A Practical Approach to Convert Silo Web Application into Multi-Tenant Cloud-based SaaS Application

by

Omar Hassan Khan
19166019

A project submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
M.Sc. in Computer Science and Engineering

Department of Computer Science and Engineering
BRAC University
October 2025

Declaration

It is hereby declared that

1. The thesis submitted is my own original work while completing degree at Brac University.

2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.

3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.

4. We have acknowledged all main sources of help.

Student's Full Name and Signature:

Omar Hassan Khan
19166019

Approval

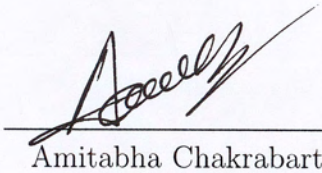
The project titled "A Practical Approach to Convert Silo Web Application into Multi-Tenant Cloud-based SaaS Application" submitted by

1. Omar Hassan Khan (19166019)

Of Summer, 2025 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of M.Engg. in Computer Science and Engineering on October 10, 2025.

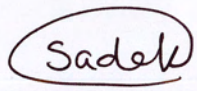
Examining Committee:

Supervisor:
(Member)



Amitabha Chakrabarty, PhD
Professor
Department of Computer Science and Engineering
Brac University

Program Coordinator:
(Member)



Md Sadek Ferdous, PhD
Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Sadia Hamid Kazi, PhD
Chairperson
Department of Computer Science and Engineering
Brac University

Abstract

Generally, SME software firms start developing an application with specific requirements and intended to use it as Application Service Provider (ASP). With no argument ASP fits best for agile development and meets development deadlines. In ASP model, we deploy the system and cater update, customization and troubleshooting by ownership. When we have the application deployed for more than a handful of users then it gets difficult to maintain the application for each owner. In Software as a Service (SaaS) model service is provided to each subscriber individually but underneath the application is maintained as one single instance. Converting ASP model applications to SaaS model application helps the provider to reduce development and operational costs, effortlessly update and do bug fixes, and attain troubleshooting effectively. The conversion of ASP to SaaS is not straightforward. The conversion starts with separation common modules and subscriber specific modules. Then there is existing database customization along with master table implementation for the subscription. Finally, we need to select proper cloud-based IaaS with different services to obtain maximum usage with minimal billing.

This project address the challenges of converting an ASP to a multi-tenant SaaS model with scalability, cost efficiency and ease of maintenance. Using the cloud as infrastructure gives the advantage of high security, reduces overhead cost and streamlines integration and deployment.

This paper aims to provide a structured implementation technique to convert an existing singleton web application to SaaS on multi-tenant shared infrastructure model. These improvements help software firms achieve better operational efficiency while offering a more sustainable SaaS solution. Also helpful for the researchers and engineers to easily implement multi-tenant architecture in action.

Keywords: Software-as-a-service, SaaS, Multi-tenancy, Technology Adoption, Cloud Deployment

Dedication

I would like to dedicate this work to Ms. Monjuri Mallick who has provided me the opportunity and pushed me to design a system beyond our comfort zone.

Thank you for being an amazing boss!

Acknowledgement

I express my gratitude to almighty Allah for successfully completing this project.

I would also like to extend my sincere thanks to my esteemed supervisor, Amitabha Chakrabarty, Professor of the Department of Computer Science and Engineering at Brac University, for his guidance and support throughout this project. His constant supervision and advice kept me on track and I am truly grateful for having him as my supervisor.

I would like to acknowledge the unwavering support of my parents, as well as my wife, who provided tremendous assistance during every stage of my master's work.

Lastly, I would like to express my appreciation to Brac University for providing me with the necessary resources and support to conduct this research.

Table of Contents

List of Figures

List of Tables

List of Code

Nomenclature

The next list describes several symbols & abbreviation that will be later used within the body of the document

$	United States Dollar
$Ver.$	Version
AI	Artificial Intelligence
ASP	Application Service Provider
AWS	Amazon Web Services
AZ	Availability Zones
CAPEX	Capital Expense
CDN	Content Delivery Network
CFIP	Concerns for Information Privacy
CI-CD	Continuous Integration and Continuous Delivery/Deployment
CRUD	Create Read Update Delete operation
DB	Database
DC-DR	Data Center–Disaster Recovery
DNS	Domain Name System
DR	Disaster Recovery
ECS	Elastics Compute Service
IaaS	Infrastructure-as-a-service
IP	Internet Protocol
IUPC	Internal User's Privacy Concerns

MFA Multi-Factor Authentication

MVC Model View Controller

OnM Operations and Maintenance

OPEX Operational Expense

OSS Object Storage Service

PaaS Platform-as-a-service

PIPC Private information privacy concerns

RBAC Role-Based Access Control

RDS Relational Database Service

RND Research and Development

SaaS Software-as-a-service

SDLC Software Development Life-cycle

SLA service-level agreements

SLS Simple Log Service

SME Small and medium-sized enterprises

SSD Solid-State Drive

TCO Total Cost of Ownership

UUID Universally Unique Identifier

VPN Virtual Private Network

VPS Virtual Private Server

Chapter 1

Introduction

According to Gartner Survey Report, the cloud service industry is predicted to reach from $679 billion in 2024 to around $1 trillion in 2027 worldwide[17]. Cloud computing provides massive computing power to its users dynamically. One of the major characteristics of cloud is the billing is much lower comparing to the infrastructure operational costs. Therefore, small to medium sized development firms can produce software which is faster, scalable and secure. In SaaS model the cost of cloud service can be easily adjusted as operational cost rather than the infrastructure cost.

Milind Govekar, Distinguished VP Analyst at Gartner said, "Organizations are actively investing in cloud technology due to its potential to foster innovation, create market disruptions, and enhance customer retention in order to gain a competitive edge. While many organizations have started to seize the technical advantages of cloud, only a few have unlocked its full potential in supporting business transformation. As a result, organizations are using the cloud to launch a new wave of disruption driven by AI, enabling them to unlock business value at scale."[17]

Currently most of the software development firms are using cloud services as a service facilitator to transform the non-cloud data-centre oriented infrastructure. As the demand for SaaS product keeps uprising the key decision point at present is how to serve as much client as possible keeping the CAPEX and OPEX in balance. While silo architecture services are very popular as it does not require extensive developments yet it comes with its own limitations such as scalability, low infrastructure utilisation and many more. Converting the same application into a cloud-based SaaS application will allow the provider to offer services to a broader market and access extensive computing power, resulting in numerous benefits.

Developing an existing application from scratch requires full efforts in the SDLC. Whereas there are standard techniques to convert silo application in to SaaS application. The method varies depending on the type of application, traffic size, architecture and its components. Rather developing a SaaS product form scratch; live applications can be converted into SaaS application very easily using different libraries and proposed architecture.

1.1 Motivation

The need of business solution has manifolds in past few years. Business owners are more attracted to SaaS applications due to the cost effectiveness, billing in as-per-usage method and reduced infrastructure costs. Popular web development frameworks like Laravel are mostly used for silo deployments and silo architecture lacks the auto deployment feature of a multi tenant application.

Transforming these siloed Laravel application into multi tenant SaaS presents a significant challenge but also a great worldwide opportunity. This projects aims to develop a straightforward technique specially tailored for web applications developed in Laravel framework. Laravel has reached almost one million live web application worldwide till March,2024[20]. Laravel's robust architecture, dynamic libraries and active developed communities has given it a strong foundation to build scalable and maintainable multi-tenant applications. By revamping existing application to multi-tenancy, application owners can unlock the potential for wider user reach, improved resource utilization within the application, and the potential for recurring revenue streams.

Moreover, this projects offers a unique focus on catering the vast young Laravel developers who often lacks the development expertise and experience of multi-tenant SaaS architecture. The step-by-step details in this paper can help individual developer to understand the challenges and best practices of multi-tenant SaaS purely based on Laravel. A successful outcome will not only empower individual developers but also contribute valuable insights and best practices to the broader Laravel community, encouraging the migration of existing applications to the lucrative SaaS model.

Finally, Applications owners need to move the infrastructure from on-premise to cloud service. This enables any software business to stretch the infra according to clients need with no CAPEX and invoice the cloud OPEX to clients. There is always a certain market for physical servers to satisfy client demand, fostering cloud service can help application owners to grab the opportunity of the SaaS market and providing valuable guidance for businesses seeking to leverage the cloud for wider reach and recurring revenue streams.

While working with a existing codebase, I have received new requirement to transform the existing siloed web application to a multi-tenant version. The primary requirement was to centralize the codebase for easy maintenance and central reporting portal for a detailed company-wide overview. In the RND phase, we could not find any suitable document for such conversion. Thus, I have decided to document the transformation journey as my academic paper with a view that it will help the Laravel developer community as well as the researchers to understand the process.

1.2 Problem Statement

The widespread adoption of cloud services has driven the rapid growth of business software implementation. Prior to cloud services, business used to subscribe VPS which was inexpensive in shot-term but piles a huge OPEX in long run. Also, businesses are increasingly attracted to SaaS due to its cost-effectiveness. However, the transformation of existing business software's particularly those built with popular frameworks like Laravel into multi-tenant SaaS offerings with Laravel holds substantial difficulties. This siloed architecture presents several hurdles when migrating to a multi-tenant SaaS model:

1. **Data Isolation:** Inter-tenant data leakage is a paramount in a multi-tenant environment. Existing Laravel built-in database driver lacks to effectively manage the tenant database at query level and potentially exposing sensitive information from one tenant to another[15].

2. **Resource Management:** While deploying SaaS applications into cloud architecture; resource management is an inevitable work to efficiently manage multiple tenant. Siloed Laravel applications may not have the necessary architecture to handle tenant-specific processing mechanism and resource allocation, potentially leading to performance bottlenecks or resource exhaustion. While high user tenant consuming majority of the resources; light user might face delayed services.

3. **Scalability:** SaaS applications are expected to scale as the subscriber grows. Laravel applications are designed for single-tenancy and it may struggle to handle the increased load and complexity of a multi-tenant environment.

4. **Deployment and Maintenance:** Migrating a web-application from physical server to cloud server such as ECS or Kubernetes requires specialised configuration and processes. Direct server deployments may not be optimised for such remote environments. Traditional deployment may cause service interruption, data discrepancy or maintenance overhead.

5. **Limited Community Resources:** Although Laravel has a vast knowledge base and supportive developer community; there is huge knowledge gap in complex system architecture specially in multi-tenant architecture. While working on this project, we have only found a couple of libraries with proper documentation for converting an existing Laravel application. SaaS applications require subscription panel with payment option. Such bootstrap package is found in "Tenancy for Laravel" package only[22].

6. **Security:** Security is another critical issue for multi-tenant Laravel applications. Applications should have a robust security model that ensures the confidentiality, integrity, and availability of tenant data. Security concerns are not limited to data isolation but also include access control, user files isolation, regular system checks and tenant wise data encryption.

This project aims to address these issues by developing a practical approach that leverages the strengths of Laravel while providing solutions for the specific challenges of multi-tenancy.

1.3 Contribution

This project aims to provide a practical approach for transforming siloed Laravel applications into multi-tenant SaaS solutions. As mentioned in the last section, there is a gap in readily available resources for such transformation process. Developers often rely on scattered information and experimentation which does not follow the standard techniques. Thus, hindering the adoption of the SaaS model within the Laravel ecosystem.

By outlining each steps of the conversion process, this project empowers developers to confidently migrate their Laravel applications to the SaaS model. The project's findings and best practices will be documented and shared with the Laravel community, fostering wider adoption of SaaS solutions for Laravel applications.

1.4 Organization of the report

The chapters of this reports include the following.

Chapter 2: Literature Review, begins by examining the evolution from Application Service Provider (ASP) models to Software-as-a-Service (SaaS), followed by the role of cloud services in enabling SaaS. Additionally, it discusses multi-tenant architectures and the critical privacy concerns in SaaS applications.

Chapter 3: System Design, outlines the design and architecture of the multi-tenant SaaS system. It covers the development stack and the overall system architecture.

Chapter 4: Solution Overview, discusses the re-engineering method, the steps for SaaS application conversion, the cloud services required to support multi-tenancy, and a detailed migration plan to ensure a smooth transition.

Chapter 5: Implementation, covers the application transformation process, the setup of cloud infrastructure, and the final deployment and go-live procedures, ensuring a seamless transition to the new multi-tenant architecture.

Chapter 6: Benefits and Impact of Multi-Tenant SaaS Conversion, highlights the advantages of converting to a multi-tenant SaaS model, including cost efficiency, scalability, improved maintenance, enhanced security, and broader market reach, while also discussing its positive business and environmental impacts.

Finally, **Chapter 7**, draws the threads together, by envisioning the future developments and offering a comprehensive conclusion to the project.

Chapter 2

Literature Review

2.1 The concept of ASP and SaaS

The concept of ASP (Application Service Provider) refers to the process where provider installs a business application on the host server and charges an amount to the users. The host server can be on premise or remote. In ASP model the service and hardware remain single tenant and does not share with other businesses. Typically, ASP charges its users license fee based on various service models depending on the relationship with customers.

The concept of SaaS (Software as a Service) refers to cloud-based service in which a service provider develops, hosts, manages and delivers software via the Internet [9]. Most of the SaaS clients purchase subscription as per their required modules and number of users. Comparing with ASP it has advantages like reduced infrastructure cost, improved performance, easy to maintain clients, global accessibility etc.

Although SaaS is an improved form of ASP, it differs from ASP in many contexts. Number of software users differs in SaaS and ASP. By using SaaS method, the software provider can deliver many subscribers from a single service. While hosting multiple users in a single cloud instance the operational cost is reduced for all the users which adds a pricing advantage for both provider and client [8].

2.2 Use of cloud services for SaaS

The base concept of cloud computing is availability to computing resources on demand via internet network so users can utilize the resources anytime. Traditionally, the ASP used to purchase required hardware and software to support the system running centrally. Only limited number of users were given access over the internet to reduce over utilization of CPU processing, bandwidth and security risk. Then comes the era of shared hosting where systems are hosted with limited resources and replicated into multiple host as per requirement. Now, under cloud computing environment SaaS providers enjoys the benefit of virtualization, on-demand computing, scalability and security. Developers just need internet enables terminal to access the cloud environment. Combining it with SaaS model providers can easily define its required resources prior to deployment and subscribe with pay per use scheme.

There main four layer of cloud computing are defined in hierarchical orders[5].SaaS delivers the code layers, following with PaaS as managed services and IaaS for shared

physical resources. The bottom layer is hardware and infrastructure which includes servers, storage, network, generator and other resources.

SaaS fits best with cloud computing as it allows provider to avoid the complexity and expense of infrastructure and brains to create a highly scalable server. The subscribers enjoy to option to choose required modules. This helps customers to concise their subscription fee with required only features. Such example can be Atlassian products. You can purchase the complete Jira suite or choose specific products based on your budget.

2.3 Multi-Tenant models

Generally, a SME software built on monolith model and hosted in silo model. If the server is used for one single system then it is referred as independent infrastructure. On the other hand, if there are multiple system hosted within a single server then it is referred as shared infrastructure. To have the maximum utilization of CPU, service providers use the shared infrastructure model.

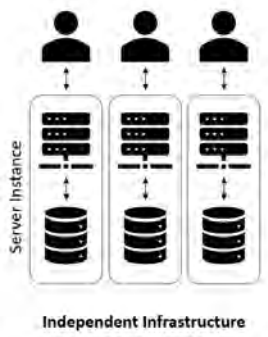


Figure 2.1: Silo Model

There are different multi-tenancy models. Two most popular models are “Shared code base with shared database” and “Shared code base with distributed database”.

2.3.1 Shared code base with shared database

The shared code base with shared database model is considered as fully mutitenant deployment. In this model components are shared and only one set of infrastructure is deployed and maintained. This is model is attractive because of cost effectiveness and easy maintainability. Even if there is a requirement to upgrade the server, migration can be done very easily. Additionally, if a user want to migrate or merge from one tenant to another; process can be handled by changing the identifier in the database.

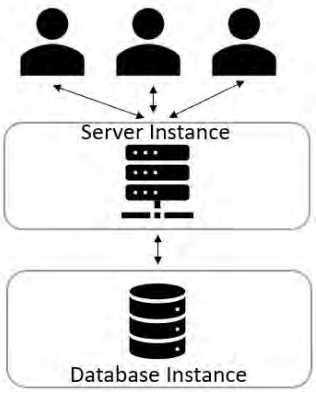


Figure 2.2: Shared code base with shared database

The major disadvantage of this model is data leak risks. A simple logical mistake in the database query can leak data between tenant. The next challenge is to allocate CPU usage from system end. If one tenant is performing heavy queries on regular basis it will also affect other tenants as well. Scaling shared system

can be factor to consider as well. Infrastructure resource limit can be reached.

2.3.2 Shared code base with distributed database

This model is also known as the Horizontally Partitioned model. In this model the code base is shared, which means that only one web server is deployed. The database can be deployed on different servers or it can also be hosted on same database servers. Every tenant will have his database. Based on tenant identification, the web server will choose the underlying database

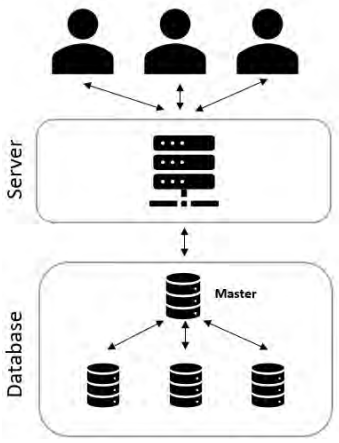


Figure 2.3: Shared code base with distributed database

This eradicates the noisy-neighbor issue. If a tenant sends a large number of queries, then the performance of that specific neighbor is affected rather than slowing the data queries of other tenants. The data leak issue from previous model is also resolved in this system as it use separate database.

Scalability is one of the prime issues in this model. Traffic is routed to web server and then to the database servers. NAT gateway needs to be configured with precision to serve every tenant from a single source code. Managing tenant specific modules is another challenge for the developers using this mode. After developing a monolithic system and then shifting it to this multi-tenant model consumes more time in redefining the tenant-specific logic and bug fixing.

2.4 Privacy in SaaS Application

Along with the opportunities of using hosting all the clients in single or multiple cloud-based infrastructure, a great challenge has been identified as in terms of client's privacy and data security which hinders the SaaS adoption. This breach can be occurred by unauthorized access to services, server access breach, lack of experienced cloud administrator and code level information breach. Gashami et al (2015)[11] has mentioned key heuristics for SaaS adoptions in the industry which includes the following points.

1. Trust towards the software service provider.

2. Private information privacy concerns (PIPC) affecting trust.

3. Overall perceived benefits positively affecting intention to use and trust.

4. Cost effectiveness, software performance and user experience positively affecting overall perceived benefits.

Their research used human behavioral factors to understand the privacy concerns in business system implementation. Users faces a great anxiety attack while understanding the process of releasing the data in cloud infrastructure. Smith et al[1] developed "Concerns for Information Privacy" (CFIP) to understand the individual privacy concerns while moving into online based business solutions. Another great privacy concern is the "Internal User's Privacy Concerns" (IUPC). Modern applications have dynamic roles and features according to user needs. While access can be secured from back-end permission verification but loopholes can still remain when the developer miss roles-based permission. This scenario is also valid for the cross-tenant information breach. The privacy features should be strengthened from all aspects in software development life-cycle.

2.5 Research Gap

As the project involved refactoring a legacy system rather than developing an entirely new one, it was challenging to locate prior academic studies or comprehensive documentation directly focused on system refactoring and SaaS transformation. Most of the literature highlights SaaS architecture, economics and operational advantages including better better scalability, reduced infrastructure cost, centralised maintenance and migration plans[8][9]. Cold services further enhance these benefits by offering virtualization, pay-as-you-go pricing and serve broader markets with efficient pricing [5][17]. In reality, the actual redevelopment works of legacy monolithic application to cloud based multi tenant architecture poses complex challenges. Zimmermann [10] and Kerievsky [3] propose architectural refactoring methods and design patterns, yet practical implementations remain underrepresented in Laravel-based ecosystems.

Although Laravel is the most popular web application development framework, the central application and tenant application model is not addressed in the literatures where core authentication, accesses and subscriptions are shared with two different systems. Most of the academic works are focused on the high level implementation without suggesting the platform specific strategies [7]. Additionally, the tenant identification mechanism for cloud based deployment is missing the scholarly journals. Although the import aspects like managing data isolation, routing logics and migration plans can be found [22] still the actual development plan or methods are nowhere found in academic papers. This paper contributes these technical voids to demonstrate a structured approach for tenant separation, authentication and cloud based service delivery.

Chapter 3

System Design

3.1 Types and modes of multi-tenancy architecture

From database perspective we have two types of multi-tenancy; single database tenancy or multi-database tenancy. For this project we will use muti-database tenancy where each new tenant will have their own database while authentication and service details remain within the master table.

From code perspective there are two types of modes; automatic and manual mode. In the automatic mode after the tenant is identified the default database, cache files, file system, configurations etc. are switched to current tenant context. This will help us to attain the tenant separation and privacy. As tenant selection happens one layer below the application using automatic mode will help the developer to write the code as monolithic architecture without any idea of multi-tenant architecture. Thus, we will be using the automatic mode for this project.

3.2 Tenant Identification

In SaaS architecture, there are many types of tenant resolver. Most common tenant identification methods are:

- Domain identification *(mydomain.com)*
- Subdomain identification *(company1.mydomain.com)*
- Domain OR subdomain identification *(both of the above)*
- Path identification *(mydomain.com/company1/dashboard)*
- Request data identification *(mydomain.com/users?tenant_id=company1)*

For this project we will use domain identification method.

3.3 Two Applications

Following two terms will be used throughout this document.

- Central Application: This application will host our master database for sort of admin logic, tenant creation and identification and central dashboard for services.

- Tenant Application or SaaS Application: This application is the SaaS service or web application you will provide to the users. Depending on the number of the clients this application will consume larger part of the infrastructure.

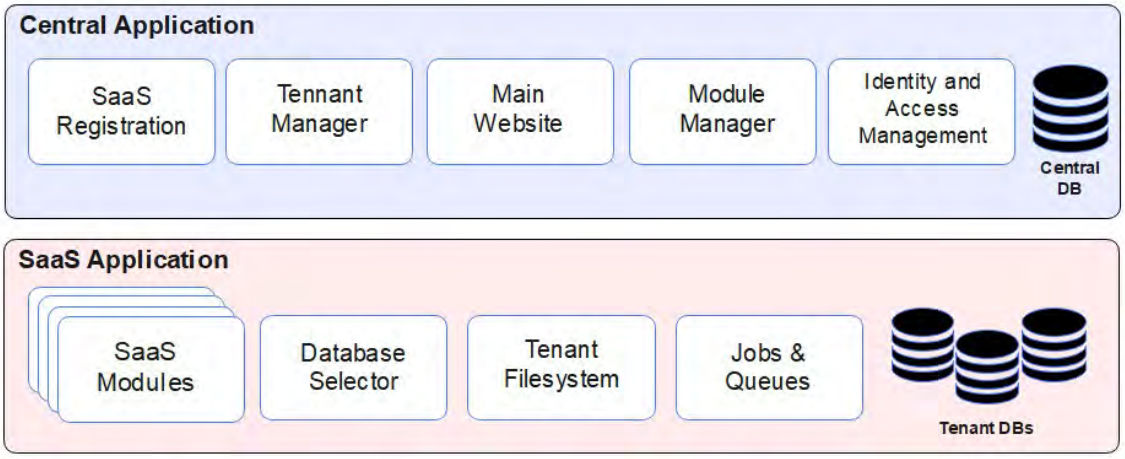


Figure 3.1: Two Application models

3.4 Development Stack

The development stack along with the versions are as following.

- Development Language: PHP *(Ver.8)*

- Database: MySQL *(Ver. 7.2.*)*

- Framework: Laravel *(Ver. 9)*

- Local-host Server: Laragon *(Ver. 6.0)*

- Cloud Service: Elastic Computing Service, Managed Database Service, Object Storage

- Web-server: Ngnix *(Ver. 1.26)*

Laravel Libraries or dependencies required

- Composer *(Ver. 2.7.6)*

- Tenancy for Laravel *(Ver. 3)*

3.5 System Architecture

Our goal is to achieve a standard re-engineer method for the developers transforming singleton legacy web application to mutitenant cloud based web application. The greatest benefit of cloud based benefit is scalability based on the number of tenants.

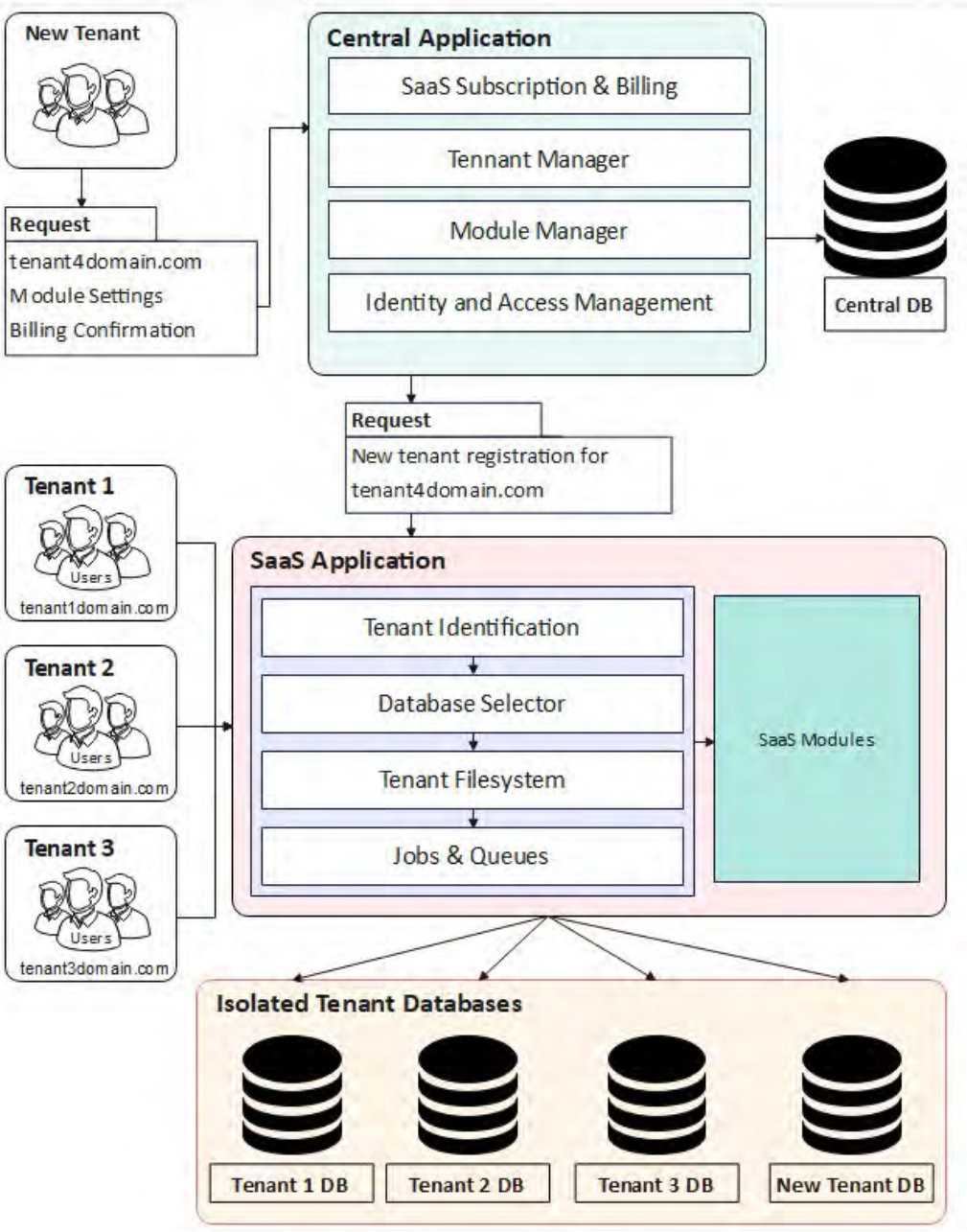


Figure 3.2: High level system architecture and user flow

3.6 Development prerequisite

3.6.1 Domains

We will need one central domain to host your SaaS product. Based on the tenant identification models mentioned in 3.2; number of domains might differ as service. As we will be using domain-based identification for this project; we will need one domain *(mysaasapplication.test)* for central application and two domains *(client1mysaas.test, client2mysaas.test)* for our tenants. The sample domains in this section are based on local development environment.

3.6.2 Databases

We need to setup one database for our central application. This database will be used for our tenant registration and tenant creation.

Tenant database will be created based with migrations and modified Laravel MySQL driver will be used select the database for the tenants.

3.6.3 Tenant Identification

Along with domain-based identification we will need an UUID for internal tenant identification. After the tenant creation is fired an auto incremented value will be tagged with the tenant. For better understanding the tenant id will be prefixed by "tenant_" in other usage.

3.6.4 Tenants File-system and Cache

Using two tenants will be very simple as in just create two folders in the public folder location. Creating tenant aware file-system is challenging and it requires unique identifier for this purpose. The file-system bootstrapper makes your app's Storage facade and the storage_path() and asset() helpers tenant-aware by modifying the paths they return.

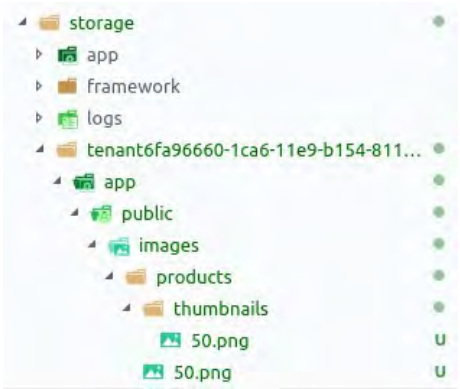


Figure 3.3: Laravel file-system

Storage path helper

The bootstrapper suffixes the path returned by storage_path() to make the helper tenant-aware.

- The suffix is built by appending the tenant key to your suffix_base. The suffix_base is tenant by default, but feel free to change it in the tenancy.filesystem

config. For example, the suffix will be tenant42 if the tenant's key is 42 and the suffix_base is tenant.

- After the suffixing, storage_path() helper returns "/$path_to_your_application/storage/tenant42

Since storage_path() will be suffixed, your folder structure will look like Figure 3.3. Logs will be saved in storage/logs regardless of any changes to storage_path() and regardless of the tenant.

Cache tenancy bootstrapper
The cache tenancy bootstrapper replaces the Laravel's CacheManager instance with a custom CacheManager that adds tags with the current tenant's ids to each cache call. This scopes cache calls and lets you selectively clear tenants' caches: php artisan cache:clear –tag=tenant_123

3.6.5 Jobs and Queues

In the multi-database model jobs are already tenant aware and executed from the tenant DB. Queued jobs dispatched from the tenant context will be automatically tenant-aware. For the simplicity of this project, we will implement DB based job queues. Otherwise, it is preferred to implement cache based in-memory data service like Redis, Cosmo DB, Couchbase etc. for high frequency jobs dispatcher. It will also help to reduce the traffic and operation load from core database.

To fire up new tenant or run jobs on existing tenants we need central queues. Jobs dispatched from the central context will remain central. However, it's recommended not to mix queue connections for central and tenant jobs due to potential leftover global state, e.g. central jobs thinking they're in the previous tenant's context.

Chapter 4

Solution Overview

4.1 Re-engineering method and process

The purpose of this section is to guide developers transforming a silo model independent web application into a cloud based multi-tenant web application.

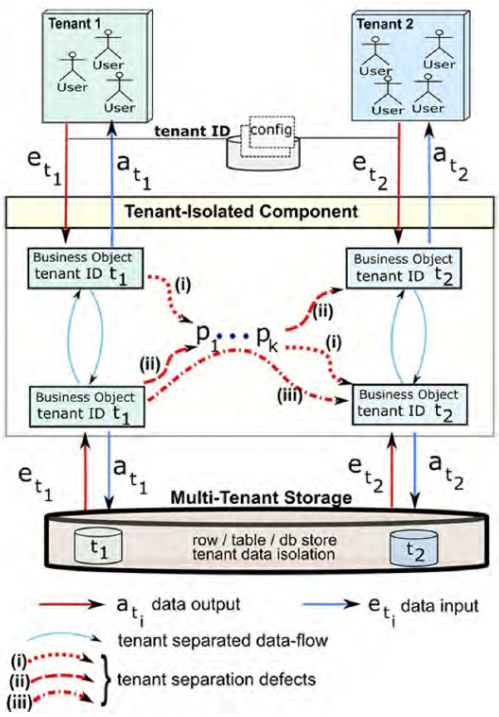


Figure 4.1: Illustration of data flow[12]

This project focuses on the widely used MVC pattern in legacy web application. As Laravel already used MVC pattern it can easily be refactored using standard refactoring patterns[3], [7]. According to Zimmermann[10] the architectural refactoring can be outlined in three major steps.

- Step 1: Identify and define
  Identify data access, user interfaces, services and data processing. Define the application functional layer based on the identification.

- Step 2: Decompose
  Break down each layer into independent components so that it can deployed as tenant independent modules.

- Step 3: Re-engineer to support multitenancy.
  In this step the MVC components are reassembled to support the multitenancy. This step is further described in details over the next sections.

4.2 SaaS Application Conversion

A typical MVC pattern structured into three different layers; presentation layer, data access layer and processing layer. View works in the presentation layer where as controller acts as the middleware between model and view. Model works in the data access layer and tenant database will be selected in the processing layer of the application for each request. This project focuses on the converting the existing MVC configuration to accommodate tenant specific shifting in each request. Details of the conversion techniques are mention over the next subsections.

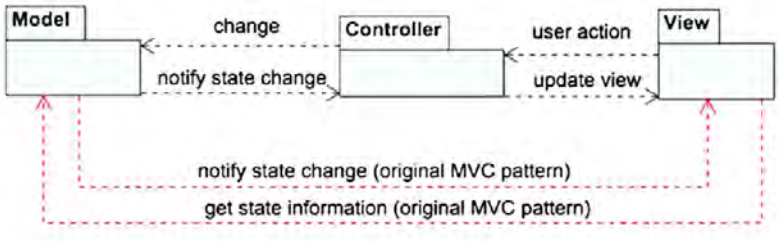


Figure 4.2: Laravel MVC Pattern

4.2.1 Model Conversion

Primary requirement of multi tenant model is to store data object into right database and retrieve data assuring tenant data security.

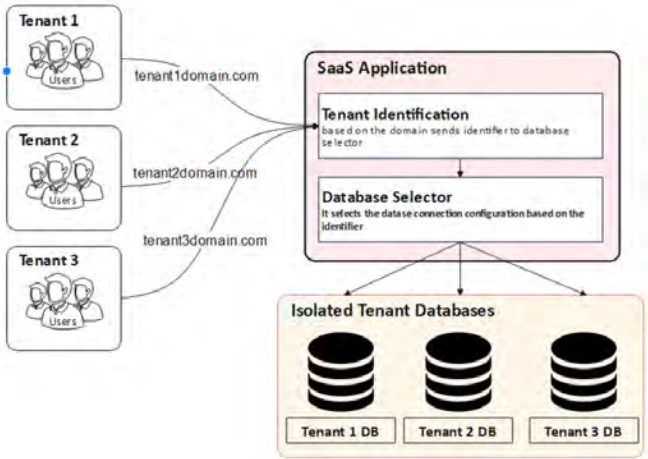


Figure 4.3: Illustration of database selector

This operation will be performed in the data access layer. Laravel database driver fetches connection configuration before the accessing database. A database selector based on the tenant identifier will be used for data storing and retrieval.

Another common concern is data separation in between tenants. Using the tenant domain as identifier for database selector; tenant data separation is confirmed. This

connection to remains as one-to-one relation between model and database. There is minor chance in loading cross tenant data or accessing data due to configuration faultiness or cache issue. This sort of error can be identified in runtime and resolved immediately based on identification. While the fault identification remains as a challenge; implementing cloud-based log service can help us tracing such issues.

In the figure 4.1 when a tenant initialise an request it hit the tenant identification layer first then then it goes to application layer. Once the the tenant is identified using its domain, the database sector selects the configuration based on the tenant identification or the tenant id. Based on the identification and the configuration tenant database is selected, connected and used for operation.

4.2.2 Controller Conversion

Ideally, Laravel handles the MVC functionalities very from it's own kernal. Each requests are identified and processed individually. While working with the multi tenant MVC we need to keep in mind that all the tenants will not have access to all the features. In this scenario there are two possible option.

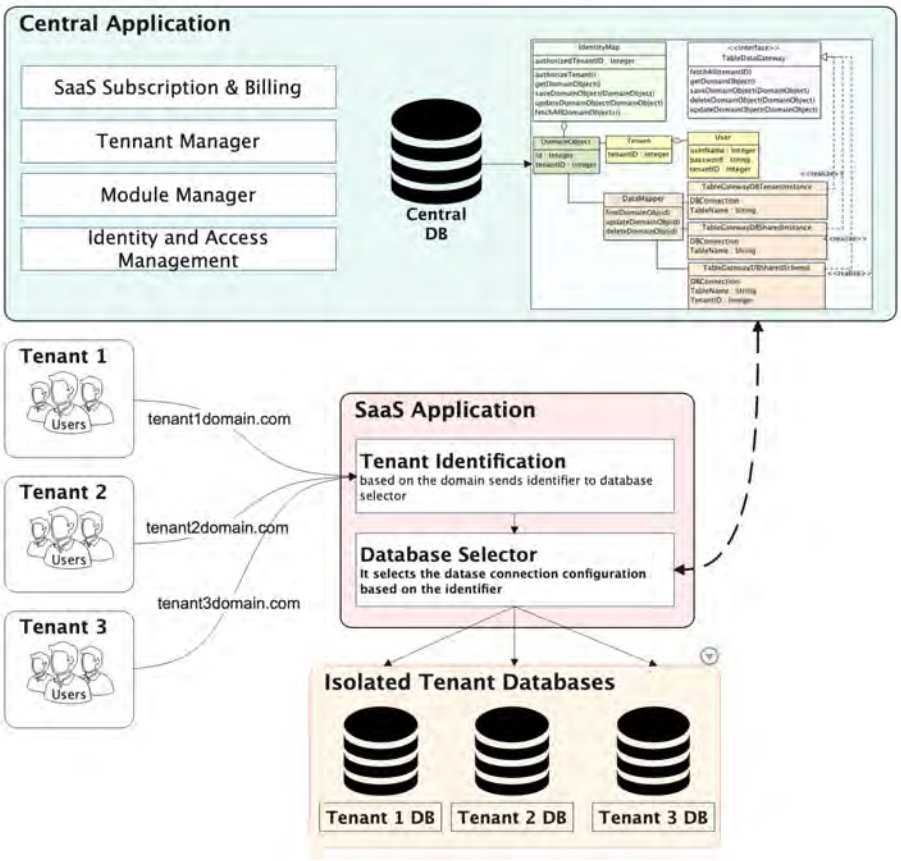


Figure 4.4: Multitenant MVC model based on Identity Map, Domain Model, Identity Field, Data Mapper, and Table Data Gateway

1. **Application configuration in central application**
   In this option module subscription details are stored in the central application and inserted into SaaS application based on CRUD operation.

The greatest advantage of implementing this model is central operation. Central admin can easily access any application permission from central application and perform any permission related operation very easily.

Downside of this model is in-development the tenant identifier and database selector need to be flawless. In any scenario if there is a bug in the logic then it can hamper any number of users in SaaS. Along with this issue we have cache and configuration issues in this model such as when the configurations are modified from central application the existing configurations need to cleared. Thus the new configurations are loaded in to system.

2. **Application configuration in SaaS application**
   This option is already implemented for most of developed application in Laravel application as Role and permission. As this project focuses on the shared codebase and isolated database model if role and permission is already implemented in an application it can easily converted into multi tenant architecture.

   Benefits in this model is easy conversion with almost no change in code level. For any tenant side changes central admin needs to visit each tenant individually which is the ultimate decliner for a SaaS with more than a dozen of subscriber. In some instance if the role and permission needs to be re-evaluated for new access rules which is another downside of this option.

As shown in figure 4.4 central application and SaaS application database driver is always connected and any update from central application is pushed via SaaS database selector.

In this project, application configuration will be stored in central application and mirrored into tenant database directly. This will ease the tenant based controller operation with Laravel Guard.

The routes and dynamic menu controller along with role and permission we will also implement the an module access in the Guard to ensure SaaS client and end-users has the sufficient permissions for modules before accessing it.

4.2.3 View Conversion

Most of the application allow users to configure layouts such as menu, titles, custom forms, named routes and parameters routes. For view conversion Furda et al [12] has suggested a Two-Step-View which will allow developers to decouple displayed information from views in the presentation layer.

The first step is common for all the application which is rendered into second step where configurable tenant based logics are implemented. In cases tenant may request exclusive page design or view features. In this case we create a third step to logically setup tenant's specific request.

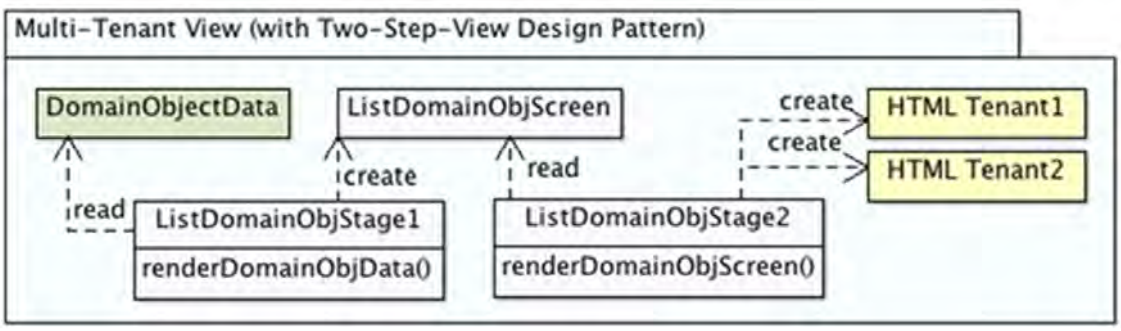


Figure 4.5: Two step view pattern for multitenant MVC

To start the view conversion start finding all the common pages such as login, home, reports etc. Transfer all the common pages under a common section in route file. Then start the second step of the conversion as in tenant based logic implementation. These pages can be combined with common pages such as home page contains menu or report page contains tenant based access to specific reports. To complete step two please add security gates to each pages. We will use Laravel auth guard and sessions to secure our application.

4.3 Cloud services requirement to support multi-tenancy

The majority of previous systems are deployed in self hosted servers or VPS. The MVC structure typically divided in the three different layers as in data access layer for model, presentation layer for views and processing layers for controllers.

For this project we will use the cloud services mentioned over the next sections.

4.3.1 ECS (Elastics Compute Service)

Elastics Compute Service or Elastic Container Service is a fully managed orchestration service to manage, deploy and host your application. Central application and SaaS application can be hosted in same or separate ECS instance. Although for high traffic SaaS products it is recommended to use separate instance.

4.3.2 RDS (Relational Database Service)

Relational Database Service is a managed database service to enhance the application performance. Using RDS effectively reduces the SQL query time, confirms high availability and assures on-time backup. RDS will be used to host central application database and enable the auto creation option of tenant databases.

4.3.3 Object Storage

High performance memory or SSDs with ECS instance are very costly in terms of OPEX. Solely for this purpose it is recommended to use cloud-based object storage

service. The benefits include memory scalability, data protection, high availability and object durability options. Comparing with ECS memory options; Object Storage service will guarantee highest performance with lowest price. Tenants uploaded objects or any sort of objects; which are not part of configuration or system files are stored directly into object storage.

4.3.4 Log Service

Cloud based log service enables its users to centralize logs from all the components into single platform. Once all the logs are centralized; system admin can easily monitor, query, detect and debug the application in runtime. For multitenant system it is very useful for the DevOps team.

4.3.5 Domain Service

Generally, the subdomain comes with the domain panel. Engineering an automatic subdomain creation along with reliable traffic configuration requires cloud-based Domain service. As we are using subdomain for tenant identification Domain Service will help to easily register new tenants. Easily register new tenant subdomain complemented with routing policies, reduced DNS resolving latency, enhancing application availability and complying with security. Another benefit of using Domain service is restricting tenant with VPN or IP address based validation.

4.4 Migration Plan

This project focuses on the migration of a developed Laravel application inspired from the E. Wolff [14] migration approaches. Although for large SaaS application that are already in-use should follow the Strangler approach [2]. The overall plan is divided into three main phases as in analysis phase, transformation phase and implementation phase.

1. **Analysis Phase**
   In this phase we determine the internal process flow of an application and how we can split the tenant specific logic. The goal is to group the module code requirements in order simplify the migration process.

2. **Transformation Phase**
   In the second step we adapt the application current design to match the new cloud-based multitenant architecture. If the existing setup cannot be changed easily then the legacy codebase is converted to support the new infrastructure. This includes separate storage for each tenant, and a gateway that acts as an intermediary between two applications and the internal communication system between the modules. Some of the work to move existing parts of the application into multitenant can be automated, for instance using the method developed by Christoforou et al [13] or Stancl [22].

3. **Implementation Phase**

The last step involves building the central application and rebuilding the SaaS application we planned earlier and integrating them with the new cloud-based infrastructure we suggested in the section 4.3.
Furda et al [12] has used a realistic example to show how a monolithic application can be transformed into a multitenant application. Their approach consists of two prioritization. The first one prioritizes minimal changes to the code and conflicts. The other approach (explained in section 4.1), uses a MVC pattern-based architecture. This second method is better suited to keeping each tenant's data separate.

Chapter 5

Implementation

This section will cover the transformation techniques along with the cloud migration and backup procedures at the end. For the implementation we will mostly working with a Laravel package called "TenancyForLaravel"[22]. In this transformation process, we have considered an already developed inventory app which is siloed for multiple businesses. Consider the fact all the deployed version are carbon-copies of single codebase and databases are separately hosted. In this process we first convert the developed the application; then deploy the central app for tenant registration and migrating to cloud infrastructure. Through out the implementation steps, we consider that the reader is familiar with the development stack mentioned in section 3.4 and have prior knowledge in PHP-Laravel development. Installation and usage details of some of the prerequisite can be found the in Appendix-ABCD.

5.1 Transforming the Developed Application

Composer is a dependency management tool for PHP which will allow us to easily manage the Laravel packages such as install, update or remove. Composer runs on a per-project basis. For this reason, we need to run the Composer commands from the application root.

Artisan is the CLI included with Laravel and at the root of the application to assist with helpful commands for building and running the application. Same as Composer, Artisan needs to be run from the document root.

5.1.1 Installing TenancyForLaravel Package

Run the following command from application root folder.

```
$ composer require stancl/tenancy
$ php artisan tenancy:install
```

Code Block 5.1: Installing the TenancyForLaravel Package from CLI

It will create the following files in Laravel Applicaiton

- migrations
- A configuration file (config/tenancy.php),

- A routes file (routes/tenant.php),
- A service provider file (app/Providers/TenancyServiceProvider.php).

To bootstrap the the tenancy feature we need to add the the service provider to bootstrap/providers.php file:

```
1 return [
2     App\Providers\AppServiceProvider::class,
3     App\Providers\TenancyServiceProvider::class, // add this
      line
4 ];
```

Code Block 5.2: Adding Tenancy ServiceProvider

As we want to keep the central database completely separate, a new database connection parameters needs to be added in the .env file.

```
// Other Codes.....

CENTRAL_APP_DB_CONNECTION=mysql
CENTRAL_APP_DB_HOST=[DB_HOST_IP]
CENTRAL_APP_DB_PORT=[DB_HOST_PORT]
CENTRAL_APP_DB_DATABASE=[DB_NAME]
CENTRAL_APP_DB_USERNAME=[DB_USERNAME]
CENTRAL_APP_DB_PASSWORD=[DB_PASSWORD]

// Other Codes.....
```

Code Block 5.3: Addding new central database in .env file

The same DB connection informations needs to be added in the in the "config/-database.php" file as "cental". Make sure it's the same name as the tenancy.central_connection config.

```
'connections'=>[

// Other Codes.....

    'cental'=>[
        'driver'=>'mysql',
        'url'=>env('DATABASE_URL'),
        'host'=>env('CENTRAL_APP_DB_HOST', '127.0.0.1'),
        'port'=>env('CENTRAL_APP_DB_PORT', '3306'),
        'database'=>env('CENTRAL_APP_DB_DATABASE', 'forge'),
        'username'=>env('CENTRAL_APP_DB_USERNAME', 'forge'),
        'password'=>env('CENTRAL_APP_DB_PASSWORD', ''),
        'unix_socket'=>env('CENTRAL_APP_DB_SOCKET', ''),
        'charset'=>'utf8mb4',
        'collation'=>'utf8mb4_unicode_ci',
        'prefix'=>'',
        'prefix_indexes'=>true,
```

```
            'strict'=>true,
            'engine'=>null,
            'options'=>extension_loaded('pdo_mysql') ?
        array_filter([
                PDO::MYSQL_ATTR_SSL_CA=>env('MYSQL_ATTR_SSL_CA'),
            ]) : [],
        ],

    // Other Codes.....
```

Code Block 5.4: Addding new central database in database config file

Now we can run the migration command to generate the base tables into our database.

```
$ php artisan migrate
```

Code Block 5.5: Artisan Migrate Command

5.1.2 Creating a tenant model

The package comes with a default tenant model which has many features. We need to create a custom tenant model to use domain identification and database isolation.

You can run the following command for Model creation or create a file in Create the file app/Models/Tenant.php.

```
$ php artisan make:model Tenant
```

Code Block 5.6: Artisan Create Model Command

Insert the following code block to create a Tenant model.

```
1  <?php
2
3  namespace App\Models;
4
5  use Stancl\Tenancy\Database\Models\Tenant as BaseTenant;
6  use Stancl\Tenancy\Contracts\TenantWithDatabase;
7  use Stancl\Tenancy\Database\Concerns\HasDatabase;
8  use Stancl\Tenancy\Database\Concerns\HasDomains;
9
10 class Tenant extends BaseTenant implements TenantWithDatabase
11 {
12     use HasDatabase, HasDomains;
13 }
```

Code Block 5.7: Tenant Model

Modify the custom tenant model information in the tenancy config file (config/tenancy.php).

```
1 'tenant_model' => \App\Models\Tenant::class,
```

Code Block 5.8: Custom model in tenancy config

5.1.3 Events

The defaults will work out of the box here, but a short explanation will be useful. The TenancyServiceProvider file in your app/Providers directory maps tenancy events to listeners. By default, when a tenant is created, it runs a JobPipeline (a smart thing that's part of this package) which makes sure that the CreateDatabase, MigrateDatabase and optionally other jobs (e.g. SeedDatabase) are ran sequentially.

In other words, it creates and migrates the tenant's database after it is created — and it does this in the correct order, because normal event-listener mapping would execute the listeners in some stupid order that would result in things like the database being migrated before it's created, or seeded before it's migrated.

5.1.4 Routes Modification

Central Routes and Central Domain

We need a small change in our existing route file to make sure that central routes are registered on the central domains only. The changes is required for routes/web.php, api.php or any other central route files.

```
foreach (config('tenancy.central_domains') as $domain) {
    Route::domain($domain)->group(function () {
        // Application actual routes
    });
}
```

Now we need to actually specify the central domains. A central domain is a domain that serves your "central app" content. Add the following code block in config/tenancy.php file.

```
'central_domains' => [
    'mysaasapplication.test',
],
```

For development purpose it is recommended to add the [dot]test domains in the host file. Check the appendix section for host file modification.

Tenant Routes

Your routes/tenant.php will look like this by default:

```
Route::middleware([
    'web',
    InitializeTenancyByDomain::class,
    PreventAccessFromCentralDomains::class,
])->group(function () {
    Route::get('/', function () {
        dd(\App\Models\User::all());
        return 'This is your multi-tenant application. The id
    of the current tenant is ' . tenant('id');
    });
});
```

This modification in tenant base URL will show the all the users for this specific tenant.

The ***InitializeTenancyBySubdomain*** middleware, to check whether the current hostname is a subdomain on one of your central domains.

These routes will only be accessible on tenant (non-central) domains — the ***PreventAccessFromCentralDomains*** middleware enforces that.

5.1.5 Tenant Database and env file

.env file

Add the following lines for defining the session in env file

```
\\ ...
SESSION_DRIVER=database
SESSION_LIFETIME=120
SESSION_ENCRYPT=false
SESSION_PATH=/
SESSION_DOMAIN=mysaasapplication.test
\\ ...
```

Tenant Migration Files

The tenant migration files needs to be separated from the central migration folder. Currently, we have all the migrations in one folder of ‘database/migrations‘. But the package created the ‘tenant‘ sub folder, which is empty, and you need to move all the migrations of the Tenantable data to that folder. So, we have projects and task migrations that should be in the ‘tenant‘ folder. All the existing migration files required to launch a fresh laravel system should be added under database/migrations/tenant directory.

You can execute the migration files by using the following command.

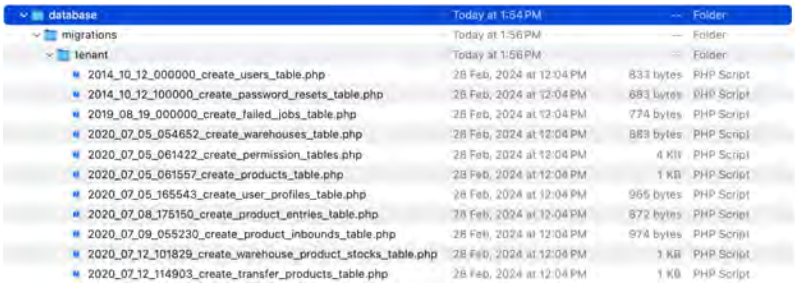


Figure 5.1: Tenant Migration Files

```
php artisan tenants:migrate.
```

Tenant Seeder Files

For seeding the database we need to move all the existing seed files under the folder 'database/seeds/tenant'.

The same as migration parameters, but for tenants:seed and the SeedDatabase job.

5.2 Cloud Infrastructure Setup

This section outlines the steps for configuring cloud infrastructure, including selecting cloud services like ECS, RDS, and object storage, and other required cloud services step by step. For this part of the project we will use Alibaba Cloud Services.

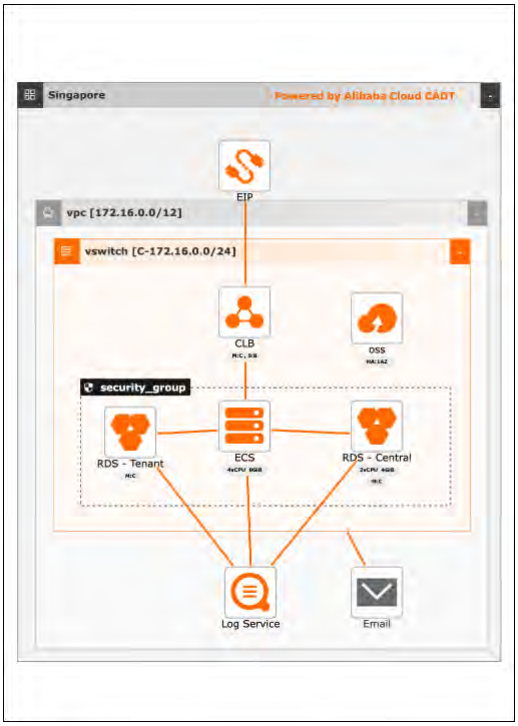


Figure 5.2: Cloud Service Architecture

To create an Alibaba Cloud account, visit the alibabacloud.com. Once verified, your account will be activated, granting access to Alibaba Cloud services and the management console.

5.2.1 Setting an ECS Instance

Go to the ECS console. Perform the following steps to create or select the basic resources required to create an ECS instance.

1. Select a region and a billing method.
   Region: Singapore
   Billing Method: Subscription

2. Select an instance type and an image
   Instance: 2 vCPU 2GB Memory
   Operating System: CentOS

3. Configure disks in the Storage section
   ESSD: 20GB

4. Assign a public IP or EIP

5. Create a security group
   Open IPv4 Ports/Protocols: select SSH (TCP:22), RDP (TCP:3389), HTTP (TCP:80), and HTTPS (TCP:443).

6. Create a key pair You can bind key pairs to ECS instances and use the key pairs as security credentials to authenticate your identity when you log on to the instances.

7. Create and view an ECS instance

The above mention selections are used to run minimal application. Configure parameters based on business requirements.

5.2.2 MySQL instance and configure databases

Go to the ApsaraDB RDS

1. Select a region and a billing method.
   Region: Singapore
   Billing Method: Subscription

2. Configure the Database Engine, Edition, Product Type, and Storage Type parameters.
   RDS Basic Edition with MySQL 7.2.*

3. Configure the Storage Type parameter as General ESSD (recommend)

4. Configure the Storage Capacity parameter as 10GB

Go to the RDS Instances page. In the top navigation bar, select the region in which your RDS instance resides. Then, find the RDS instance. Using the cloud console you can create desired databases and accounts.

5.2.3 Other Cloud Services

Log Service

Log Service is a cloud-native observability and analytics platform that provides large-scale, low-cost, and real-time services to process multiple types of data such as logs, metrics, and traces. Simple Log Service allows you to collect, transform, query, analyze, visualize, ship, and consume data. You can also configure alert rules in the console. Simple Log Service helps you improve digital capabilities in RND, OnM and, data security.

Object Storage Service

OSS is a secure, cost-effective, and highly reliable cloud storage service that allows you to store large amounts of data. OSS provides platform-independent API operations, which allow you to upload and access your data from any application, anytime, and anywhere.

Domain Management

Domain Names is a domain name management platform that provides domain name registration, transaction, monitoring, and protection services. This will ease the domain name server management and subdomain creation.

5.3 Deployment and Go-Live

This section summarizes the final steps to deploy and launch the SaaS application into the cloud server.

5.3.1 Connecting ECS with Git

The code deployment can be very challenging without connecting the github account with the hosting server. Also the CI-CD operations cannot be accommodated without this connection. The ECS instance was configured to pull the application code via SSH. After setting up SSH keys between ECS and Git server:

```
$ ssh-keygen -t rsa
$ cat ~/.ssh/id_rsa.pub
```

The public key was added to the Git account, and the project was cloned into the ECS web root:

```
$ git clone git@github.com:user/project.git /var/www/saas-app
```

5.3.2 Connecting ECS with RDS

Database credentials for the ApsaraDB RDS instance were added to the .env file. Both central and tenant databases were configured, allowing Laravel to switch context based on tenant identification. Laravel's default DB connection was updated to point to the central database, and the tenancy package handled runtime binding to tenant databases.

```
DB_CONNECTION=mysql
DB_HOST=<rds-host>
DB_DATABASE=central_db
DB_USERNAME=admin
DB_PASSWORD=********
```

5.3.3 Domain Redirection

Tenant identification was implemented using subdomains. In the Alibaba Cloud Domain Management console, wildcard DNS records (*.mysaasapp.com) were configured to point to the ECS public IP. The Laravel middleware InitializeTenancyByDomain ensured each subdomain request was mapped to the correct tenant context.

5.3.4 Additional Deployment Requirements

To effectively utilize the multi-tenant behavior the following configuration should be considered as well.

- Event System and Job Pipelines: Tenant creation events automatically triggered jobs like CreateDatabase, MigrateDatabase, and SeedDatabase, coordinated through Laravel's queue system.

- TenancyServiceProvider: Registered in config/app.php to hook into Laravel's lifecycle.

- Tenancy Bootstrappers: Services like Cache, Filesystem, Database, and Queue were scoped per tenant via Tenancy bootstrappers[22].

Chapter 6

Benefits and Impact of Multi-Tenant SaaS Conversion

This chapter will highlight the advantages and positive outcomes of converting a silo web application into a multi-tenant cloud-based SaaS application. It will demonstrate how the project addresses the challenges outlined earlier and provides value to both application providers and end users.

6.1 Cost Efficiency

Transitioning to a cloud infrastructure significantly reduces the infrastructure costs. While on-premise infrastructure requires an initial investment, cloud infrastructure eliminates any such investment. Table 6.1 shows the capital expenses in a high-level cost quotation.

Particulars	**On Premise Cost**	**Cloud Cost**
Server Infrastructure (CapEx)	$6,000	-
Network Hardware (CapEx)	$3,000	-
Utility Infrastructure (CapEx)	$4,000	-
Total Capital Expense	**$13,000**	**$0**

Table 6.1: On-Premise Infrastructure Setup Capital Expense

Cloud services typically yield approximately 30 percent operational cost savings compared to on-premise servers by shifting expenditures from upfront capital expenses to usage-based operational costs.

Additionally, cloud providers offer usage-based pricing policy that scale with actual server activity. This option is extremely beneficial for applications which have periodical high traffic. On-premise infrastructure requires capacity planning for peak usage, leading to potential over-provisioning and unnecessary costs, while cloud infrastructure scales flexibly with demand, ensuring predictable and optimized expenses.

Table 6.2 shows an equal comparison for an application with one million traffic per hour.

Particulars	On Premise Cost	Cloud Cost
ECS Subscription	-	$12,000
RDS Subscription	-	$8,000
Other Subscriptions	-	$2,000
Amortized Annual cost for infrastructure	$1,600	-
Electricity and Cooling	$6,000	-
Staff and IT Administration (salary)	$30,000	$10,000
Software Licenses	$4,000	-
Logging/Monitoring Services	$3,000	$1,500
Networking and Internet Connectivity	$5,000	$2,000
Load Balancer and NAT Gateway	$1,000	$1,500
Backups and DR	$3,000	$1,200
Annual Cost	$53,600	$38,200
5-Year Total Cost	**$268,000**	**$191,000**
Differences	$77,000	

Table 6.2: Operational Cost for On-Premise (Annual) versus AWS Cloud (Annual)

The cost analysis clearly shows opting for cloud-based service is the most price-effective and sustainable solution for operating SaaS service.

6.2 Scalability and Flexibility

Scalability and flexibility are the cornerstones of modern applications. Cloud computing provides companies with the opportunity to adapt to the changing market conditions. The targeted growth and innovation can be secured only by the cloud infrastructure without significant investment. Flexibility refers to the ability to quickly and easily adjust IT infrastructure to meet current needs, without significant financial investment. Scalability, on the other hand, allows for the dynamic increase or decrease of resources (such as computing power or storage) in response to changing business requirements.

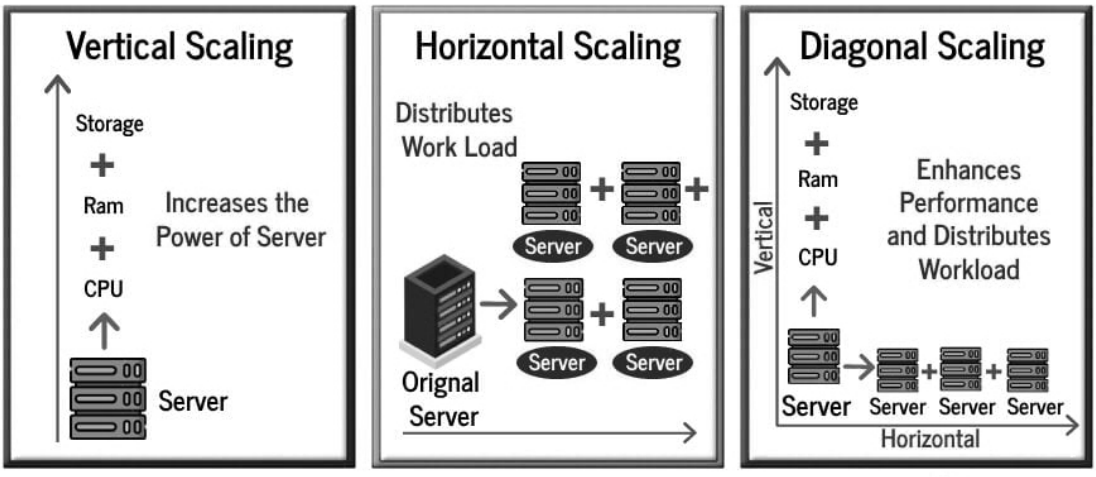


Figure 6.1: Cloud Computing Scaling [23]

For example, the mentioned SaaS product is subscribed to by a company that re-

quired database segregation and separate hosting. These sorts of requirements can be easily met by the cloud service flexibility option. Another example could be the report service is blocking the other write queries at the end of each month. The slow query issue can be resolved by either increasing the managed DB instance CPU sizes or creating a separate read DB only. If we want to implement such solutions within the on-premise server either it will require significant investment or it will need resource sharing within the existing resources. As S. Kumar et al. have described the advantages of cloud scalability; some key benefits include performance optimization, flexibility, adaptability, improved reliability, maximizing utilization, and global reach[23].

Along with the cloud native benefits the multi-tenant architecture offers additional points in terms of scalability and flexibility. The approach will help the SaaS providers to reach a broader market along with the flexibility to ensure proper resource utilization, reliable infrastructure and guaranteed up-time.

6.3 Improved Maintenance and Updates

The multi-tenant model significantly simplifies software maintenance compared to traditional monolithic model. Since multiple tenants share the same application and database instance, updates, bug fixes, and security patches can be applied centrally and immediately reflected across all tenants. This centralized maintenance process drastically reduces the overhead typically involved in managing separate deployments for each client. Consequently, maintenance tasks become far less complicated, ensuring better software reliability.

The following table captures the key characteristics of multi-tenancy, along with benefits and challenges.

In comparison, Multi-tenancy provides significant advantages over monolithic architecture. The difference is easily visible when there are dozen of tenants deployed in isolated infrastructure needs to be maintained. The operation is quite cumbersome for an individual and prone of human error. Automated backup and restoration mechanisms ease the infrastructure-related fault tolerance and improve application maintainability. However, these benefits hinge upon properly addressing the challenges of tenant isolation and data security, underscoring the necessity of strategic design and continuous monitoring in multi-tenant SaaS.

Key Characteristic	Benefits	Challenges
Hardware Resource Sharing	- Higher utilization of hardware and infrastructure. - Reduced overall hosting cost.	- Performance bottlenecks if one tenant consumes excessive resources. - Resource contention risks.
High Degree of Configurability	- Tenants perceive the application as dedicated. - Supports diverse business requirements.	- Increases code complexity. - Requires robust configuration management.
Shared Application Instance	- Simplified deployment and centralized updates. - Lower operational and maintenance costs.	- Any defect can affect all tenants. - Requires strict version control and testing.
Shared Database (with isolation)	- Easier data aggregation and analytics. - Reduced storage overhead.	- Strong need for tenant isolation to prevent data leakage.
Centralized Maintenance and Deployment	- Single point for applying updates, patches, and fixes. - Faster rollout of new features.	- Risk of widespread impact during failures. - Zero-downtime deployment requirements.
Economy of Scale	- Lower total cost of ownership (TCO) across tenants. - More competitive pricing for SMEs.	- Initial reengineering cost to migrate monolithic apps. - Requires well-designed multi-tenant architecture.

Table 6.3: Key Characteristics, Benefits, and Challenges of Multi-Tenancy

6.4 Business Benefits for Application Providers

In the modern software economy, scalability is more than just a technical benchmark—it's a business imperative. From startups to government agencies, organizations are racing to deliver secure, reliable, and cost—efficient digital services to users across the globe. One architectural approach stands out in this race: multi-tenancy. Multi-tenant SaaS architecture offers significant strategic and operational advantages for SaaS application providers. By allowing multiple customers to share the same infrastructure and application instance while maintaining isolation and configurability, providers can achieve economies of scale, reduce operational overhead, and deliver services more efficiently. This approach not only optimizes resource utilization and lowers total cost of ownership but also accelerates feature deployment and enhances market competitiveness.

Multi-tenant SaaS architecture offers significant strategic and operational advantages for application providers. By allowing multiple customers to share the same infrastructure and application instance while maintaining isolation and configurability, providers can achieve economies of scale, reduce operational overhead, and deliver services more efficiently. This approach not only optimizes resource utilization

and lowers total cost of ownership (TCO) but also accelerates feature deployment and enhances market competitiveness.

6.4.1 Efficiency at Scale

Multi-tenant SaaS architecture enables providers to manage a single shared codebase and infrastructure across multiple tenants, eliminating the need to maintain separate environments. This dramatically lowers overhead for updates, patching, and version control, allowing engineering teams to focus on development rather than deployment

6.4.2 Lower Total Cost of Ownership (TCO)

Service providers benefit from significant cost savings through resource pooling. Shared hardware, licensing, and maintenance costs deliver better economies of scale, resulting in reduced operational costs and competitive pricing for customers.

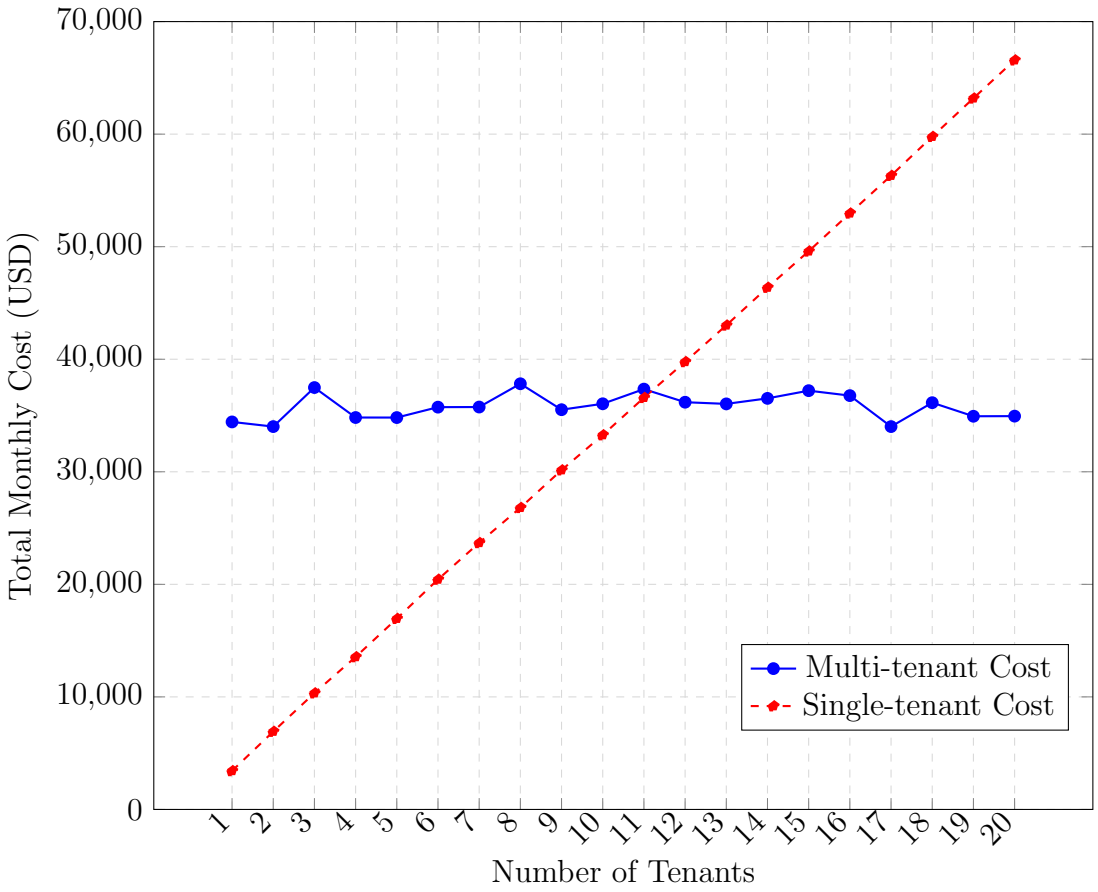


Figure 6.2: Comparison of Multi-tenant and Single-tenant SaaS Costs

6.4.3 Elastic Resource Utilization

Multi-tenancy supports elastic scaling—allocating compute, storage, and network resources dynamically based on demand. Providers can accommodate variable traffic without over-provisioning, maintaining high performance while minimizing wasted capacity.

Auto Scaling helps vendor to ensure the infrastructure availability to handle the incoming traffic. If you specify scaling policies, then Amazon EC2 Auto Scaling can launch or terminate instances as demand on your application increases or decreases.

For example, the following Auto Scaling group has a minimum size of four instances, a desired capacity of six instances, and a maximum size of twelve instances. The scaling policies that you define adjust the number of instances, within your minimum and maximum number of instances, based on the criteria that you specify.

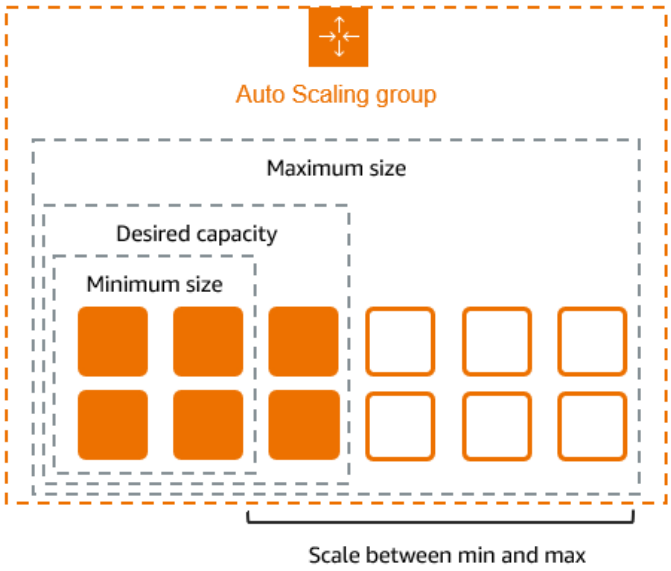


Figure 6.3: Cloud Service Auto Scaling Group

There are no additional fees with Amazon EC2 Auto Scaling, therefore users only pay for the instances they use.

6.4.4 Competitive Advantage and Market Reach

By reducing infrastructure costs and enabling rapid scaling, providers can offer more competitive pricing and reach diverse market segments—from SMEs to enterprise clients—without significant operational burden. This also allows providers to allocate more resources to innovation and customer success.

Multi-tenancy enables providers to serve diverse market segments efficiently, from small businesses to large enterprises, without incurring the operational overhead of separate infrastructure for each customer. Cloud infrastructure providers such as Amazon Web Services further enhances this advantage through their global network of Availability Zones. These AZs allow application providers to strategically host their multi-tenant instances in regions closest to their customers, reducing latency and improving performance. Moreover, providers can adopt a deployment strategy that allocates tenants to specific regions or zones based on client size, compliance requirements, and service-level agreements[18].

This flexibility not only improves user experience but also enables providers to expand into new markets quickly while maintaining cost efficiency and operational

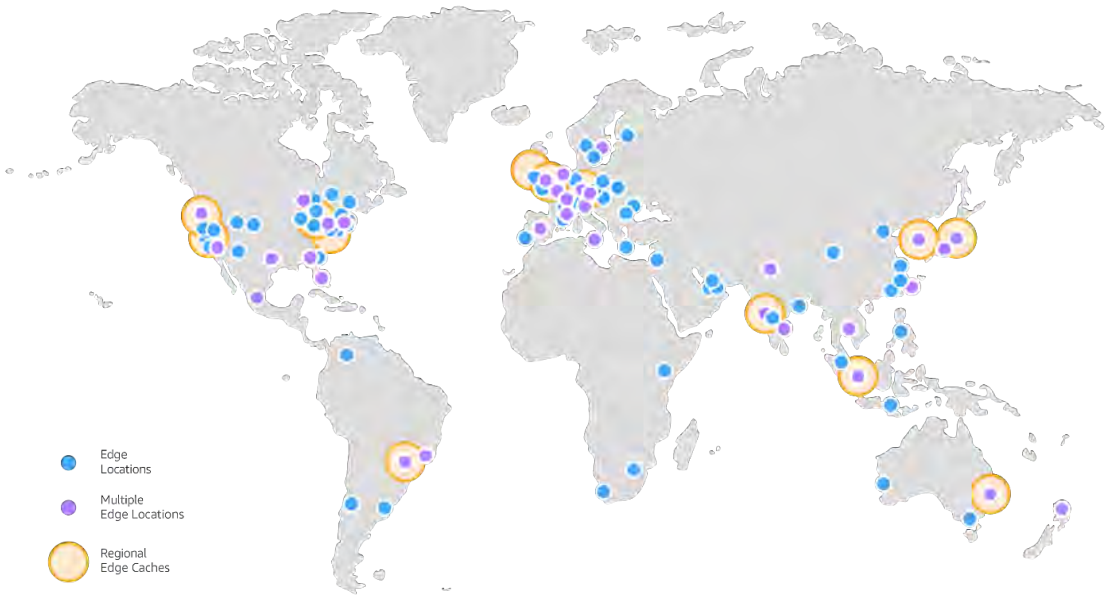


Figure 6.4: AWS Points of presence[18]

consistency across the globe.

The transition to a multi-tenant SaaS model delivers measurable advantages in operational efficiency, scalability, and cost optimization. By consolidating resources and centralizing maintenance, application providers can reduce infrastructure overhead while improving update cycles and service consistency. As Bezemer noted , "The major benefits of multi-tenancy are increased utilization of hardware resources and improved ease of maintenance... These benefits should result in lower overall application costs."[4]. Furthermore, the inherent scalability and flexibility of multi-tenant systems allow businesses to meet variable demand efficiently, improve customer experience, and maintain competitiveness in a global market. Ultimately, multi-tenancy supports sustainable growth by aligning technical efficiency with strategic business objectives.

Chapter 7

Conclusion and Future Works

This project set out to design and implement a practical, structured approach for converting a silo-based web application into a multi-tenant, cloud-based SaaS application using Laravel. The work was carried out over approximately six months during my official duties, in collaboration with software engineers under my leadership. As the team lead, I was responsible for making key architectural decisions on how multi-tenancy could be effectively implemented, ultimately deciding to document the process as part of this academic research to benefit the wider community.

The proposed methodology addressed critical challenges such as tenant identification, database isolation, application re-engineering, and cloud service integration. Along with the tenancy packages, the conversion process demonstrated how existing ASP model applications can be modernized for scalability, cost-efficiency, and easier maintenance. Furthermore, cloud infrastructure was utilized to maximize resource utilization, ensure high availability, and enhance system security.

Future works

In the next iterations, Nginx scripts, advanced caching mechanisms, and CI-CD practices can be implemented to reduce manual interventions. Automating subdomain creation and registration through an Nginx script stream-lines tenant onboarding in a multi-tenant SaaS environment. Implementing advanced caching mechanisms (e.g., Redis, Memcached) will reduce database load and improve response times.Adopting CI/CD practices using platforms such as Jenkins and GitHub Actions can streamline automated deployment and updates for multi-tenant SaaS applications, ensuring faster release cycles with consistent quality and minimal downtime.

While the project achieved its primary objective, certain improvements and enhancements are open for future researchers to explore. As SaaS adoption continues to evolve; emerging technologies, shifting market demands, and regulatory considerations will present both opportunities and challenges for application providers. Future work should focus on enhancing scalability through advanced deployment techniques, improving tenant-specific customization, implementing microservice architecture, RBAC implementation, and integrating intelligent monitoring to optimize performance. Incorporating DC-DR (Data Center–Disaster Recovery) techniques such as real-time replication and automated fail-over ensures business continuity during outages.

The adoption of multi-tenant cloud implementation not only reduces operational overhead but also empowers application providers to reach broader markets, streamline updates, and offer competitive pricing models. This project reaffirms that, when properly designed and managed, multi-tenancy can deliver significant technical and business value, marking a decisive step toward sustainable and scalable SaaS delivery in the modern software landscape.

Bibliography

[1] H. J. Smith, S. J. Milberg, and S. J. Burke, "Information privacy: Measuring individuals' concerns about organizational practices," *MIS quarterly*, pp. 167–196, 1996.

[2] M. Fowler, "Stranglerfigapplication," *Website. Letzter Zugriff*, vol. 19, p. 2019, 2004.

[3] J. Kerievsky, *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.

[4] C.-P. Bezemer and A. Zaidman, "Multi-tenant saas applications: Maintenance dream or nightmare?" In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, ser. IWPSE-EVOL '10, Antwerp, Belgium: Association for Computing Machinery, 2010, pp. 88–92, ISBN: 9781450301282. DOI: 10.1145/1862372.1862393. [Online]. Available: https://doi.org/10.1145/1862372.1862393.

[5] C. Gang, "Data center management plan in cloud computing environment," in *2010 3rd International Conference on Information Management, Innovation Management and Industrial Engineering*, IEEE, vol. 4, 2010, pp. 393–396.

[6] A. Khajeh-Hosseini, D. Greenwood, and I. Sommerville, "Cloud migration: A case study of migrating an enterprise it system to iaas," in *2010 IEEE 3rd International Conference on Cloud Computing*, IEEE, 2010, pp. 450–457. DOI: 10.1109/CLOUD.2010.37.

[7] F. Trucchia, J. Romei, and A. Saray, *Pro PHP refactoring*. Springer, 2010.

[8] W. Kim, J. H. Lee, C. Hong, C. Han, H. Lee, and B. Jang, "An innovative method for data and software integration in saas," *Computers and Mathematics with Applications*, vol. 64, no. 5, pp. 1252–1258, 2012.

[9] V. Cho and A. Chan, "An integrative framework of comparing saas adoption for core and non-core business operations: An empirical study on hong kong industries," vol. 17, no. 3, pp. 629–644, 2013. DOI: 10.1007/s10796-013-9450-9. [Online]. Available: https://doi.org/10.1007/s10796-013-9450-9.

[10] O. Zimmermann, "Architectural refactoring: A task-centric view on software evolution," *IEEE Software*, vol. 32, no. 2, pp. 26–29, 2015.

[11] J. P. G. Gashami, Y. Chang, J. J. Rho, and M.-C. Park, "Privacy concerns and benefits in saas adoption by individual users: A trade-off approach," *Information Development*, vol. 32, no. 4, pp. 837–852, 2016.

[12] A. Furda, C. Fidge, A. Barros, and O. Zimmermann, "Reengineering data-centric information systems for the cloud–a method and architectural patterns promoting multitenancy," in *Software Architecture for Big Data and the Cloud*, Elsevier, 2017, pp. 227–251.

[13] A. Christoforou, L. Odysseos, and A. S. Andreou, "Migration of software components to microservices: Matching and synthesis," 2019.

[14] E. Wolff, "Migrating monoliths to microservices: A survey of approaches," in *International Conference on Microservices*, vol. 2019, 2019.

[15] A. Mondal, S. Paul, R. T. Goswami, and S. Nath, "Cloud computing security issues challenges: A review," in *2020 International Conference on Computer Communication and Informatics (ICCCI)*, 2020, pp. 1–5. DOI: 10.1109/ICCCI48352.2020.9104155.

[16] H. Shah, *How google moved beyond search to reinvent productivity with g suite*, Mar. 2020. [Online]. Available: https://nira.com/g-suite-history/.

[17] S. Choubey and L. Goasduff, *Gartner says cloud will become a business necessity by 2028*, Nov. 2023. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2023-11-29-gartner-says-cloud-will-become-a-business-necessity-by-2028#:~:text=Most%20companies%20currently%20consider%20the,to%20accelerate%20their%20business%20initiatives..

[18] M. Haken, "AWS fault isolation boundaries," Tech. Rep., Feb. 2023, p. 6. [Online]. Available: https://docs.aws.amazon.com/pdfs/whitepapers/latest/aws-fault-isolation-boundaries/aws-fault-isolation-boundaries.pdf#points-of-presence.

[19] E. T. Nordli, S. G. Haugeland, P. H. Nguyen, H. Song, and F. Chauvel, "Migrating monoliths to cloud-native microservices for customizable saas," *Information and Software Technology*, vol. 160, p. 107 230, 2023.

[20] *Laravel usage statistics by builtwith*, Mar. 2024. [Online]. Available: https://trends.builtwith.com/framework/Laravel.

[21] T. Ripe, *G suite is history – please welcome google workspace!* Jan. 2024. [Online]. Available: https://google.thecloudpeople.com/blog/g-suite-is-history-please-welcome-google-workspace.

[22] S. Štancl, *Tenancy for laravel documentation*, Apr. 2024. [Online]. Available: https://tenancyforlaravel.com/docs/v3/introduction/.

[23] S. Kumar, *Scaling in cloud computing*, Jun. 2025. [Online]. Available: https://www.educba.com/scaling-in-cloud-computing/.

[24] Nxt, *The Power of Multi-Tenancy: Scaling Smarter in the Cloud Era*, Apr. 2025. [Online]. Available: https://nxt1.cloud/blog/cloud-architecture/multi-tenancy-saas-architecture.