   return response, err
}

// create refresh token
refreshClaim := &authEntities.RefreshClaims{
   BusinessID: user.BusinessID,
   UserID: user.ID,
   UserType: user.Type,
   EmailVerified: user.Verified,
   DeviceID: login.DeviceID,
   DeviceType: login.DeviceType,
   IssuedAt: currentTime.Unix(),
   ExpiredAt: currentTime.Unix()+ svc.days*3600*24,
}

// validate minimal refresh token information
err = refreshClaim.Valid()
if err != nil {
   logger.Error("error validate refresh token", logger.String("email", login.Email), logger.String("err", err.Error()))
   return response, err
}

// generate refresh token
response.RefreshToken, err = svc.adapter.GenerateRefresh(ctx, refreshClaim)
if err != nil {
   logger.Error("error generate refresh token", logger.String("email", login.Email), logger.String("err", err.Error()))
```

```
        return response, err
    }

    // create session
    session := authEntities.Session {
        UserID: user.ID,
        BusinessID: user.BusinessID,
        DeviceID: login.DeviceID,
        UserType: user.Type,
        AccessToken: response.AccessToken,
        RefreshToken: response.RefreshToken,
        CreatedAt: currentTime.Format(time.RFC3339),
        ExpiredAt: currentTime.Unix()+ svc.days*3600*24,
    }

    err = session.Valid()
    if err != nil {
        logger.Error("error validate session", logger.String("email", login.Email), logger.String("err", err.Error()))
        return response, err
    }

    err = svc.repo.StoreSession(ctx, session)
    if err != nil {
        logger.Error("error store session", logger.String("email", login.Email), logger.String("err", err.Error()))
        return response, err
    }

    if login.UserType == users.BUSINESS_ADMIN {
        // get business entity
        business, err := svc.buss.Get(ctx, user.BusinessID)
        if err != nil {
            logger.Error("error get business", logger.String("email", login.Email), logger.String("err", err.Error()))
            return response, err
        }

        appPlane, err := svc.appPlane.Get(ctx, business.AppPlaneID)
        if err != nil {
            logger.Error("error get app plane", logger.String("email", login.Email), logger.String("err", err.Error()))
            return response, err
        }

        response.Endpoint = "https://" + appPlane.Endpoint
    }

    return response, err
}

// Get Business Admin Session
func (svc auth) GetSession(ctx context.Context, token string) (*authEntities.Session, error) {

    claims, err := svc.adapter.ValidateJWT(ctx, token)
    if err != nil {
        logger.Error("error validate token", logger.String("err", err.Error()))
        return nil, err
    }

    if claims.UserType != users.BUSINESS_ADMIN {
        logger.Error("user type is not business admin", logger.String("user_type", string(claims.UserType)))
        return nil, entities.NewError("access denied", 403)
    }

    session, err := svc.repo.GetSession(ctx, claims)
    if err != nil {
        logger.Error("error get session", logger.String("token", token), logger.String("err", err.Error()))
        return nil, err
    }

    err = session.Valid()
    if err != nil {
        logger.Error("error validate session", logger.String("err", err.Error()))
        return nil, err
    }

    return session, nil
```

Appendix - B - Part of Control Plane Terraform Code

```
// AWS Apprunner
resource "aws_apprunner_service" "app_runner_service" {
 service_name = "${var.infra}-${var.region}-app-runner"

 instance_configuration {
  instance_role_arn = var.app_runner_instance_role_arn
  cpu = "0.25 vCPU"
  memory = "0.5 GB"
 }

 auto_scaling_configuration_arn = aws_apprunner_auto_scaling_configuration_version.app_runner_service.arn

 tags = {
  infra = "${var.infra}-${var.region}"
 }
 source_configuration {
  image_repository {
   image_repository_type = "ECR"
   image_identifier = "${var.ecr_repository_url}:${var.ecr_image_tag}"
   image_configuration {
    port = "8080"

    runtime_environment_variables = {
     "ENVIRONMENT" = "production"
     "MODE"        = "test"
     "APP_REGION" = var.region
     "APP_VERSION" = var.ecr_image_tag
     "APP_HOST" = "0.0.0.0"
     "APP_PORT" = "8080"
     "APP_DOMAIN" = var.domain_name
     "EMAIL_REDIRECT_ENDPOINT" = "/"
     "RESET_PASSWD_REDIRECT_ENDPOINT" = "/"

     "PROFILE" = var.profile
     "REGION" = var.region

     "SECRET_KEY" = "hello-word"
     "VERSION" = "v1"
     "ACCESS_TOKEN_TTL" = 86400
     "REFRESH_TOKEN_TTL" = 30
     "EMAIL_TOKEN_TTL" = 3600
     "PASSWD_RESET_TOKEN_TTL" = 300
     "ISSUER" = "tenant-manager"

     "TABLE_NAME" = var.db_name

     "STMP_HOST" = "smtppro.zoho.com"
     "STMP_PORT" = "587"
     "STMP_USER_NAME" = "no-reply@sellertower.com"
     "STMP_PASSWD" = "xx"

     "BASE_URL" = "https://gitlab.com/api/v4"
     "ACCESS_TOKEN" = "xxx"
     "TRIGGER_TOKEN" = "xxx"
     "APP_PLANE_ID" = "xxx"
     "RETAIL_MANAGER_ID" = "xx"
    }
   }
  }
  auto_deployments_enabled = false
  authentication_configuration {
   access_role_arn = var.app_runner_access_role_arn
  }
 }

 network_configuration {
  ingress_configuration {
   is_publicly_accessible = true
  }
 }
}
```

```
resource "aws_apprunner_auto_scaling_configuration_version" "app_runner_service" {
 auto_scaling_configuration_name = "${var.infra}-${var.region}-as"

 max_concurrency = 10
 max_size        = 2
 min_size        = 1

 tags = {
   app = "${var.infra}-${var.region}"
 }
}

resource "aws_apprunner_custom_domain_association" "app_runner_service" {
 domain_name = var.domain_name
 service_arn = aws_apprunner_service.app_runner_service.arn
}

// AWS Dynamo DB
provider "aws" {
 alias  = "main"
 region = var.primary_region
 profile = var.profile
}

data "aws_dynamodb_table" "primary_table" {
 provider = aws.main
 name   = var.table_name
}

# Create a replica only if the replica region is different from the primary region
resource "aws_dynamodb_table_replica" "replica" {
 global_table_arn = data.aws_dynamodb_table.primary_table.arn
 count = var.region != var.primary_region ? 1 : 0

 tags = {
   app = "${var.infra}-${var.region}"
 }
}

// AWS Route53
data "aws_apprunner_hosted_zone_id" "main" {}

resource "null_resource" "cert_validations_records" {
 provisioner "local-exec" {
   when = create
   command = <<EOT
   aws route53 --profile ${var.profile} change-resource-record-sets --hosted-zone-id ${var.zone_id} --change-batch '{
     "Changes": [
       ${join(", ", [
         for record in var.certificate_validation_records : jsonencode({
           "Action"            = "UPSERT"
           "ResourceRecordSet" = {
             "Name"            = record.name
             "Type"            = "CNAME"
             "TTL"             = 60
             "ResourceRecords" = [
               { "Value" = record.value }
             ]
           }
         })
       ])}
     ]
   }'
   EOT
 }

 depends_on = [var.custom_domain_service_arn]
}

resource "null_resource" "cert_validations_records_destroy" {
 triggers = {
   # Trigger on change to certificate_validation_records
   validation_records = jsonencode(var.certificate_validation_records)
   zone_id = var.zone_id
   profile = var.profile
 }
```

```
 provisioner "local-exec" {
   when    = destroy
   command = <<EOT
   export AWS_PROFILE=${self.triggers.profile}
   aws route53 change-resource-record-sets --hosted-zone-id ${self.triggers.zone_id} --change-batch '{
     "Changes": [
       ${join(", ", [
         for record in jsondecode(self.triggers.validation_records) : jsonencode({
           "Action"            = "DELETE"
           "ResourceRecordSet" = {
             "Name"            = record.name
             "Type"            = "CNAME"
             "TTL"             = 60
             "ResourceRecords" = [
               { "Value" = record.value }
             ]
           }
         })
       ])}
     ]
   }'
   EOT
 }

 depends_on = [var.custom_domain_service_arn]
}

// Simple routing
resource "aws_route53_record" "app_runner_alias" {
 zone_id = var.zone_id
 name    = var.domain_name
 type    = "A"

 alias {
   name                   = var.target_dns_name
   zone_id                = data.aws_apprunner_hosted_zone_id.main.id
   evaluate_target_health = false
 }

 set_identifier = "${var.infra}-${var.region}"
 latency_routing_policy {
   region = var.region  # Traffic from users near this region will be routed here
 }
}
```

Appendix - C - Part of Control Plane Gitlab Pipeline Code

```
// CI Pipeline
variables:
 DOCKER_HOST: tcp://docker:2375
 DOCKER_DRIVER: overlay2

validate_tag:
 stage: .pre
 rules:
  - if: '$CI_COMMIT_TAG =~ /^v\d+\.\d+\.\d+$/'
 script:
  - allowed_branches='main'
  - git fetch --unshallow || true
  - git fetch --all --tags
  - |
    if git merge-base --is-ancestor "$CI_COMMIT_SHA" "origin/main"; then
       echo "Tag $CI_COMMIT_TAG belongs to an acceptable branch: main"
       exit 0
    else
       echo "Tag $CI_COMMIT_TAG does not belong to any acceptable branch."
       exit 1
    fi

build:
 stage: build
 image:
  name: docker:24.0.5  # Use a Docker image that supports CLI commands
  entrypoint: [""]
 services:
  - docker:24.0.5-dind
 rules:
  - if: '$CI_COMMIT_TAG =~ /^v\d+\.\d+\.\d+$/'
 before_script:
  - apk add --no-cache curl python3 py3-pip  # Install dependencies
 script:
  # Verify tools
  - docker --version

  # Build Docker image
  - echo "Building Docker image with tag - $CI_COMMIT_TAG"
  - docker build -t $DOCKER_REGISTRY/$APP_NAME:$CI_COMMIT_TAG .
  - docker save $DOCKER_REGISTRY/$APP_NAME:$CI_COMMIT_TAG > image.tar
 artifacts:
  paths:
   - image.tar

upload:
 stage: upload
 image:
  name: docker:24.0.5  # Use a Docker image that supports CLI commands
  entrypoint: [""]
 services:
  - docker:24.0.5-dind
 rules:
  - if: '$CI_COMMIT_TAG =~ /^v\d+\.\d+\.\d+$/'
 before_script:
  - apk add --no-cache curl python3 py3-pip  # Install dependencies
  - pip3 install awscli                      # Install AWS CLI
 script:
  # Verify tools
  - aws --version
  - docker --version

  - echo "Loading Docker image with tag - $CI_COMMIT_TAG"
  - docker load < image.tar

  # Authenticate with AWS CLI
  - aws configure set aws_access_key_id $AWS_ACCESS_KEY_ID
  - aws configure set aws_secret_access_key $AWS_SECRET_ACCESS_KEY
  - aws configure set region $AWS_DEFAULT_REGION

  # Login to AWS ECR
  - aws ecr get-login-password --region $AWS_DEFAULT_REGION | docker login --username AWS --password-stdin
$DOCKER_REGISTRY
```

```
    # Push Docker image
    - docker push $DOCKER_REGISTRY/$APP_NAME:$CI_COMMIT_TAG

  release:
   stage: release
   image: registry.gitlab.com/gitlab-org/release-cli:latest
   rules:
    - if: '$CI_COMMIT_TAG =~ /^v\d+\.\d+\.\d+$/'
   script:
    - echo "running release_job for $CI_COMMIT_TAG"
   release:
    name: 'Release $CI_COMMIT_TAG'
    description: 'Created the release for tag $CI_COMMIT_TAG'
    tag_name: '$CI_COMMIT_TAG'

  // CD Pipeline
  val-vars:
   stage: val-vars
   rules:
    - if: '($CI_PIPELINE_SOURCE == "trigger" || $CI_PIPELINE_SOURCE == "web") && $CI_COMMIT_TAG =~
  /^v\d+\.\d+\.\d+$/ && $ACTION == "deploy"'
   script:
    - echo "Validating required variables..."
    - |

      # Pass via GitLab variables
      if [ -z "$AWS_ACCESS_KEY_ID" ]; then
        echo "AWS_ACCESS_KEY_ID is required"; exit 1;
      fi

      if [ -z "$AWS_SECRET_ACCESS_KEY" ]; then
        echo "AWS_SECRET_ACCESS_KEY is required"; exit 1;
      fi

      if [ -z "$ECR_REGISTRY" ]; then
        echo "ECR_REGISTRY is required"; exit 1;
      fi

      if [ -z "$COMMON_INFRA_REGION" ]; then
        echo "COMMON_INFRA_REGION is required"; exit 1;
      fi

      if [ -z "$HOSTED_ZONE_ID" ]; then
        echo "HOSTED_ZONE_ID is required"; exit 1;
      fi

      if [ -z "$INFRA" ]; then
        echo "INFRA is required"; exit 1;
      fi

      if [ -z "$REPO_NAME" ]; then
        echo "REPO_NAME is required"; exit 1;
      fi

      if [ -z "$DOMAIN_NAME" ]; then
        echo "DOMAIN_NAME is required"; exit 1;
      fi

      # Pass when triggering the pipeline
      if [ -z "$REGION" ]; then
        echo "REGION is required"; exit 1;
      fi

      if [ -z "$APP_VERSION" ]; then
        echo "APP_VERSION is required"; exit 1;
      fi

      if ! echo "$APP_VERSION" | grep -Eq "^v[0-9]+\.[0-9]+\.[0-9]+$"; then
        echo "APP_VERSION must match format 'vX.X.X'. Provided, $APP_VERSION"; exit 1;
      fi

    - echo "AWS_ACCESS_KEY_ID=$AWS_ACCESS_KEY_ID"
    - echo "AWS_SECRET_ACCESS_KEY=$AWS_SECRET_ACCESS_KEY"
    - echo "ECR_REGISTRY=$ECR_REGISTRY"
    - echo "COMMON_INFRA_REGION=$COMMON_INFRA_REGION"
```

```
  - echo "INFRA=$INFRA"
  - echo "REPO_NAME=$REPO_NAME"
  - echo "DOMAIN_NAME=$DOMAIN_NAME"
  - echo "REGION=$REGION"
  - echo "APP_VERSION=$APP_VERSION"
  - echo "Required variables validated successfully."

val-ecr-img:
 stage: val-ecr-img
 image:
  name: docker:24.0.5 # Use a Docker image that supports CLI commands
  entrypoint: [""]
 services:
  - name: docker:24.0.5-dind
 variables:
  DOCKER_HOST: tcp://localhost:2375
 rules:
  - if: '($CI_PIPELINE_SOURCE == "trigger" || $CI_PIPELINE_SOURCE == "web") && $CI_COMMIT_TAG =~
/^v\d+\.\d+\.\d+$/ && $ACTION == "deploy"'
 before_script:
  - apk add --no-cache curl python3 py3-pip # Install dependencies
  - pip3 install awscli # Install AWS CLI
 script:
  # Verify required tools
  - aws --version
  - docker --version

  - echo "Validating ECR image, REPO:$REPO_NAME VERSION:$APP_VERSION"

  - echo "Configuring AWS CLI..."
  - aws configure set aws_access_key_id $AWS_ACCESS_KEY_ID
  - aws configure set aws_secret_access_key $AWS_SECRET_ACCESS_KEY
  - aws configure set region $COMMON_INFRA_REGION

  - |
    IMAGE_URI=$ECR_REGISTRY/$REPO_NAME:$APP_VERSION

    echo "Logging in to AWS ECR in region $COMMON_INFRA_REGION"

    aws ecr get-login-password --region "$COMMON_INFRA_REGION" | docker login --username AWS --password-stdin
"$ECR_REGISTRY"

    echo "Inspecting image manifest for $IMAGE_URI";

    if ! docker manifest inspect "$IMAGE_URI" > /dev/null 2>&1; then
      echo "ECR image $IMAGE_URI does not exist or is unavailable"; exit 1;
    fi

  - echo "ECR image validation completed successfully."

deploy:
 stage: deploy
 image:
  name: docker:24.0.5 # Use a Docker image that supports CLI commands
  entrypoint: [""]
 services:
  - docker:24.0.5-dind
 rules:
  - if: '($CI_PIPELINE_SOURCE == "trigger" || $CI_PIPELINE_SOURCE == "web") && $CI_COMMIT_TAG =~
/^v\d+\.\d+\.\d+$/ && $ACTION == "deploy"'
 before_script:
  # Install prerequisites
  - echo "Installing dependencies..."
  - apk update
  - apk add --no-cache git wget unzip bash python3 py3-pip # Install dependencies
  - pip3 install awscli # Install AWS CLI
 script:
  # Install Terraform
  - echo "Installing Terraform..."
  - wget https://releases.hashicorp.com/terraform/${TF_VERSION}/terraform_${TF_VERSION}_linux_amd64.zip
  - unzip -d /usr/bin/ terraform_${TF_VERSION}_linux_amd64.zip
  - chmod +x /usr/bin/terraform
  - terraform --version
  - rm terraform_${TF_VERSION}_linux_amd64.zip

  # Set AWS credentials using environment variables
```

```
  - echo "Configuring AWS CLI..."
  - aws configure set --profile $AWS_PROFILE aws_access_key_id $AWS_ACCESS_KEY_ID
  - aws configure set --profile $AWS_PROFILE aws_secret_access_key $AWS_SECRET_ACCESS_KEY
  - aws configure set --profile $AWS_PROFILE region $REGION

  - echo "Checking out Git tag $CI_COMMIT_TAG..."
  - git fetch --tags
  - git checkout $CI_COMMIT_TAG

  - echo "Deploying $INFRA:$CI_COMMIT_TAG tenant-manager:$APP_VERSION to $REGION from
REPO:$REPO_NAME"

  # Create variables file for cleaner Terraform commands
  - |
    cat > terraform.tfvars <<EOF
    profile = "$AWS_PROFILE"
    region = "$REGION"
    infra = "$INFRA"
    ecr_repository_name = "$REPO_NAME"
    common_infra_region = "$COMMON_INFRA_REGION"
    ecr_image_tag = "$APP_VERSION"
    zone_id = "$HOSTED_ZONE_ID"
    domain_name = "$DOMAIN_NAME"
    EOF

  ## Terraform initialization
  - echo "Initializing Terraform..."
  - terraform init -reconfigure -backend-config="region=$COMMON_INFRA_REGION"
-backend-config="key=$INFRA/$REGION/terraform.tfstate" 2>&1

  # Plan Terraform changes
  - echo "Planning Terraform changes..."
  - terraform plan -input=false -var-file=terraform.tfvars

  # Apply Terraform changes
  - echo "Applying Terraform changes..."
  - terraform apply -input=false -auto-approve -var-file=terraform.tfvars

  # Output handling
  - echo "Retrieving App Runner service ARN..."
  - |
    SERVICE_ARN=$(terraform output -raw app_runner_service_arn 2>&1)
    if [[ "$SERVICE_ARN" =~ Error|Warning ]]; then
      echo "Terraform output failed: $SERVICE_ARN"
      exit 1
    fi

    echo "Successfully retrieved Service ARN: $SERVICE_ARN"
    echo "SERVICE_ARN=$SERVICE_ARN" > artifact.txt

  # Clean up
  - rm -f terraform.tfvars

 artifacts:
   when: always
   paths:
     - artifact.txt
   expire_in: 1 hour

verify-deploy:
 stage: verify-deploy
 image:
   name: docker:24.0.5 # Use a Docker image that supports CLI commands
   entrypoint: [""]
 services:
   - docker:24.0.5-dind
 rules:
   - if: '($CI_PIPELINE_SOURCE == "trigger" || $CI_PIPELINE_SOURCE == "web") && $CI_COMMIT_TAG =~
/^v\d+\.\d+\.\d+$/ && $ACTION == "deploy"'
 before_script:
   # Install prerequisites
   - echo "Installing dependencies..."
   - apk update
   - apk add --no-cache git python3 py3-pip # Install dependencies
   - pip3 install awscli # Install AWS CLI
 script:
```

```
  - echo "Retrieving App Runner service ARN..."
  - |
    if [ -f artifact.txt ]; then
      export SERVICE_ARN=$(awk -F'=' '/^SERVICE_ARN/ {print $2}' artifact.txt)
      echo "SERVICE_ARN is $SERVICE_ARN"
    fi

    if [ -z "$SERVICE_ARN" ]; then
      echo "Error - SERVICE_ARN is empty"; exit 1;
    fi

  - echo "Validating App Runner service status for ARN - $SERVICE_ARN"

  # Set AWS credentials using environment variables
  - echo "Configuring AWS CLI..."
  - aws configure set --profile $AWS_PROFILE aws_access_key_id $AWS_ACCESS_KEY_ID
  - aws configure set --profile $AWS_PROFILE aws_secret_access_key $AWS_SECRET_ACCESS_KEY
  - aws configure set --profile $AWS_PROFILE region $REGION

  # Check the last deployment status
  - echo "Check last deployment status..."
  - |
    DEPLOYMENT_STATUS=$(aws apprunner list-operations --service-arn "$SERVICE_ARN" --query
"OperationSummaryList[0].Status" --output text)

    if echo "$DEPLOYMENT_STATUS" | grep -qE "Warning|Error"; then
      echo "List operations failed. Error - $(echo "$DEPLOYMENT_STATUS" | grep -o 'Error:.*\|Warning:.*')"; exit 1;
    fi

    if [ -z "$DEPLOYMENT_STATUS" ]; then
      echo "No deployment status found or operations are empty"; exit 1;


    elif [[ "$DEPLOYMENT_STATUS" == "FAILED" || "$DEPLOYMENT_STATUS" == "ROLLBACK_SUCCEEDED" ]];
then
      echo "Deployment failed or rolled back"; exit 1;

    elif [[ "$DEPLOYMENT_STATUS" == "SUCCEEDED" ]]; then
      echo "Deployment succeeded. Current status: $DEPLOYMENT_STATUS";

    else
      echo "Deployment not succeeded: $DEPLOYMENT_STATUS"; exit 1;
    fi

  - echo "Validating App Runner service status..."
  - |
    SERVICE_STATUS=$(aws apprunner describe-service --service-arn "$SERVICE_ARN" --query "Service.Status" --output
text)

    if echo "$SERVICE_STATUS" | grep -qE "Warning|Error"; then
      echo "Describe service failed. Error - $(echo "$SERVICE_STATUS" | grep -o 'Error:.*\|Warning:.*')"; exit 1;
    fi

    if [[ "$SERVICE_STATUS" == "RUNNING" || "$SERVICE_STATUS" == "PAUSED" ]]; then
      echo "Validation succeeded. Current status: $SERVICE_STATUS";

    else
      echo "Validation failed. Current status: $SERVICE_STATUS"; exit 1;
    fi

  - echo "SUCCESS=Deploy pipeline succeeded" | tee -a artifact.txt

 dependencies:
  - deploy # Ensure artifacts from deploy stage are available here
```

Appendix - D - Part of Retail Manager API Code

```
// Business User Login
func (uc *auth) Login(ctx context.Context, login authEntities.Login) (authEntities.LoginResponse, error) {

  response := authEntities.LoginResponse{}

  user, err := uc.user.GetUserByEmail(ctx, users.UserRequest{
     Email:      login.Email,
     BusinessID: login.BusinessID,
  })
  if err != nil {
     logger.Error("failed to fetch user by email", logger.String("email", login.Email), logger.String("err", err.Error()))
     return response, errors.New("invalid email or password")
  }

  if !user.Status {
     logger.Error("user account is disabled", logger.String("userID", user.ID))
     return response, entities.NewError("account disabled", 403)
  }

  matched, err := uc.adapter.ComparePasswd(user.Password, login.Password)
  if err != nil {
     logger.Error("failed to compare passwords", logger.String("err", err.Error()))
     return response, errors.New("internal server error")
  }

  if !matched {
     logger.Error("invalid password attempt", logger.String("userID", user.ID))
     return response, errors.New("invalid email or password")
  }

  currentTime := time.Now()
  accessToken, refreshToken, err := uc.generateTokens(ctx, user, login.DeviceID, login.DeviceType, currentTime)
  if err != nil {
     logger.Error("failed to generate tokens", logger.String("err", err.Error()))
     return response, errors.New("failed to generate tokens")
  }

  session := authEntities.Session{
     UserID:       user.ID,
     BusinessID:   user.BusinessID,
     StoreID:      user.StoreID,
     DeviceID:     login.DeviceID,
     UserType:     user.Type,
     AccessToken:  accessToken,
     RefreshToken: refreshToken,
     CreatedAt:    currentTime.String(),
     ExpiredAt:    currentTime.Unix() + uc.days*3600*24,
  }

  if err := uc.repo.StoreSession(ctx, session); err != nil {
     logger.Error("failed to store session", logger.String("err", err.Error()))
     return response, errors.New("failed to store session")
  }

  response.AccessToken = accessToken
  response.RefreshToken = refreshToken

  return response, nil
}


// Get Business
func (handler business) Get(w http.ResponseWriter, r *http.Request) {

  ctx := r.Context()

  session := ctx.Value(global.SESSION_KEY).(*authEntities.Session)

  buss, err := handler.service.Get(ctx, session.BusinessID)
  if err != nil {
     logger.Error("failed to get business", logger.String("err", err.Error()))
     response.HandleError(w, err, http.StatusUnprocessableEntity)
```

```
        return
    }

    payload := response.Encode(buss, nil, true, nil)
    response.Send(w, payload, http.StatusOK)
}

// Create Business User
func (uc *user) Create(ctx context.Context, user users.CreateUser) (string, error) {

    session := ctx.Value(global.SESSION_KEY).(*authEntities.Session)

    if user.BusinessID != session.BusinessID {
        logger.Error("unauthorized attempt to create user for another business", logger.String("sessionBusinessID",
session.BusinessID), logger.String("userBusinessID", user.BusinessID))
        return "", entities.NewError("access denied", 403)
    }

    passwd, err := uc.auth.HashAndSalt(user.Password)
    if err != nil {
        logger.Error("failed to hash password", logger.String("err", err.Error()))
        return "", entities.NewError("failed to hash password", http.StatusUnprocessableEntity)
    }

    now := time.Now()
    user.ID = libs.GenerateUniqueID()
    user.CreatedAt = now.String()
    user.UpdatedAt = now.String()
    user.Status = true
    user.Password = passwd
    user.CreatedBy = session.UserID
    user.UpdatedBy = session.UserID

    // Create user
    id, err := uc.repo.Create(ctx, user)
    if err != nil {
        logger.Error("failed to create user", logger.String("email", user.Email), logger.String("err", err.Error()))
        return "", errors.New("failed to create user")
    }

    return id, nil
}


// Authorization Middleware
func (mdlWare *Auth) Middleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {

        ctx := r.Context()

        authHeader := r.Header.Get("Authorization")

        if !strings.HasPrefix(authHeader, "Bearer ") {
            response.HandleError(w, errors.New("token not provided"), http.StatusUnauthorized)
            return
        }

        token := strings.TrimPrefix(authHeader, "Bearer ")
        claims, err := mdlWare.adapter.ValidateJWT(ctx, token)

        var session *authEntities.Session

        if err != nil && !errors.Is(err, entities.ErrAdminToken) {
            response.HandleError(w, err, http.StatusUnauthorized)
            return
        }

        if err != nil && errors.Is(err, entities.ErrAdminToken) {
            sess, e := mdlWare.cpAdapter.GetSession(ctx, token)
            if e != nil {
                response.HandleError(w, err, http.StatusUnauthorized)
                return
            }

            session = sess
        }
```

```
        if err == nil {
            session, err = mdlWare.repo.GetSession(ctx, claims)
            if err != nil {
                response.HandleError(w, err, http.StatusUnauthorized)
                return
            }
        }

        err = session.Valid()
        if err != nil {
            response.HandleError(w, err, http.StatusUnauthorized)
            return
        }

        ctx = context.WithValue(ctx, global.SESSION_KEY, session)
        next.ServeHTTP(w, r.WithContext(ctx))
    })
}


func (cp *CP) GetSession(ctx context.Context, token string) (*auth.Session, error) {
    url := cp.cpDomain + cp.cpRemotePath

    req, err := http.NewRequestWithContext(ctx, http.MethodGet, url, nil)
    if err != nil {
        logger.Error("Failed to create request", logger.String("url", url), logger.String("error", err.Error()))
        return nil, fmt.Errorf("failed to create request: %w", err)
    }

    req.Header.Set("Content-Type", "application/json")
    req.Header.Set("Authorization", "Bearer "+token)

    resp, err := cp.httpClient.Do(req)
    if err != nil {
        logger.Error("Failed to send request", logger.String("url", url), logger.String("error", err.Error()))
        return nil, fmt.Errorf("failed to send request: %w", err)
    }
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {
        body, _ := io.ReadAll(resp.Body)
        logger.Error("Received non-OK response", logger.String("url", url), logger.Int("statusCode", resp.StatusCode))
        return nil, fmt.Errorf("non-OK response from CP: %s", resp.Status)
    }

    var cpResponse entities.Response
    if err := json.NewDecoder(resp.Body).Decode(&cpResponse); err != nil {
        logger.Error("Failed to decode response", logger.String("url", url), logger.String("err", err.Error()))
        return nil, fmt.Errorf("failed to decode response: %w", err)
    }

    if !cpResponse.Success {
        logger.Error("Failed to get session from CP", logger.String("err", cpResponse.Error))
        return nil, fmt.Errorf("failed to get session from CP: %s", cpResponse.Error)
    }

    if cpResponse.Data == nil {
        logger.Error("Session data is nil", logger.String("url", url))
        return nil, fmt.Errorf("session data is nil")
    }

    dataBytes, err := json.Marshal(cpResponse.Data)
    if err != nil {
        logger.Error("Failed to marshal session data", logger.String("err", err.Error()))
        return nil, fmt.Errorf("failed to marshal session data: %w", err)
    }

    var session auth.Session
    if err := json.Unmarshal(dataBytes, &session); err != nil {
        logger.Error("Failed to unmarshal session", logger.String("err", err.Error()))
        return nil, fmt.Errorf("failed to unmarshal session: %w", err)
    }

    return &session, nil
```

Appendix - E - Part of Application Plane Terraform Code

```
// AWS Apprunner
resource "aws_apprunner_service" "app_runner_service" {
 service_name = "${var.infra}-${var.region}-ar"

 instance_configuration {
  instance_role_arn = var.app_runner_instance_role_arn
  cpu = "0.25 vCPU"
  memory = "0.5 GB"
 }

 auto_scaling_configuration_arn = aws_apprunner_auto_scaling_configuration_version.app_runner_service.arn

 tags = {
  infra = "${var.infra}-${var.region}"
 }
 source_configuration {
  image_repository {
   image_repository_type = "ECR"
   image_identifier = "${var.ecr_repository_url}:${var.ecr_image_tag}"
   image_configuration {
    port = "8080"

    runtime_environment_variables = {
     "ENVIRONMENT" = "production"
     "APP_REGION" = var.region
     "APP_VERSION" = var.ecr_image_tag
     "APP_HOST" = "0.0.0.0"
     "APP_PORT" = "8080"
     "APP_DOMAIN" = var.domain_name
     "EMAIL_REDIRECT_ENDPOINT" = "/"
     "RESET_PASSWD_REDIRECT_ENDPOINT" = "/"

     "PROFILE" = var.profile
     "REGION" = var.region

     "SECRET_KEY" = "hello-word"
     "VERSION" = "v1"
     "ACCESS_TOKEN_TTL" = 86400
     "REFRESH_TOKEN_TTL" = 30
     "EMAIL_TOKEN_TTL" = 3600
     "PASSWD_RESET_TOKEN_TTL" = 300
     "ISSUER" = "retail-manager"
     "CP_ISSUER" = "tenant-manager"

     "TABLE_NAME" = var.db_name
     "BUCKET_NAME" = var.bucket_name

     "STMP_HOST" = "smtppro.zoho.com"
     "STMP_PORT" = "587"
     "STMP_USER_NAME" = "no-reply@sellertower.com"
     "STMP_PASSWD" = "test"

     "CP_DOMAIN" = var.cp_domain
     "CP_AUTH_PATH" = "/api/auth/get-session"
    }
   }
  }
  auto_deployments_enabled = false
  authentication_configuration {
   access_role_arn = var.app_runner_access_role_arn
  }
 }

 network_configuration {
  ingress_configuration {
   is_publicly_accessible = true
  }
 }
}

resource "aws_apprunner_auto_scaling_configuration_version" "app_runner_service" {
 auto_scaling_configuration_name = "${var.infra}-${var.region}"
```

```
 max_concurrency = 10
 max_size        = 2
 min_size        = 1

 tags = {
   infra = "${var.infra}-${var.region}"
 }
}

resource "aws_apprunner_custom_domain_association" "app_runner_service" {
 domain_name = var.domain_name
 service_arn = aws_apprunner_service.app_runner_service.arn
}

// AWS DynamoDB

resource "aws_dynamodb_table" "retail_manager" {
 name         = var.table_name
 billing_mode = "PAY_PER_REQUEST"
 hash_key     = "PK"
 range_key    = "SK"

 tags = {
   infra = "${var.infra}-${var.region}"
 }

 dynamic "attribute" {
   for_each = var.attributes

   content {
     name = attribute.value["name"]
     type = attribute.value["type"]
   }
 }

 dynamic "local_secondary_index" {
   for_each = var.lsi_list
   content {
     name               = local_secondary_index.value["name"]
     range_key          = local_secondary_index.value["range"]
     projection_type    = local_secondary_index.value["projection_type"]
   }
 }

 # Define GSIs dynamically using a loop
 dynamic "global_secondary_index" {
   for_each = var.gsi_list
   content {
     name               = global_secondary_index.value["name"]
     hash_key           = global_secondary_index.value["hash"]
     range_key          = global_secondary_index.value["range"] != null ? global_secondary_index.value["range"] : null
     projection_type    = global_secondary_index.value["projection_type"]
   }
 }
}

// AWS S3 Bucket
resource "aws_s3_bucket" "images_bucket" {
 bucket = var.bucket_name

  tags = {
   infra = "${var.infra}-${var.region}"
 }
}
```

Appendix - F - Part of Application Plane Gitlab Pipeline Code

```
// CI Pipeline
variables:
 DOCKER_HOST: tcp://docker:2375
 DOCKER_DRIVER: overlay2

validate_tag:
 stage: .pre
 rules:
  - if: '$CI_COMMIT_TAG =~ /^v\d+\.\d+\.\d+$/'
 script:
  - allowed_branches='main'
  - git fetch --unshallow || true
  - git fetch --all --tags
  - |
    if git merge-base --is-ancestor "$CI_COMMIT_SHA" "origin/main"; then
       echo "Tag $CI_COMMIT_TAG belongs to an acceptable branch: main"
       exit 0
    else
       echo "Tag $CI_COMMIT_TAG does not belong to any acceptable branch."
       exit 1
    fi

build:
 stage: build
 image:
  name: docker:24.0.5  # Use a Docker image that supports CLI commands
  entrypoint: [""]
 services:
  - docker:24.0.5-dind
 rules:
  - if: '$CI_COMMIT_TAG =~ /^v\d+\.\d+\.\d+$/'
 before_script:
  - apk add --no-cache curl python3 py3-pip  # Install dependencies
 script:
  # Verify tools
  - docker --version

  # Build Docker image
  - echo "Building Docker image with tag - $CI_COMMIT_TAG"
  - docker build -t $DOCKER_REGISTRY/$APP_NAME:$CI_COMMIT_TAG .
  - docker save $DOCKER_REGISTRY/$APP_NAME:$CI_COMMIT_TAG > image.tar
 artifacts:
  paths:
    - image.tar

upload:
 stage: upload
 image:
  name: docker:24.0.5  # Use a Docker image that supports CLI commands
  entrypoint: [""]
 services:
  - docker:24.0.5-dind
 rules:
  - if: '$CI_COMMIT_TAG =~ /^v\d+\.\d+\.\d+$/'
 before_script:
  - apk add --no-cache curl python3 py3-pip  # Install dependencies
  - pip3 install awscli                      # Install AWS CLI
 script:
  # Verify tools
  - aws --version
  - docker --version

  - echo "Loading Docker image with tag - $CI_COMMIT_TAG"
  - docker load < image.tar

  # Authenticate with AWS CLI
  - aws configure set aws_access_key_id $AWS_ACCESS_KEY_ID
  - aws configure set aws_secret_access_key $AWS_SECRET_ACCESS_KEY
  - aws configure set region $AWS_DEFAULT_REGION

  # Login to AWS ECR
```

```
  - aws ecr get-login-password --region $AWS_DEFAULT_REGION | docker login --username AWS --password-stdin
$DOCKER_REGISTRY

  # Push Docker image
  - docker push $DOCKER_REGISTRY/$APP_NAME:$CI_COMMIT_TAG

release:
 stage: release
 image: registry.gitlab.com/gitlab-org/release-cli:latest
 rules:
  - if: '$CI_COMMIT_TAG =~ /^v\d+\.\d+\.\d+$/'
 script:
  - echo "running release_job for $CI_COMMIT_TAG"
 release:
  name: 'Release $CI_COMMIT_TAG'
  description: 'Created the release for tag $CI_COMMIT_TAG'
  tag_name: '$CI_COMMIT_TAG'

// CD Pipeline
val-vars:
 stage: val-vars
 rules:
  - if: '($CI_PIPELINE_SOURCE == "trigger" || $CI_PIPELINE_SOURCE == "web") && $CI_COMMIT_TAG =~
/^v\d+\.\d+\.\d+$/ && $ACTION == "DEPLOY"'
 script:
  - echo "Validating required variables..."
  - |
    # List of required variables
    declare -A REQUIRED_VARS=(
      ["AWS_ACCESS_KEY_ID"]="AWS Access Key ID"
      ["AWS_SECRET_ACCESS_KEY"]="AWS Secret Access Key"
      ["ECR_REGISTRY"]="ECR Registry URL"
      ["COMMON_INFRA_REGION"]="Common Infrastructure Region"
      ["HOSTED_ZONE_ID"]="Route53 Hosted Zone ID"
      ["INFRA"]="Infrastructure Name"
      ["REPO_NAME"]="ECR Repository Name"
      ["REGION"]="AWS Region"
      ["APP_VERSION"]="Application Version"
      ["DOMAIN_NAME"]="Domain Name"
      ["CP_DOMAIN"]="Control Plane Domain"
      ["DB_NAME"]="DynamoDB Table Name"
      ["S3_BUCKET"]="S3 Bucket Name"
    )

    # Validate presence of all required variables
    for var in "${!REQUIRED_VARS[@]}"; do
      if [ -z "${!var}" ]; then
        echo "Error: ${REQUIRED_VARS[$var]} ($var) is required"
        exit 1
      fi
    done

    # Validate APP_VERSION format
    if ! [[ "$APP_VERSION" =~ ^v[0-9]+\.[0-9]+\.[0-9]+$ ]]; then
      echo "Error: APP_VERSION must match format 'vX.X.X'. Provided: $APP_VERSION"
      exit 1
    fi

    # Mask sensitive variables in logs
    echo "All required variables are present"
    echo "ECR_REGISTRY=$ECR_REGISTRY"
    echo "COMMON_INFRA_REGION=$COMMON_INFRA_REGION"
    echo "INFRA=$INFRA"
    echo "REPO_NAME=$REPO_NAME"
    echo "DOMAIN_NAME=$DOMAIN_NAME"
    echo "CP_DOMAIN=$CP_DOMAIN"
    echo "REGION=$REGION"
    echo "APP_VERSION=$APP_VERSION"
    echo "DB_NAME=$DB_NAME"
    echo "S3_BUCKET=$S3_BUCKET"

    # Only show first few characters of sensitive values
    echo "AWS_ACCESS_KEY_ID=${AWS_ACCESS_KEY_ID:0:4}..."
    echo "AWS_SECRET_ACCESS_KEY=${AWS_SECRET_ACCESS_KEY:0:4}..."

    echo "All variables validated successfully."
```

```
val-ecr-img:
 stage: val-ecr-img
 image:
  name: docker:24.0.5 # Use a Docker image that supports CLI commands
  entrypoint: [""]
 services:
  - name: docker:24.0.5-dind
 variables:
  DOCKER_HOST: tcp://localhost:2375
 rules:
  - if: '($CI_PIPELINE_SOURCE == "trigger" || $CI_PIPELINE_SOURCE == "web") && $CI_COMMIT_TAG =~
/^v\d+\.\d+\.\d+$/ && $ACTION == "DEPLOY"'
 before_script:
  - apk add --no-cache curl python3 py3-pip # Install dependencies
  - pip3 install awscli # Install AWS CLI
 script:
  # Verify required tools
  - aws --version
  - docker --version

  - echo "Validating ECR image, REPO:$REPO_NAME VERSION:$APP_VERSION"

  - echo "Configuring AWS CLI..."
  - aws configure set aws_access_key_id $AWS_ACCESS_KEY_ID
  - aws configure set aws_secret_access_key $AWS_SECRET_ACCESS_KEY
  - aws configure set region $COMMON_INFRA_REGION

  - |
    IMAGE_URI=$ECR_REGISTRY/$REPO_NAME:$APP_VERSION

    echo "Logging in to AWS ECR in region $COMMON_INFRA_REGION"

    aws ecr get-login-password --region "$COMMON_INFRA_REGION" | docker login --username AWS --password-stdin
"$ECR_REGISTRY"

    echo "Inspecting image manifest for $IMAGE_URI";

    if ! docker manifest inspect "$IMAGE_URI" > /dev/null 2>&1; then
      echo "ECR image $IMAGE_URI does not exist or is unavailable"; exit 1;
    fi

  - echo "ECR image validation completed successfully."

deploy:
 stage: deploy
 image:
  name: docker:24.0.5
  entrypoint: [""]
 services:
  - docker:24.0.5-dind
 rules:
  - if: '($CI_PIPELINE_SOURCE == "trigger" || $CI_PIPELINE_SOURCE == "web") && $CI_COMMIT_TAG =~
/^v\d+\.\d+\.\d+$/ && $ACTION == "DEPLOY"'
 before_script:
  - echo "Installing dependencies..."
  - apk update
  - apk add --no-cache git wget unzip bash python3 py3-pip jq
  - pip3 install --no-cache-dir awscli
 script:
  # Terraform installation
  - echo "Installing Terraform ${TF_VERSION}..."
  - wget -q https://releases.hashicorp.com/terraform/${TF_VERSION}/terraform_${TF_VERSION}_linux_amd64.zip
  - unzip -q -d /usr/bin/ terraform_${TF_VERSION}_linux_amd64.zip
  - chmod +x /usr/bin/terraform
  - terraform --version
  - rm terraform_${TF_VERSION}_linux_amd64.zip

  # AWS Configuration
  - echo "Configuring AWS CLI..."
  - aws configure set --profile $AWS_PROFILE aws_access_key_id $AWS_ACCESS_KEY_ID
  - aws configure set --profile $AWS_PROFILE aws_secret_access_key $AWS_SECRET_ACCESS_KEY
  - aws configure set --profile $AWS_PROFILE region $REGION

  # Git operations
  - echo "Checking out Git tag $CI_COMMIT_TAG..."
```

```
  - git fetch --tags --quiet
  - git checkout $CI_COMMIT_TAG --quiet

  # Environment setup
  - echo "Deploying $INFRA:$CI_COMMIT_TAG tenant-manager:$APP_VERSION to $REGION from
REPO:$REPO_NAME"

  # Create variables file for cleaner Terraform commands
  - |
    cat > terraform.tfvars <<EOF
    profile = "$AWS_PROFILE"
    region = "$REGION"
    infra = "$INFRA"
    ecr_repository_name = "$REPO_NAME"
    common_infra_region = "$COMMON_INFRA_REGION"
    ecr_image_tag = "$APP_VERSION"
    zone_id = "$HOSTED_ZONE_ID"
    domain_name = "$DOMAIN_NAME"
    cp_domain = "$CP_DOMAIN"
    dynamodb_table_name = "$DB_NAME"
    s3_bucket_name = "$DB_NAME"
    EOF

  # Terraform operations
  - echo "Initializing Terraform..."
  - terraform init -reconfigure -backend-config="region=$COMMON_INFRA_REGION"
-backend-config="key=$INFRA/$REGION/terraform.tfstate"

  - echo "Planning Terraform changes..."
  - terraform plan -input=false -var-file=terraform.tfvars

  - echo "Applying Terraform changes..."
  - terraform apply -input=false -auto-approve -var-file=terraform.tfvars

  # Output handling
  - echo "Retrieving App Runner service ARN..."
  - |
    SERVICE_ARN=$(terraform output -raw app_runner_service_arn 2>&1)
    if [[ "$SERVICE_ARN" =~ Error|Warning ]]; then
      echo "Terraform output failed: $SERVICE_ARN"
      exit 1
    fi

    echo "Successfully retrieved Service ARN: $SERVICE_ARN"
    echo "SERVICE_ARN=$SERVICE_ARN" > artifact.txt

  # Clean up
  - rm -f terraform.tfvars

 artifacts:
  when: always
  paths:
    - artifact.txt
  expire_in: 1 hour

verify-deploy:
 stage: verify-deploy
 image:
  name: docker:24.0.5 # Use a Docker image that supports CLI commands
  entrypoint: [""]
 services:
  - docker:24.0.5-dind
 rules:
  - if: '($CI_PIPELINE_SOURCE == "trigger" || $CI_PIPELINE_SOURCE == "web") && $CI_COMMIT_TAG =~
/^v\d+\.\d+\.\d+$/ && $ACTION == "DEPLOY"'
 before_script:
  # Install prerequisites
  - echo "Installing dependencies..."
  - apk update
  - apk add --no-cache git python3 py3-pip # Install dependencies
  - pip3 install awscli # Install AWS CLI
 script:
  - echo "Retrieving App Runner service ARN..."
  - |
    if [ -f artifact.txt ]; then
      export SERVICE_ARN=$(awk -F'=' '/^SERVICE_ARN/ {print $2}' artifact.txt)
```

```
      echo "SERVICE_ARN is $SERVICE_ARN"
    fi

    if [ -z "$SERVICE_ARN" ]; then
      echo "Error - SERVICE_ARN is empty"; exit 1;
    fi

  - echo "Validating App Runner service status for ARN - $SERVICE_ARN"

  # Set AWS credentials using environment variables
  - echo "Configuring AWS CLI..."
  - aws configure set --profile $AWS_PROFILE aws_access_key_id $AWS_ACCESS_KEY_ID
  - aws configure set --profile $AWS_PROFILE aws_secret_access_key $AWS_SECRET_ACCESS_KEY
  - aws configure set --profile $AWS_PROFILE region $REGION

  # Check the last deployment status
  - echo "Check last deployment status..."
  - |
    DEPLOYMENT_STATUS=$(aws apprunner list-operations --service-arn "$SERVICE_ARN" --query
"OperationSummaryList[0].Status" --output text)

    if echo "$DEPLOYMENT_STATUS" | grep -qE "Warning|Error"; then
      echo "List operations failed. Error - $(echo "$DEPLOYMENT_STATUS" | grep -o 'Error:.*\|Warning:.*')"; exit 1;
    fi

    if [ -z "$DEPLOYMENT_STATUS" ]; then
      echo "No deployment status found or operations are empty"; exit 1;


    elif [[ "$DEPLOYMENT_STATUS" == "FAILED" || "$DEPLOYMENT_STATUS" == "ROLLBACK_SUCCEEDED" ]];
then
      echo "Deployment failed or rolled back"; exit 1;

    elif [[ "$DEPLOYMENT_STATUS" == "SUCCEEDED" ]]; then
      echo "Deployment succeeded. Current status: $DEPLOYMENT_STATUS";

    else
      echo "Deployment not succeeded: $DEPLOYMENT_STATUS"; exit 1;
    fi

  - echo "Validating App Runner service status..."
  - |
    SERVICE_STATUS=$(aws apprunner describe-service --service-arn "$SERVICE_ARN" --query "Service.Status" --output
text)

    if echo "$SERVICE_STATUS" | grep -qE "Warning|Error"; then
      echo "Describe service failed. Error - $(echo "$SERVICE_STATUS" | grep -o 'Error:.*\|Warning:.*')"; exit 1;
    fi

    if [[ "$SERVICE_STATUS" == "RUNNING" || "$SERVICE_STATUS" == "PAUSED" ]]; then
      echo "Validation succeeded. Current status: $SERVICE_STATUS";

    else
      echo "Validation failed. Current status: $SERVICE_STATUS"; exit 1;
    fi

  - echo "Deploy pipeline succeeded"

 dependencies:
  - deploy # Ensure artifacts from deploy stage are available here
```

Appendix - G - Test Dataset Generation Code

```
package main

import (
  "encoding/csv"
  "flag"
  "fmt"
  "os"
  "strings"

  "github.com/brianvoe/gofakeit/v6"
)

type Business struct {
  Name        string
  Alias       string
  Type        string
  CountryCode string
  Description string
  FirstName   string
  LastName    string
  Email       string
  Password    string
}

func main() {
  // Accept country code as a flag
  countryCode := flag.String("code", "IND", "Country code for businesses")
  count := flag.Int("count", 500, "Number of businesses to generate")
  flag.Parse()

  gofakeit.Seed(0)

  fileName := fmt.Sprintf("businesses_%s.csv", *countryCode)
  file, err := os.Create(fileName)
  if err != nil {
    panic(err)
  }
  defer file.Close()

  writer := csv.NewWriter(file)
  defer writer.Flush()

  // CSV header
  writer.Write([]string{
    "name", "alias", "type", "country_code", "description",
    "first_name", "last_name", "email", "password",
  })

  usedAliases := make(map[string]bool)
  usedEmails := make(map[string]bool)

  for i := 1; i <= *count; i++ {
    name := fmt.Sprintf("Test %s Buss %d", *countryCode, i)
    alias := fmt.Sprintf("uq0-test-%s-buss-%d", strings.ToLower(*countryCode), i)
    description := fmt.Sprintf("%s Business for testing %d", *countryCode, i)

    // Ensure unique alias
    if usedAliases[alias] {
      continue
    }
    usedAliases[alias] = true

    first := gofakeit.FirstName()
    last := gofakeit.LastName()
    email := fmt.Sprintf("uq0.%s.%s.%s%d@example.com", strings.ToLower(*countryCode), first, last, i)

    // Ensure unique email
    if usedEmails[email] {
      continue
    }
    usedEmails[email] = true
```

```
        b := Business{
            Name:        name,
            Alias:       alias,
            Type:        "PRODUCT",
            CountryCode: *countryCode,
            Description: description,
            FirstName:   first,
            LastName:    last,
            Email:       email,
            Password:    "Test@1234",
        }

        writer.Write([]string{
            b.Name,
            b.Alias,
            b.Type,
            b.CountryCode,
            b.Description,
            b.FirstName,
            b.LastName,
            b.Email,
            b.Password,
        })
    }

    fmt.Printf("✅ CSV file '%s' created successfully with %d entries!\n", fileName, *count)
}
```

Appendix - H - Part of Test Dataset

// Australia
name,alias,type,country_code,description,first_name,last_name,email,password
Test AUS Buss 1,test-aus-buss-1,PRODUCT,AUS,AUS Business for testing
1,Izabella,Schmeler,aus.Izabella.Schmeler1@example.com,Test@1234
Test AUS Buss 2,test-aus-buss-2,PRODUCT,AUS,AUS Business for testing 2,April,Mraz,aus.April.Mraz2@example.com,Test@1234
Test AUS Buss 3,test-aus-buss-3,PRODUCT,AUS,AUS Business for testing 3,Felicita,Klocko,aus.Felicita.Klocko3@example.com,Test@1234
Test AUS Buss 4,test-aus-buss-4,PRODUCT,AUS,AUS Business for testing 4,Valerie,Cole,aus.Valerie.Cole4@example.com,Test@1234
Test AUS Buss 5,test-aus-buss-5,PRODUCT,AUS,AUS Business for testing 5,Fae,Bernier,aus.Fae.Bernier5@example.com,Test@1234
Test AUS Buss 6,test-aus-buss-6,PRODUCT,AUS,AUS Business for testing 6,Nathen,Bernier,aus.Nathen.Bernier6@example.com,Test@1234
Test AUS Buss 7,test-aus-buss-7,PRODUCT,AUS,AUS Business for testing 7,Ressie,Harvey,aus.Ressie.Harvey7@example.com,Test@1234
Test AUS Buss 8,test-aus-buss-8,PRODUCT,AUS,AUS Business for testing 8,Yazmin,Willms,aus.Yazmin.Willms8@example.com,Test@1234
Test AUS Buss 9,test-aus-buss-9,PRODUCT,AUS,AUS Business for testing 9,Rocky,Lubowitz,aus.Rocky.Lubowitz9@example.com,Test@1234
Test AUS Buss 10,test-aus-buss-10,PRODUCT,AUS,AUS Business for testing 10,Maye,Harber,aus.Maye.Harber10@example.com,Test@1234

// Canada
name,alias,type,country_code,description,first_name,last_name,email,password
Test CA Buss 1,test-ca-buss-1,PRODUCT,CA,CA Business for testing 1,Alek,Stroman,ca.Alek.Stroman1@example.com,Test@1234
Test CA Buss 2,test-ca-buss-2,PRODUCT,CA,CA Business for testing 2,Ransom,Wolf,ca.Ransom.Wolf2@example.com,Test@1234
Test CA Buss 3,test-ca-buss-3,PRODUCT,CA,CA Business for testing 3,Holly,Dibbert,ca.Holly.Dibbert3@example.com,Test@1234
Test CA Buss 4,test-ca-buss-4,PRODUCT,CA,CA Business for testing 4,Elisa,Blick,ca.Elisa.Blick4@example.com,Test@1234
Test CA Buss 5,test-ca-buss-5,PRODUCT,CA,CA Business for testing 5,Eddie,Monahan,ca.Eddie.Monahan5@example.com,Test@1234
Test CA Buss 6,test-ca-buss-6,PRODUCT,CA,CA Business for testing 6,May,Powlowski,ca.May.Powlowski6@example.com,Test@1234
Test CA Buss 7,test-ca-buss-7,PRODUCT,CA,CA Business for testing 7,Ollie,Bogan,ca.Ollie.Bogan7@example.com,Test@1234
Test CA Buss 8,test-ca-buss-8,PRODUCT,CA,CA Business for testing 8,Andres,Shields,ca.Andres.Shields8@example.com,Test@1234
Test CA Buss 9,test-ca-buss-9,PRODUCT,CA,CA Business for testing 9,Jaylan,Wilkinson,ca.Jaylan.Wilkinson9@example.com,Test@1234
Test CA Buss 10,test-ca-buss-10,PRODUCT,CA,CA Business for testing 10,Brock,Nicolas,ca.Brock.Nicolas10@example.com,Test@1234

// Malaysia
name,alias,type,country_code,description,first_name,last_name,email,password
Test MY Buss 1,test-my-buss-1,PRODUCT,MY,MY Business for testing 1,Mekhi,Goldner,my.Mekhi.Goldner1@example.com,Test@1234
Test MY Buss 2,test-my-buss-2,PRODUCT,MY,MY Business for testing 2,Garland,Runte,my.Garland.Runte2@example.com,Test@1234
Test MY Buss 3,test-my-buss-3,PRODUCT,MY,MY Business for testing 3,Nicole,Maggio,my.Nicole.Maggio3@example.com,Test@1234
Test MY Buss 4,test-my-buss-4,PRODUCT,MY,MY Business for testing 4,Eino,Steuber,my.Eino.Steuber4@example.com,Test@1234
Test MY Buss 5,test-my-buss-5,PRODUCT,MY,MY Business for testing 5,Kara,Beatty,my.Kara.Beatty5@example.com,Test@1234
Test MY Buss 6,test-my-buss-6,PRODUCT,MY,MY Business for testing 6,Carrie,Dooley,my.Carrie.Dooley6@example.com,Test@1234
Test MY Buss 7,test-my-buss-7,PRODUCT,MY,MY Business for testing 7,Zachery,Harris,my.Zachery.Harris7@example.com,Test@1234
Test MY Buss 8,test-my-buss-8,PRODUCT,MY,MY Business for testing 8,Baylee,Jaskolski,my.Baylee.Jaskolski8@example.com,Test@1234
Test MY Buss 9,test-my-buss-9,PRODUCT,MY,MY Business for testing 9,Ophelia,Lueilwitz,my.Ophelia.Lueilwitz9@example.com,Test@1234
Test MY Buss 10,test-my-buss-10,PRODUCT,MY,MY Business for testing 10,Mathias,Glover,my.Mathias.Glover10@example.com,Test@1234

// Sri Lanka
name,alias,type,country_code,description,first_name,last_name,email,password
Test LK Buss 1,test-lk-buss-1,PRODUCT,LK,LK Business for testing 1,Claudine,Mertz,lk.Claudine.Mertz1@example.com,Test@1234
Test LK Buss 2,test-lk-buss-2,PRODUCT,LK,LK Business for testing 2,Carey,Mraz,lk.Carey.Mraz2@example.com,Test@1234
Test LK Buss 3,test-lk-buss-3,PRODUCT,LK,LK Business for testing 3,Raymundo,Kuphal,lk.Raymundo.Kuphal3@example.com,Test@1234
Test LK Buss 4,test-lk-buss-4,PRODUCT,LK,LK Business for testing 4,Estella,Connelly,lk.Estella.Connelly4@example.com,Test@1234
Test LK Buss 5,test-lk-buss-5,PRODUCT,LK,LK Business for testing 5,Louie,Hettinger,lk.Louie.Hettinger5@example.com,Test@1234
Test LK Buss 6,test-lk-buss-6,PRODUCT,LK,LK Business for testing 6,Colby,Anderson,lk.Colby.Anderson6@example.com,Test@1234
Test LK Buss 7,test-lk-buss-7,PRODUCT,LK,LK Business for testing 7,Brenden,Ebert,lk.Brenden.Ebert7@example.com,Test@1234
Test LK Buss 8,test-lk-buss-8,PRODUCT,LK,LK Business for testing 8,Estell,Gorczany,lk.Estell.Gorczany8@example.com,Test@1234
Test LK Buss 9,test-lk-buss-9,PRODUCT,LK,LK Business for testing 9,Tyson,Dare,lk.Tyson.Dare9@example.com,Test@1234
Test LK Buss 10,test-lk-buss-10,PRODUCT,LK,LK Business for testing 10,Anabelle,Denesik,lk.Anabelle.Denesik10@example.com,Test@1234

// France
name,alias,type,country_code,description,first_name,last_name,email,password
Test FR Buss 1,test-fr-buss-1,PRODUCT,FR,FR Business for testing 1,Gerald,Quigley,fr.Gerald.Quigley1@example.com,Test@1234
Test FR Buss 2,test-fr-buss-2,PRODUCT,FR,FR Business for testing 2,Allan,Schneider,fr.Allan.Schneider2@example.com,Test@1234
Test FR Buss 3,test-fr-buss-3,PRODUCT,FR,FR Business for testing 3,Cade,Fahey,fr.Cade.Fahey3@example.com,Test@1234
Test FR Buss 4,test-fr-buss-4,PRODUCT,FR,FR Business for testing 4,Hans,Pfannerstill,fr.Hans.Pfannerstill4@example.com,Test@1234
Test FR Buss 5,test-fr-buss-5,PRODUCT,FR,FR Business for testing 5,Dorian,Ondricka,fr.Dorian.Ondricka5@example.com,Test@1234
Test FR Buss 6,test-fr-buss-6,PRODUCT,FR,FR Business for testing 6,Kameron,Grimes,fr.Kameron.Grimes6@example.com,Test@1234
Test FR Buss 7,test-fr-buss-7,PRODUCT,FR,FR Business for testing 7,Elisa,Littel,fr.Elisa.Littel7@example.com,Test@1234
Test FR Buss 8,test-fr-buss-8,PRODUCT,FR,FR Business for testing 8,Kenton,McClure,fr.Kenton.McClure8@example.com,Test@1234
Test FR Buss 9,test-fr-buss-9,PRODUCT,FR,FR Business for testing 9,Madaline,Bailey,fr.Madaline.Bailey9@example.com,Test@1234
Test FR Buss 10,test-fr-buss-10,PRODUCT,FR,FR Business for testing 10,Oceane,Walsh,fr.Oceane.Walsh10@example.com,Test@1234

Appendix - I - Automated Testing Script

```
package mai
const (
  REGISTER = "business-register"
  ADMIN_LOGIN = "admin-login"
  ADMIN_GET_BUSINESS = "admin-get-business"
  CREATE_BUSINESS_USER = "create-business-user"
  BUSINESS_USER_LOGIN = "business-user-login"
  BUSINESS_USER_GET_BUSINESS = "business-user-get-business"
)

type Test struct {
  BusinessName       string
  BusinessAlias      string
  BusinessType       string
  CountryCode        string
  Description        string
  FirstName          string
  LastName           string
  Email              string
  Password           string
  Endpoint           string
  AdminAccessToken   string
  BUserAccessToken   string
  BusinessID         string
}

type Result struct {
  PassedTests        int
  FailedTests        int
  SuccessLatencies   []int64
  CSVRows            [][]string
  Average            int64
  P90                int64
  P95                int64
  P99                int64
}

type User struct {
  FirstName string `json:"first_name"`
  LastName  string `json:"last_name"`
  Email     string `json:"email"`
  Password  string `json:"password"`
}

type Business struct {
  Name        string `json:"name"`
  Alias       string `json:"alias"`
  Type        string `json:"type"`
  CountryCode string `json:"country_code"`
  Description string `json:"description"`
  User        User   `json:"user"`
}

type BusinessUser struct {
  FirstName string `json:"first_name"`
  LastName  string `json:"last_name"`
  Email     string `json:"email"`
  Type      string `json:"type"`
  Password  string `json:"password"`
}

type ApiResponse struct {
  Data     json.RawMessage `json:"data"`
  Error    string          `json:"error"`
  Success  string          `json:"success"`
  Metadata interface{}     `json:"metadata"`
}

type LoginPayload struct {
  Email    string `json:"email"`
  Password string `json:"password"`
  DeviceID string `json:"device_id"`
```

```
  Device   string `json:"device_type"`
}

type BLoginPayload struct {
  BusinessID  string `json:"business_id"`
  Email       string `json:"email"`
  Password    string `json:"password"`
  DeviceID    string `json:"device_id"`
  Device      string `json:"device_type"`
}

type LoginResponse struct {
  AccessToken string `json:"access_token"`
  Endpoint    string `json:"endpoint"`
}

type GetBusineesResponse struct {
  ID      string `json:"id"`
  Alias   string `json:"alias"`
}

type GetInfraResponse struct {
  BusinessID      string `json:"business_id"`
  Alias           string `json:"alias"`
  Endpoint        string `json:"endpoint"`
  Region          string `json:"region"`
}

func main() {
  infra := flag.String("infra", "1CPx1AP", "Target Infra")
  code := flag.String("code", "IND", "Country code")
  region := flag.String("region", "ap-south-1", "AWS region")
  bucket := flag.String("bucket", "test", "S3 bucket name")
  bucketRegion := flag.String("bucket-region", "ap-south-1", "S3 bucket region")
  domain := flag.String("domain", "https://admin.api.sellertower.click", "S3 domain")
  flag.Parse()

  inputFileName := fmt.Sprintf("businesses_%s.csv", *code)

  inputFile, err := os.Open(inputFileName)
  if err != nil {
    panic(fmt.Sprintf("❌ Failed to open input file: %s", inputFileName))
  }
  defer inputFile.Close()

  reader := csv.NewReader(inputFile)
  records, err := reader.ReadAll()
  if err != nil {
    panic(err)
  }

  testCases := []*Test{}

  for i, row := range records {

    if i == 0 {
      continue
    }

    test := &Test{
      BusinessName: row[0],
      BusinessAlias: row[1],
      BusinessType: row[2],
      CountryCode: row[3],
      Description: row[4],
      FirstName: row[5],
      LastName: row[6],
      Email: row[7],
      Password: row[8],
    }

    testCases = append(testCases, test)
  }

  totalTests := len(records) - 1
  fmt.Printf("🏳 Running tests for country: %s\n", *code)
```

```
  fmt.Printf("📍 Running tests for region: %s\n", *region)
  fmt.Printf("🏗️ Running tests for infra: %s\n", *infra)
  fmt.Printf("🧪 Total test cases: %d\n", totalTests)

  CSVHeader := [][]string{
     {"AWS region", *region},
     {"Country code", *code},
     {"Target infra", *infra},
     {"Total Test Cases", fmt.Sprintf("%d", totalTests)},
  }

  responses :=make(map[string]*Result)

  // Run test for register
  fmt.Printf("🔧 Running Test: %s\n", REGISTER)

  responses[REGISTER] = &Result{
     PassedTests: 0,
     FailedTests: 0,
     CSVRows: [][]string{},
     SuccessLatencies: []int64{},
  }

  responses[REGISTER].CSVRows = append(responses[REGISTER].CSVRows, CSVHeader...)
  responses[REGISTER].CSVRows = append(responses[REGISTER].CSVRows, []string{"Test Type", REGISTER})
  responses[REGISTER].CSVRows = append(responses[REGISTER].CSVRows, []string{})
  responses[REGISTER].CSVRows = append(responses[REGISTER].CSVRows, []string{"email", "alias", "http_status_code",
"latency_ms", "response_error", "request_error"})

  for _, test := range testCases {

     var statusCode string
     var responseErr, requestErr string
     var latency int64

     biz := Business{
        Name:       test.BusinessName,
        Alias:      test.BusinessAlias,
        Type:       test.BusinessType,
        CountryCode: test.CountryCode,
        Description: test.Description,
        User: User{
           FirstName: test.FirstName,
           LastName:  test.LastName,
           Email:     test.Email,
           Password:  test.Password,
        },
     }

     start := time.Now()

     jsonData, _ := json.Marshal(biz)
     resp, err := http.Post(fmt.Sprintf("%s/api/buss/verify-buss", *domain), "application/json", bytes.NewBuffer(jsonData))
     latency = time.Since(start).Milliseconds()

     if err != nil {
        statusCode = "0"
        requestErr = err.Error()
        responses[REGISTER].FailedTests++
        fmt.Printf("[%s] 🔴 FAIL | alias: %-15s | Error: %s\n", REGISTER, test.BusinessAlias, requestErr)
     } else {
        defer resp.Body.Close()
        statusCode = fmt.Sprintf("%d", resp.StatusCode)
        if resp.StatusCode == 201 {
           responses[REGISTER].PassedTests++
           responses[REGISTER].SuccessLatencies = append(responses[REGISTER].SuccessLatencies, latency)
           // fmt.Printf("[%s] 🟢 PASS | alias: %-15s | Status: %s | Latency: %dms\n", REGISTER, test.BusinessAlias,
statusCode, latency)
        } else {
           body, _ := io.ReadAll(resp.Body)
           var apiResp ApiResponse
           json.Unmarshal(body, &apiResp)
           responseErr = apiResp.Error
           responses[REGISTER].FailedTests++
           fmt.Printf("[%s] 🔴 FAIL | alias: %-15s | Status: %s | Latency: %dms | Error: %s\n", REGISTER, test.BusinessAlias,
statusCode, latency, responseErr)
```

```
        }
    }

    responses[REGISTER].CSVRows = append(responses[REGISTER].CSVRows, []string{
        test.Email,
        test.BusinessAlias,
        statusCode,
        fmt.Sprintf("%d", latency),
        responseErr,
        requestErr,
    })
}

// Result of register
calculateSummary(REGISTER, responses[REGISTER])

// Run test for admin login
fmt.Printf("🔧 Running Test: %s\n", ADMIN_LOGIN)

responses[ADMIN_LOGIN] = &Result{
    PassedTests: 0,
    FailedTests: 0,
    CSVRows: [][]string{},
    SuccessLatencies: []int64{},
}

responses[ADMIN_LOGIN].CSVRows = append(responses[ADMIN_LOGIN].CSVRows, CSVHeader...)
responses[ADMIN_LOGIN].CSVRows = append(responses[ADMIN_LOGIN].CSVRows, []string{"Test Type",
ADMIN_LOGIN})
responses[ADMIN_LOGIN].CSVRows = append(responses[ADMIN_LOGIN].CSVRows, []string{})
responses[ADMIN_LOGIN].CSVRows = append(responses[ADMIN_LOGIN].CSVRows, []string{"email", "alias",
"http_status_code", "latency_ms", "response_error", "request_error"})

for i, test := range testCases {

    var statusCode string
    var responseErr, requestErr string
    var latency int64

    loginData := LoginPayload{
        Email:    test.Email,
        Password: test.Password,
        DeviceID: fmt.Sprintf("test-device-%s-%d", strings.ToLower(test.CountryCode), i),
        Device:   "cli",
    }

    start := time.Now()

    jsonData, _ := json.Marshal(loginData)
    resp, err := http.Post(fmt.Sprintf("%s/api/auth/login", *domain), "application/json", bytes.NewBuffer(jsonData))
    latency = time.Since(start).Milliseconds()

    if err != nil {
        statusCode = "0"
        requestErr = err.Error()
        responses[ADMIN_LOGIN].FailedTests++
        fmt.Printf("[%s] 🔴 FAIL | alias: %-15s | email: %-20s | Error: %s\n", ADMIN_LOGIN, test.BusinessAlias, test.Email,
requestErr)
    } else {
        defer resp.Body.Close()
        body, _ := io.ReadAll(resp.Body)
        statusCode = fmt.Sprintf("%d", resp.StatusCode)

        if resp.StatusCode == 201 {
            var apiResp ApiResponse
            json.Unmarshal(body, &apiResp)
            var loginResp LoginResponse
            json.Unmarshal(apiResp.Data, &loginResp)

            responses[ADMIN_LOGIN].PassedTests++
            responses[ADMIN_LOGIN].SuccessLatencies = append(responses[ADMIN_LOGIN].SuccessLatencies, latency)
            // fmt.Printf("[%s] 🟢 PASS | alias: %-15s | email: %-20s | Status: %s | Latency: %dms\n",  ADMIN_LOGIN,
test.BusinessAlias, test.Email, statusCode, latency)

            (*test).Endpoint = loginResp.Endpoint
            (*test).AdminAccessToken = loginResp.AccessToken
```

```
        } else {
            var apiResp ApiResponse
            json.Unmarshal(body, &apiResp)
            responseErr = apiResp.Error
            responses[ADMIN_LOGIN].FailedTests++
            fmt.Printf("[%s] 🔴 FAIL | alias: %-15s | email: %-20s | Status: %s | Latency: %dms | Error: %s\n", ADMIN_LOGIN,
test.BusinessAlias, test.Email, statusCode, latency, responseErr)
        }
    }

    responses[ADMIN_LOGIN].CSVRows = append(responses[ADMIN_LOGIN].CSVRows, []string{
        test.Email,
        test.BusinessAlias,
        statusCode,
        fmt.Sprintf("%d", latency),
        responseErr,
        requestErr,
    })
}

// Result of admin login
calculateSummary(ADMIN_LOGIN, responses[ADMIN_LOGIN])

// Run test for admin get business
fmt.Printf("🔧 Running Test: %s\n", ADMIN_GET_BUSINESS)

responses[ADMIN_GET_BUSINESS] = &Result{
    PassedTests: 0,
    FailedTests: 0,
    CSVRows: [][]string{},
    SuccessLatencies: []int64{},
}

responses[ADMIN_GET_BUSINESS].CSVRows = append(responses[ADMIN_GET_BUSINESS].CSVRows,
CSVHeader...)
responses[ADMIN_GET_BUSINESS].CSVRows = append(responses[ADMIN_GET_BUSINESS].CSVRows, []string{"Test
Type", ADMIN_GET_BUSINESS})
responses[ADMIN_GET_BUSINESS].CSVRows = append(responses[ADMIN_GET_BUSINESS].CSVRows, []string{})
responses[ADMIN_GET_BUSINESS].CSVRows = append(responses[ADMIN_GET_BUSINESS].CSVRows,
[]string{"email", "alias", "http_status_code", "latency_ms", "response_error", "request_error"})

for _, test := range testCases {

    var statusCode string
    var responseErr, requestErr string
    var latency int64

    // Call business details using token
    client := &http.Client{
        Timeout: 5 * time.Second,
    }

    start := time.Now()
    req, _ := http.NewRequest("GET", test.Endpoint+"/api/buss/get", nil)
    req.Header.Set("Authorization", "Bearer "+test.AdminAccessToken)

    resp, err := client.Do(req)
    latency = time.Since(start).Milliseconds()

    if err != nil {
        statusCode = "0"
        requestErr = err.Error()
        responses[ADMIN_GET_BUSINESS].FailedTests++
        fmt.Printf("[%s] 🔴 FAIL | alias: %-15s | email: %-20s | Error: %s\n", ADMIN_GET_BUSINESS, test.BusinessAlias,
test.Email, requestErr)
    } else {
        defer resp.Body.Close()
        body, _ := io.ReadAll(resp.Body)
        statusCode = fmt.Sprintf("%d", resp.StatusCode)

        if resp.StatusCode == 200 {
            var apiResp ApiResponse
            json.Unmarshal(body, &apiResp)
            var res GetBusineesResponse
            json.Unmarshal(apiResp.Data, &res)
```

```
            responses[ADMIN_GET_BUSINESS].PassedTests++
            responses[ADMIN_GET_BUSINESS].SuccessLatencies =
append(responses[ADMIN_GET_BUSINESS].SuccessLatencies, latency)
            // fmt.Printf("[%s] 🟢 PASS | alias: %-15s | email: %-20s | Status: %s | Latency: %dms\n",
ADMIN_GET_BUSINESS, test.BusinessAlias, test.Email, statusCode, latency)

            (*test).BusinessID = res.ID
        } else {
            responseErr = fmt.Sprintf("biz fetch failed %s", statusCode)
            responses[ADMIN_GET_BUSINESS].FailedTests++
            fmt.Printf("[%s] 🔴 FAIL | alias: %-15s | email: %-20s | Status: %s | Error: %s\n", ADMIN_GET_BUSINESS,
test.BusinessAlias, test.Email, statusCode, responseErr)
        }
    }

    responses[ADMIN_GET_BUSINESS].CSVRows = append(responses[ADMIN_GET_BUSINESS].CSVRows, []string{
        test.Email,
        test.BusinessAlias,
        statusCode,
        fmt.Sprintf("%d", latency),
        responseErr,
        requestErr,
    })

}

// Result of admin get business
calculateSummary(ADMIN_GET_BUSINESS, responses[ADMIN_GET_BUSINESS])

// Run test for create business user
fmt.Printf("🔧 Running Test: %s\n", CREATE_BUSINESS_USER)

responses[CREATE_BUSINESS_USER] = &Result{
    PassedTests: 0,
    FailedTests: 0,
    CSVRows: [][]string{},
    SuccessLatencies: []int64{},
}

responses[CREATE_BUSINESS_USER].CSVRows = append(responses[CREATE_BUSINESS_USER].CSVRows,
CSVHeader...)
responses[CREATE_BUSINESS_USER].CSVRows = append(responses[CREATE_BUSINESS_USER].CSVRows,
[]string{"Test Type", CREATE_BUSINESS_USER})
responses[CREATE_BUSINESS_USER].CSVRows = append(responses[CREATE_BUSINESS_USER].CSVRows,
[]string{})
responses[CREATE_BUSINESS_USER].CSVRows = append(responses[CREATE_BUSINESS_USER].CSVRows,
[]string{"email", "alias", "http_status_code", "latency_ms", "response_error", "request_error"})

for _, test := range testCases {

    var statusCode string
    var responseErr, requestErr string
    var latency int64

    user := BusinessUser{
        FirstName: test.FirstName,
        LastName:  test.LastName,
        Email:     test.Email,
        Password:  test.Password,
        Type: "BUSINESS_USER",
    }

    // Call business details using token
    client := &http.Client{
        Timeout: 5 * time.Second,
    }

    start := time.Now()

    jsonData, _ := json.Marshal(user)
    req, _ := http.NewRequest("POST", fmt.Sprintf("%s/api/user/%s/create", test.Endpoint, test.BusinessID),
bytes.NewBuffer(jsonData))

    req.Header.Set("Authorization", "Bearer "+test.AdminAccessToken)
    req.Header.Set("Content-Type", "application/json")
```

```
    resp, err := client.Do(req)
    latency = time.Since(start).Milliseconds()

    if err != nil {
      statusCode = "0"
      requestErr = err.Error()
      responses[CREATE_BUSINESS_USER].FailedTests++
      fmt.Printf("[%s] 🔴 FAIL | alias: %-15s | Error: %s\n", CREATE_BUSINESS_USER, test.BusinessAlias, requestErr)
    } else {
      defer resp.Body.Close()
      statusCode = fmt.Sprintf("%d", resp.StatusCode)
      if resp.StatusCode == 201 {
        responses[CREATE_BUSINESS_USER].PassedTests++
        responses[CREATE_BUSINESS_USER].SuccessLatencies =
append(responses[CREATE_BUSINESS_USER].SuccessLatencies, latency)
        // fmt.Printf("[%s] 🟢 PASS | alias: %-15s | Status: %s | Latency: %dms\n", CREATE_BUSINESS_USER,
test.BusinessAlias, statusCode, latency)
      } else {
        body, _ := io.ReadAll(resp.Body)
        var apiResp ApiResponse
        json.Unmarshal(body, &apiResp)
        responseErr = apiResp.Error
        responses[CREATE_BUSINESS_USER].FailedTests++
        fmt.Printf("[%s] 🔴 FAIL | alias: %-15s | Status: %s | Latency: %dms | Error: %s\n", CREATE_BUSINESS_USER,
test.BusinessAlias, statusCode, latency, responseErr)
      }
    }

    responses[CREATE_BUSINESS_USER].CSVRows = append(responses[CREATE_BUSINESS_USER].CSVRows,
[]string{
      test.Email,
      test.BusinessAlias,
      statusCode,
      fmt.Sprintf("%d", latency),
      responseErr,
      requestErr,
    })
  }

  // Result of create business user
  calculateSummary(CREATE_BUSINESS_USER, responses[CREATE_BUSINESS_USER])

  // Run test for business user login
  fmt.Printf("🔧 Running Test: %s\n", BUSINESS_USER_LOGIN)

  responses[BUSINESS_USER_LOGIN] = &Result{
    PassedTests: 0,
    FailedTests: 0,
    CSVRows: [][]string{},
    SuccessLatencies: []int64{},
  }

  responses[BUSINESS_USER_LOGIN].CSVRows = append(responses[BUSINESS_USER_LOGIN].CSVRows,
CSVHeader...)
  responses[BUSINESS_USER_LOGIN].CSVRows = append(responses[BUSINESS_USER_LOGIN].CSVRows,
[]string{"Test Type", BUSINESS_USER_LOGIN})
  responses[BUSINESS_USER_LOGIN].CSVRows = append(responses[BUSINESS_USER_LOGIN].CSVRows, []string{})
  responses[BUSINESS_USER_LOGIN].CSVRows = append(responses[BUSINESS_USER_LOGIN].CSVRows,
[]string{"email", "alias", "http_status_code", "latency_ms", "response_error", "request_error"})

  for i, test := range testCases {

    var statusCode string
    var responseErr, requestErr string
    var latency int64

    // Call business details using token
    client := &http.Client{
      Timeout: 5 * time.Second,
    }

    start := time.Now()
    req, _ := http.NewRequest("GET", fmt.Sprintf("%s/api/buss/infra/%s", *domain, test.BusinessAlias), nil)
    resp, err := client.Do(req)
```

```
    if err != nil {
        statusCode = "0"
        requestErr = err.Error()
        responses[BUSINESS_USER_LOGIN].FailedTests++
        fmt.Printf("[%s][get-infra] 🔴 FAIL | alias: %-15s | Error: %s\n", BUSINESS_USER_LOGIN, test.BusinessAlias,
requestErr)
    } else {
        defer resp.Body.Close()
        body, _ := io.ReadAll(resp.Body)
        statusCode = fmt.Sprintf("%d", resp.StatusCode)

        if resp.StatusCode == 200 {
            var apiResp ApiResponse
            json.Unmarshal(body, &apiResp)
            var res GetInfraResponse
            json.Unmarshal(apiResp.Data, &res)

            fmt.Printf("[%s][get-infra] 🔵 INFRA-REGION | business-alias: %-20s | region: %-12s | endpoint: %57s\n",
BUSINESS_USER_LOGIN, test.BusinessAlias, res.Region, res.Endpoint)

            (*test).Endpoint = res.Endpoint

            loginData := BLoginPayload{
                BusinessID: res.BusinessID,
                Email:    test.Email,
                Password: test.Password,
                DeviceID: fmt.Sprintf("test-device-%s-%d", strings.ToLower(test.CountryCode), i),
                Device:   "cli",
            }

            jsonData, _ := json.Marshal(loginData)
            resp, err := http.Post(fmt.Sprintf("%s/api/auth/login", res.Endpoint), "application/json", bytes.NewBuffer(jsonData))
            latency = time.Since(start).Milliseconds()

            if err != nil {
                statusCode = "0"
                requestErr = err.Error()
                responses[BUSINESS_USER_LOGIN].FailedTests++
                fmt.Printf("[%s][login] 🔴 FAIL | alias: %-15s | email: %-20s | Error: %s\n", BUSINESS_USER_LOGIN,
test.BusinessAlias, test.Email, requestErr)
            } else {
                defer resp.Body.Close()
                body, _ := io.ReadAll(resp.Body)
                statusCode = fmt.Sprintf("%d", resp.StatusCode)

                if resp.StatusCode == 201 {
                    var apiResp ApiResponse
                    json.Unmarshal(body, &apiResp)
                    var loginResp LoginResponse
                    json.Unmarshal(apiResp.Data, &loginResp)

                    responses[BUSINESS_USER_LOGIN].PassedTests++
                    responses[BUSINESS_USER_LOGIN].SuccessLatencies =
append(responses[BUSINESS_USER_LOGIN].SuccessLatencies, latency)
                    // fmt.Printf("[%s][login] 🟢 PASS | alias: %-15s | email: %-20s | Status: %s | Latency: %dms\n",
BUSINESS_USER_LOGIN, test.BusinessAlias, test.Email, statusCode, latency)

                    (*test).Endpoint = res.Endpoint
                    (*test).BUserAccessToken = loginResp.AccessToken

                } else {
                    var apiResp ApiResponse
                    json.Unmarshal(body, &apiResp)
                    responseErr = apiResp.Error
                    responses[BUSINESS_USER_LOGIN].FailedTests++
                    fmt.Printf("[%s][login] 🔴 FAIL | alias: %-15s | email: %-20s | Status: %s | Latency: %dms | Error: %s\n",
BUSINESS_USER_LOGIN, test.BusinessAlias, test.Email, statusCode, latency, responseErr)
                }
            }

        } else {
            responseErr = fmt.Sprintf("infra fetch failed %s", statusCode)
            responses[BUSINESS_USER_LOGIN].FailedTests++
            fmt.Printf("[%s][get-infra] 🔴 FAIL | alias: %-15s | Status: %s | Error: %s\n", BUSINESS_USER_LOGIN,
test.BusinessAlias, statusCode, responseErr)
        }
```

```
    }

    responses[BUSINESS_USER_LOGIN].CSVRows = append(responses[BUSINESS_USER_LOGIN].CSVRows, []string{
        test.Email,
        test.BusinessAlias,
        statusCode,
        fmt.Sprintf("%d", latency),
        responseErr,
        requestErr,
    })
  }

  // Result of business user login
  calculateSummary(BUSINESS_USER_LOGIN, responses[BUSINESS_USER_LOGIN])

  // Run test for business user get business
  fmt.Printf("🔧 Running Test: %s\n", BUSINESS_USER_GET_BUSINESS)

  responses[BUSINESS_USER_GET_BUSINESS] = &Result{
    PassedTests: 0,
    FailedTests: 0,
    CSVRows: [][]string{},
    SuccessLatencies: []int64{},
  }

  responses[BUSINESS_USER_GET_BUSINESS].CSVRows =
append(responses[BUSINESS_USER_GET_BUSINESS].CSVRows, CSVHeader...)
  responses[BUSINESS_USER_GET_BUSINESS].CSVRows =
append(responses[BUSINESS_USER_GET_BUSINESS].CSVRows, []string{"Test Type",
BUSINESS_USER_GET_BUSINESS})
  responses[BUSINESS_USER_GET_BUSINESS].CSVRows =
append(responses[BUSINESS_USER_GET_BUSINESS].CSVRows, []string{})
  responses[BUSINESS_USER_GET_BUSINESS].CSVRows =
append(responses[BUSINESS_USER_GET_BUSINESS].CSVRows, []string{"email", "alias", "http_status_code",
"latency_ms", "response_error", "request_error"})

  for _, test := range testCases {

    var statusCode string
    var responseErr, requestErr string
    var latency int64

    // Call business details using token
    client := &http.Client{
      Timeout: 5 * time.Second,
    }

    start := time.Now()
    req, _ := http.NewRequest("GET", test.Endpoint+"/api/buss/get", nil)
    req.Header.Set("Authorization", "Bearer "+test.BUserAccessToken)
    resp, err := client.Do(req)
    latency = time.Since(start).Milliseconds()

    if err != nil {
      statusCode = "0"
      requestErr = err.Error()
      responses[BUSINESS_USER_GET_BUSINESS].FailedTests++
      fmt.Printf("[%s] 🔴 FAIL | alias: %-15s | email: %-20s | Error: %s\n", BUSINESS_USER_GET_BUSINESS,
test.BusinessAlias, test.Email, requestErr)
    } else {
      defer resp.Body.Close()
      body, _ := io.ReadAll(resp.Body)
      statusCode = fmt.Sprintf("%d", resp.StatusCode)

      if resp.StatusCode == 200 {
        var apiResp ApiResponse
        json.Unmarshal(body, &apiResp)
        var res GetBusineesResponse
        json.Unmarshal(apiResp.Data, &res)

        responses[BUSINESS_USER_GET_BUSINESS].PassedTests++
        responses[BUSINESS_USER_GET_BUSINESS].SuccessLatencies =
append(responses[BUSINESS_USER_GET_BUSINESS].SuccessLatencies, latency)
        // fmt.Printf("[%s] 🟢 PASS | alias: %-15s | email: %-20s | Status: %s | Latency: %dms\n",
BUSINESS_USER_GET_BUSINESS, test.BusinessAlias, test.Email, statusCode, latency)
```

```
            test.BusinessID = res.ID
        } else {
            responseErr = fmt.Sprintf("biz fetch failed %s", statusCode)
            responses[BUSINESS_USER_GET_BUSINESS].FailedTests++
            fmt.Printf("[%s] 🔴 FAIL | alias: %-15s | email: %-20s | Status: %s | Error: %s\n",
BUSINESS_USER_GET_BUSINESS, test.BusinessAlias, test.Email, statusCode, responseErr)
        }
    }

    responses[BUSINESS_USER_GET_BUSINESS].CSVRows =
append(responses[BUSINESS_USER_GET_BUSINESS].CSVRows, []string{
        test.Email,
        test.BusinessAlias,
        statusCode,
        fmt.Sprintf("%d", latency),
        responseErr,
        requestErr,
    })

  }

  // Result of business user get business
  calculateSummary(BUSINESS_USER_GET_BUSINESS, responses[BUSINESS_USER_GET_BUSINESS])

  for label, result := range responses {

    fileName, err := saveCSV(label, *code, result.CSVRows)
    if err != nil {
        fmt.Println("CSV write error:", err)
        continue
    }

    key := fmt.Sprintf("%s/%s/%s", *infra, label, fileName)
    err = uploadFileToS3(*bucket, key, fileName, *bucketRegion)
    if err != nil {
        fmt.Println("S3 upload error:", err)
    }
  }
}

func calculateSummary(label string, result *Result) {

  result.CSVRows = append(result.CSVRows, []string{})
  result.CSVRows = append(result.CSVRows, []string{"Final Summary", ""})
  result.CSVRows = append(result.CSVRows, []string{"passed", fmt.Sprintf("%d", result.PassedTests)})
  result.CSVRows = append(result.CSVRows, []string{"failed", fmt.Sprintf("%d", result.FailedTests)})

  // After all tests have run, log the final stats
  fmt.Printf("\n📝 Test run complete: %s\n", label)
  fmt.Printf("✅ Passed: %d\n", result.PassedTests)
  fmt.Printf("❌ Failed: %d\n", result.FailedTests)
  fmt.Println("")

  if result.PassedTests == 0 {
    return
  }

  var totalLatency int64
  for _, l := range result.SuccessLatencies {
    totalLatency += l
  }
  avg := totalLatency / int64(len(result.SuccessLatencies))
  getP := func(p float64) int64 {
    idx := int(float64(len(result.SuccessLatencies)) * p)
    if idx >= len(result.SuccessLatencies) {
        idx = len(result.SuccessLatencies) - 1
    }
    return result.SuccessLatencies[idx]
  }
  sort.Slice(result.SuccessLatencies, func(i, j int) bool {
    return result.SuccessLatencies[i] < result.SuccessLatencies[j]
  })

  result.CSVRows = append(result.CSVRows, []string{"avg_latency", fmt.Sprintf("%d", avg)})
  result.CSVRows = append(result.CSVRows, []string{"p90", fmt.Sprintf("%d", getP(0.90))})
  result.CSVRows = append(result.CSVRows, []string{"p95", fmt.Sprintf("%d", getP(0.95))})
```

```
    result.CSVRows = append(result.CSVRows, []string{"p99", fmt.Sprintf("%d", getP(0.99))})

    fmt.Printf("Final Summary: %s\n", label)
    fmt.Printf("⚡ Average Latency (successful responses): %dms\n", avg)
    fmt.Printf("📊 90th Percentile Latency: %dms\n", getP(0.90))
    fmt.Printf("📊 95th Percentile Latency: %dms\n", getP(0.95))
    fmt.Printf("📊 99th Percentile Latency: %dms\n", getP(0.99))
    fmt.Println("")
}

func saveCSV(label, code string, rows [][]string) (string, error) {

    outputFileName := fmt.Sprintf("output_%s_%s.csv", code, label)

    file, err := os.Create(outputFileName)
    if err != nil {
        return "", err
    }
    defer file.Close()

    writer := csv.NewWriter(file)
    defer writer.Flush()

    writer.WriteAll(rows)
    writer.Write([]string{}) // Add an empty row for spacing

    fmt.Printf("✅ Results written to %s\n", outputFileName)
    fmt.Println("")

    return outputFileName, nil
}

func uploadFileToS3(bucket, key, filename, region string) error {

    cfg, err := config.LoadDefaultConfig(context.TODO(), config.WithRegion(region))
    if err != nil {
        return err
    }
    s3Client := s3.NewFromConfig(cfg)

    file, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer file.Close()

    _, err = s3Client.PutObject(context.TODO(), &s3.PutObjectInput{
        Bucket: aws.String(bucket),
        Key:    aws.String(key),
        Body:   file,
    })

    if err != nil {
        return err
    }

    fmt.Printf("📤 File uploaded to S3 bucket '%s' at key '%s'\n", bucket, key)
    return nil
}
```

Appendix - J - Automated Testing Infra Terraform Code

```
// EC2
provider "aws" {
 region = var.region
 alias = "ec2"
}

data "aws_ami" "amazon_linux" {
 provider = aws.ec2
  most_recent = true
 owners      = ["amazon"]

 filter {
   name   = "name"
   values = ["amzn2-ami-hvm-*-x86_64-gp2"]
 }
}

resource "aws_instance" "load-test" {
 provider = aws.ec2
 ami           = data.aws_ami.amazon_linux.id
 instance_type        = "t3.micro"
 iam_instance_profile = var.profile
 security_groups = [aws_security_group.ec2_sg.name]

 user_data = <<-EOF
            #!/bin/bash
            # Install required packages
            yum update -y
            yum install -y amazon-cloudwatch-agent unzip golang

            # Configure Go environment
            mkdir -p /home/ec2-user/go
            mkdir -p /home/ec2-user/.cache/go-build
            echo 'export GOPATH=/home/ec2-user/go' >> /home/ec2-user/.bashrc
            echo 'export GOCACHE=/home/ec2-user/.cache/go-build' >> /home/ec2-user/.bashrc
            echo 'export PATH=$PATH:/home/ec2-user/go/bin' >> /home/ec2-user/.bashrc
            source /home/ec2-user/.bashrc

            # Create directory for app
            mkdir -p /home/ec2-user/app

            # Create the CloudWatch config file
            cat <<'EOC' > /home/ec2-user/app/cloudwatch-config.json
            {
              "logs": {
                "logs_collected": {
                  "files": {
                    "collect_list": [
                      {
                        "file_path": "/home/ec2-user/app/app.log",
                        "log_group_name": "GolangAppLogs-${var.region}",
                        "log_stream_name": "{instance_id}"
                      },
                      {
                        "file_path": "/home/ec2-user/app/src/output.log",
                        "log_group_name": "GolangAppOutput-${var.region}",
                        "log_stream_name": "{instance_id}"
                      }
                    ]
                  }
                }
              },
              "metrics": {
                "metrics_collected": {
                  "statsd": {},
                  "cpu": {
                    "resources": ["*"],
                    "measurement": [
                      {"name": "cpu_usage_idle", "rename": "CPU_USAGE_IDLE", "unit": "Percent"},
                      {"name": "cpu_usage_iowait", "unit": "Percent"}
                    ],
                    "totalcpu": false
```

```
            }
          }
        }
      }
      EOC

      # Download the application from S3
      aws s3 cp s3://${var.golang_source}/src.zip /home/ec2-user/app/src.zip

      # Unzip the application
      unzip /home/ec2-user/app/src.zip -d /home/ec2-user/app/

      # Create empty log files before starting the agent
      touch /home/ec2-user/app/app.log
      touch /home/ec2-user/app/src/output.log

      # Start CloudWatch Agent first
      /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl \
        -a fetch-config \
        -m ec2 \
        -s \
        -c file:/home/ec2-user/app/cloudwatch-config.json

      # Build and run the Go application
      cd /home/ec2-user/app

      ./myapp -infra ${var.target_infra} -bucket ${var.result_bucket} -bucket-region ${var.default_region} -domain
${var.cp_domain} -region ${var.region} -code ${var.code} > /home/ec2-user/app/app.log 2>&1 &
      EOF

  tags = {
    Name = "load-test-${var.region}"
  }
}
```

Appendix - K - Example Results of Automated Testing

// Australia - Admin Get Business
AWS region,ap-southeast-4
Country code,AUS
Target infra,1CPxnAP
Total Test Cases,500
Test Type,admin-get-business

email,alias,http_status_code,latency_ms,response_error,request_error
aus.Izabella.Schmeler1@example.com,test-aus-buss-1,200,727,,
aus.April.Mraz2@example.com,test-aus-buss-2,200,176,,
aus.Felicita.Klocko3@example.com,test-aus-buss-3,200,177,,
aus.Valerie.Cole4@example.com,test-aus-buss-4,200,176,,
aus.Fae.Bernier5@example.com,test-aus-buss-5,200,175,,
aus.Nathen.Bernier6@example.com,test-aus-buss-6,200,175,,
…

Final Summary,
passed,500
failed,0
avg_latency,186
p90,179
p95,189
p99,637

// Canada - Business Register
AWS region,ca-central-1
Country code,CA
Target infra,1CPxnAP
Total Test Cases,500
Test Type,business-register

email,alias,http_status_code,latency_ms,response_error,request_error
ca.Alek.Stroman1@example.com,test-ca-buss-1,201,1649,,
ca.Ransom.Wolf2@example.com,test-ca-buss-2,201,1201,,
ca.Holly.Dibbert3@example.com,test-ca-buss-3,201,1185,,
ca.Elisa.Blick4@example.com,test-ca-buss-4,201,1177,,
ca.Eddie.Monahan5@example.com,test-ca-buss-5,201,1180,,
ca.May.Powlowski6@example.com,test-ca-buss-6,201,1197,,
ca.Ollie.Bogan7@example.com,test-ca-buss-7,201,1359,,
ca.Andres.Shields8@example.com,test-ca-buss-8,201,1204,,
…
Final Summary,
passed,500
failed,0
avg_latency,1204
p90,1228
p95,1235
p99,1290