Instituto Politécnico de Tomar

Escola Superior de Tecnologia de Tomar

Shoaib Feda

Multi-level Nested Multi-tenancy in Web Software Engineering

Relatório de estágio

Estágio orientado por:

Prof. Doutor Luís Miguel Lopes de Oliveira - Instituto Politécnico de Tomar

Prof. Doutor Renato Eduardo Silva Panda - Instituto Politécnico de Tomar

Engeneiro Masood Ahmad – Ebtikarat for Investment

Relatório apresentado ao Instituto Politécnico de Tomar
para cumprimento dos requisitos necessários à obtenção do
grau de Mestrado em Engenharia Informática – Internet das Coisas

"الإبداع خيال، والخيال تمرد..."

“Creativity is imagination, and imagination is rebellion…”

- Arabic Proverb

ABSTRACT

An internship was done at a company that ran a suite of in-house built SaaS applications. Since many of the different applications needed the same features such as authentication, authorization, subscription, database management, etc., a single core macroservice was built to serve all the different applications and their different tenants from a single codebase (application). This greatly reduced time-to-market for new SaaS projects and enabled teams to focus more on business and domain features with an estimated 80% reduction in time spent on maintenance and development of the commonly shared modules. This greatly increased productivity among engineer teams and a subjective survey of 12 engineers of various ages and experience levels indicating high satisfaction levels and increase in productivity. The primary focus was to build a solution that provides data isolation for customers which is why a nested multi-level multi-tenant, multi-database architecture was used, providing a isolated databases, cache stores, queues and logs for each customer. A no-estimation mindset was used in a hybrid kanban-agile scrum methodology which helped achieve more accurate projections with regards to time to complete projects. Furthermore, asynchronous PHP was explored to boost performance were usable which resulted in 54% increase in number of requests that can be handled given the same resources for some endpoints that can make use of concurrency. The language and framework of choice was PHP Laravel and different ways of achieving multi-tenancy in it was discussed, weighed and some implemented.

Keywords: Multi-tenancy, SaaS, server, backend, multi-database, multi-client, isolated databases, isolated data stores, core API, nested multi-tenancy, multi-tenant, shared SaaS application, shared application, shared server, shared core API, async PHP, Laravel Octane, Swoole, asynchronous request processing, event loops, concurrency, no-estimates, no-estimation, kanban, agile, scrum, storyboard, story points, epics, user stories

RESUMO

Foi realizado um estágio numa empresa que geria um conjunto de aplicações SaaS desenvolvidas internamente. Como muitas das diferentes aplicações necessitavam dos mesmos recursos, como autenticação, autorização, assinatura, gestão de base de dados, etc., foi construído um macroserviço central único para servir todas as diferentes aplicações e os seus diferentes locatários a partir de uma única base de código (aplicação). Isto reduziu consideravelmente o tempo de lançamento no mercado de novos projetos SaaS e permitiu que as equipas se concentrassem mais nos negócios e nos recursos de domínio, com uma redução estimada de 80% no tempo gasto na manutenção e no desenvolvimento dos módulos comummente partilhados. Isto aumentou muito a produtividade entre as equipas de engenheiros e um inquérito subjetivo a 12 engenheiros de diferentes idades e níveis de experiência indicou níveis elevados de satisfação e aumento da produtividade. O foco principal foi construir uma solução que fornecesse isolamento de dados para os clientes, e é por isso que foi utilizada uma arquitetura multi-inquilino e multi-base de dados aninhada, fornecendo bases de dados isoladas, armazenamentos de cache, filas e registos para cada cliente. Foi utilizada uma mentalidade de não estimativa numa metodologia híbrida kanban-ágil scrum que ajudou a obter projeções mais precisas em relação ao tempo para a conclusão dos projetos. Além disso, o PHP assíncrono foi explorado para aumentar o desempenho utilizável, o que resultou num aumento de 54% no número de pedidos que podem ser tratados com os mesmos recursos para alguns endpoints que podem fazer uso de simultaneidade. A linguagem e framework escolhida foi o PHP Laravel e diferentes formas de conseguir multilocação no mesmo foram discutidas, ponderadas e algumas implementadas.

Palavras-chave: Multi-tenancy, SaaS, server, backend, multi-database, multi-client, isolated databases, isolated data stores, core API, nested multi-tenancy, multi-tenant, shared SaaS application, shared application, shared server, shared core API, async PHP, Laravel Octane, Swoole, asynchronous request processing, event loops, concurrency, no-estimates, no-estimation, kanban, agile, scrum, storyboard, story points, epics, user stories

ACKNOWLEDGMENTS

I hereby thank my father, mother & family for their endless support and love throughout my journey in university.

I also thank my supervisors Professor Luis Oliveira, Professor Renato Panda and Engineer Masood Ahmad for giving me the chance to collaborate with them and providing assistance and guidance whenever needed. It has been an honour collaborating with them.

INDEX

INDEX OF FIGURES

INDEX OF TABLES

INDEX OF GRAPHS/CHARTS

LIST OF ACRONYMS

API – Application Programming Interface

DX – Developer Experience

JSON – Javascript Object Notation

MVC – Model-view-controller

OS – Operating System

OTP – One Time Password

REST – Representational State Transfer

SaaS – Software as a Service

1. INTRODUCTION

1.1 Motivation and Context

With the ever-increasing demand for SaaS applications, engineers find themselves rebuilding the same features for many different projects, often within the same company/team. This leads not only to waste of effort and time during the initial development phase, but also to an increase in maintenance efforts at later stages during the lifetime of the project. A single security fix or a shipment of a new feature for the common modules now has to be done separately for every single application in the company/team. Modules such as authentication, authorization, subscription, invoicing, notifications, etc. often are exactly the same across different applications in different domains. They seldom require customization and when they do, these customization requirements are often minimal as the functionality of such modules are pretty well defined over the years by the increasingly inter-connected and integrated industry. Hence, a unified solution that aims at serving multiple SaaS applications designed to reduce engineering efforts and be cost-effective becomes a critical requirement, especially for companies with small teams and limited engineering capacity.

1.2 Problem Statement

There are essentially three main challenges with a traditional architecture in which each SaaS application includes its own modules:

- **Initial development effort:** Building full-fledged authentication, authorization, subscription, invoicing, notifications, etc. modules for each SaaS project is no minor task. Although a common boilerplate could be utilized to reduce this problem in particular, that in itself introduces new issues such as package management, versioning and backward compatibility difficulties, programming language and framework restrictions and more
- **Maintenance overhead & cost:** Maintaining what is essentially “duplicated code” across multiple repositories very quickly becomes challenging. A single bug or new feature needs to be worked multiple times often across multiple teams and environments. This intern also means increased cost of maintenance due to human labour and wasted engineering capacity that could’ve otherwise been utilized elsewhere
- **Security vulnerabilities:** Similar to maintenance overhead, security also becomes a repetitive and increasingly requiring task. More importantly, maintaining multiple codebases potentially could mean delayed delivery of security patches

1.3 Objectives

The objective of this project is to develop a core “macroservice” (as supposed to microservice) that services common functions for multiple tenants (SaaS applications) and their own sub-tenants (hence, the term “multi-level nested multi-tenancy") using a REST API in a secure, isolated, maintainable and scalable manner. The main goal is to achieve reduction in development and maintenance efforts and costs.

1.4 Scope of Work

This project will focus on solving the issue of effort duplication for an engineering team building multiple cloud native web SaaS applications by building a core API service. The core API service and the applications operate in a microservices architecture using similar protocols for inter-service communication. We will also cover using asynchronous PHP to boost performance for some endpoints and carryout some performance benchmarks. The details and specifics of client applications is irrelevant and will not be covered in this report. Challenges arising from the use of a microservices architecture such as monitoring, observability, error tracing, scaling, etc. are also out of scope and will not be covered. We will also cover the concept of no-estimates and how they can be useful to have accurate projections on software engineering projects compared to estimating story points.

1.5 Structure of Report

This report is divided into 6 chapters:

- Chapter 1: Introduction to the project completed during the internship
- Chapter 2: State-of-the-art, where the work done is compared to other similar academic projects
- Chapter 3: Workflow Strategy, where the team’s work methodology is discussed
- Chapter 4: System design and architecture, where the requirements and system parameters are discussed
- Chapter 5: Methodology, where the implementation is discussed in detail
- Chapter 6: Results, where the built software is tested and utilized in real world applications
- Chapter 7: Conclusion and future work, where the conclusions, achievements, future opportunities and possibilities are discussed

2. STATE-OF-THE-ART

The problem of rewriting and maintaining the same common features across multiple applications is not a new phenomenon and has existed since the early days of software engineering, especially with the rise of cloud-based SaaS applications. Many solutions have been developed over the years and one in particular, multi-tenancy, solves two problems at the same time. It provides a centralized way of managing code changes while also enabling different tenants or clients to share resources in the pursuit of cost savings.

2.1 Overview of Multi-Tenancy in SaaS

Multi-tenancy enables multiple clients or “tenants” to share infrastructure significantly reducing cost per tenant among other benefits. This architecture is not limited to SaaS applications and was a common practice since the early days of computing where companies in the 1960s rented space and computing power on mainframe computers (time-sharing) to reduce computing expenses [1].

Today, multi-tenant architecture is a very common practice in the SaaS industry due to the importance of cost savings, centralized management, data isolation and ease of maintenance.

2.2 Types of Multi-Tenant SaaS Architectures

There are mainly two different models of multi-tenant architectures in the web SaaS application domain each with their own pros and cons [2]. Identified mainly by their approach to data segregation and management, these are:

- **Shared Data Storage**
  In this model, different tenants store their data within the same database, cache, filesystem, etc. and are identified using identifiers such as tenant ids. Advantages include maximum resource sharing, reduced complexity and reduced cost. The main disadvantage is the risk of data leakage between clients.

users		
id	**tenant_id**	**name**
1	1	Shoaib
2	2	Suhail
3	1	Pedro
4	1	Maria
5	3	Mohammad
6	3	Sultan

Figure 1: Example of a table in a shared database multi-tenanct architecture

- **Isolated/Segregated Data Storage**
  In this model, different tenants store their data in isolated databases, cache instances and filesystems (and/or partitions). This provides the maximum data segregation and greatly reduces risk of data leakage between tenants. It does however come with its own disadvantages such as increased complexity in maintaining data sources.

users (tenant 1)	
id	**name**
1	Shoaib
2	Pedro
3	Maria

users (tenant 2)	
id	**name**
1	Suhail

users (tenant 3)	
id	**name**
1	Mohammad
2	Sultan

Figure 2: Example of a table in an isolated multi-tenant architecture

Of course, it is also possible to have a hybrid approach with different levels of isolation in terms of database, cache system and file system.

2.3 Nested Multi-Tenancy

If we consider an accounting SaaS product as an application, then the companies or the customers that use application would be the tenants. So each application would have multiple tenants. This is the most common usecase for multi-tenant architecture in SaaS development.

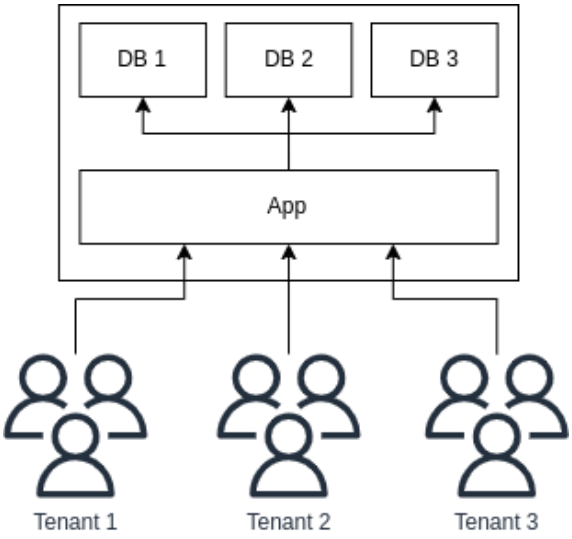


Figure 3: Isolated Database, Multi-tenant Architecture

But let’s consider another possibility. What if our application itself was also a tenant? A tenant of say a core backend service that provides common functionality such as authentication, authorization, subscription, etc. to other applications or “tenants”. If we put these two pieces together, we would essentially have multiple tenants that have multiple tenants themselves, i.e. multi-level nested multi-tenancy.

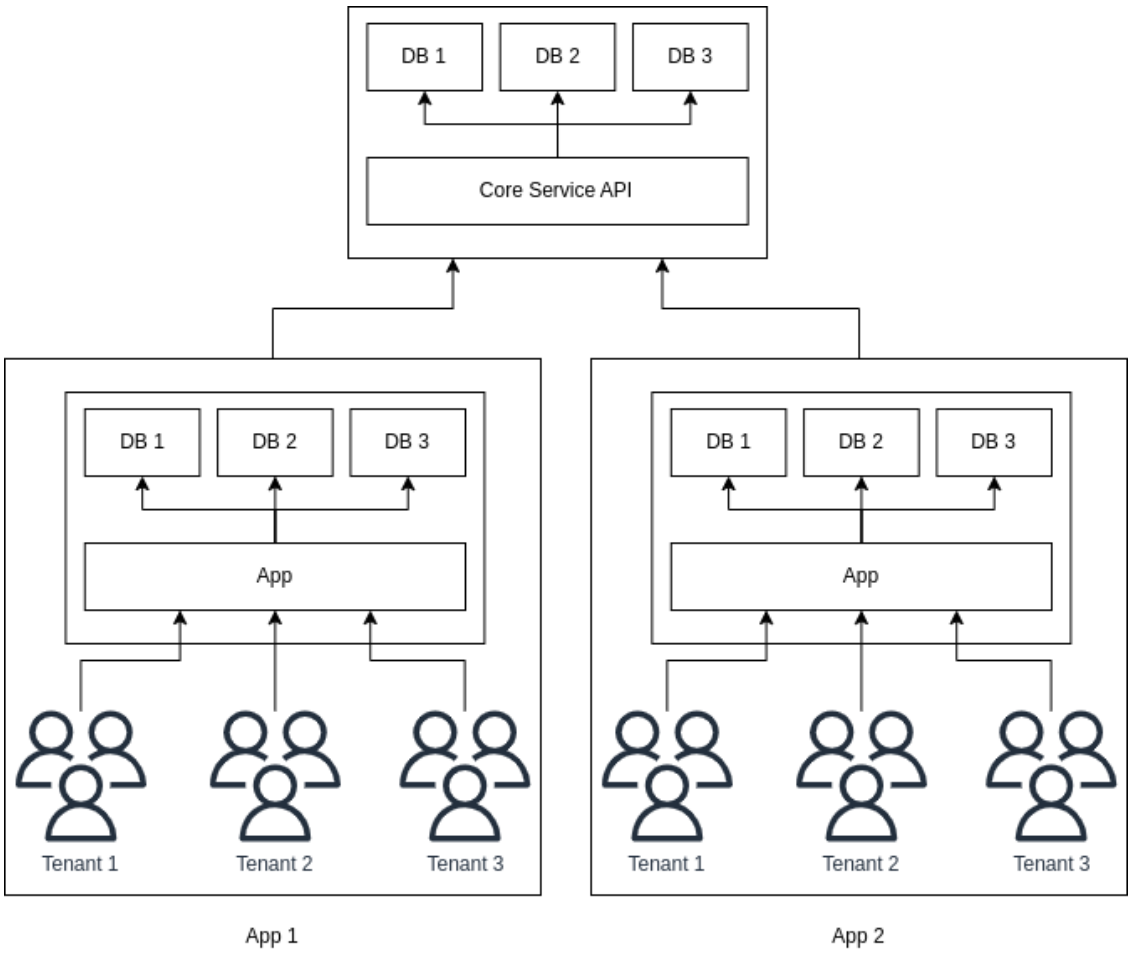


Figure 4: Multi-level nested multi-tenant architecture

Multi-level nested multi-tenancy like many things comes with its own advantages and disadvantages. However, it can really shine given an environment where a relatively small team manages multiple SaaS applications. It would help reduce development costs by reducing the amount of code that needs to be maintained, operating costs by taking advantage of resource sharing and would save the team a lot of unnecessary duplicated maintenace work.

Furthermore, although it might not be obvious, this is essentially the architecture we achieve when we integrate third-party APIs that also have multi-tenancy into our own app that in itself is a single level multi-tenant architecture.

2.4 Existing Solutions and Limitations

For the purposes of this internship at the company, the focus was only on solutions related to PHP Laravel as backend technology. For PHP Laravel, there exists quite a few libraries & opensource repositories that aim to provide multi-tenancy. These mainly are:

- **“stancl/tenancy”**
  stancl/tenancy is a flexible multi-tenancy Laravel package that comes with lots of features out-of-the-box and doesn't stand in your way when you need anything custom [3]. It provides many features such as automatic data separation, single or multi-database, event based architecture, shared users, etc. It can be utilized to achieve multi-tenancy in the client apps. However, it does not provide multi-level nested multi-tenancy which means it cannot be used to in core API service as there we need a nested multi-tenancy in which the core service hosts applications and their sub-tenants.
- **“spatie/laravel-multitenancy”**
  Similar to stancl/tenancy, laravel-multitenancy by spatie also provides very similar features and is a great package to start with. However, it also does not support multi-level nested multi-tenancy.

The problem essentially is that these packages are designed for the same application to have multiple tenants. The problem on hand is that the core service needs to “pretend” to be the client application. In other words, it needs to impersonate the client application and configure it self accordingly when it comes to application name, emails, links, etc. Therefore, the core service almost at all times needs to be aware of 3 things when it receives a request, which application is using it, what tenant of that application it is serving and what user is currently causing the request. This is because it is required to store data related to the services it provides in an isolated manner for each application and each tenant.

2.5 Summary

The current solutions do not suffice the requirement for the company and hence a project was initiated to design, build, test and deploy a core API service that services multiple applications with their sub-tenants in a multi-tenant architecture.

3. WORKFLOW STRATEGY

Over the years working in agile scrum teams, several challenges were identified with the traditional agile scrum methodology, particularly regarding the estimation of efforts for tasks (tickets). Engineers often estimated tickets either above or below their actual effort, leading to inaccuracies and inefficiencies in the workflow and almost always a guaranteed delay in the project delivery. These estimation inaccuracies often stemmed from the inherent unpredictability of software engineering tasks, where even seemingly simple issues can turn out to be unexpectedly complex. Furthermore, due to the repeated frustration of product management every time a project deadline was not met, software engineers by time found themselves ever so virtually overestimating effort for tickets in order to have a “safe buffer zone”. This meant that many a times the team was not working at full capacity. To address these issues, the team opted for a more Kanban-based approach when it comes to the planning part of the workflow, wherein engineers finish one ticket and immediately move to the next without the constraints of pre-planned sprint cycles. This continuous workflow minimized downtime and allowed the team to maintain a steady focus on delivering value. Nevertheless, other benefits of the agile scrum methodology were not overlooked and still benefited from. For example, there still was a two-week sprint at the end of which a sprint review was presented to stakeholders talking about the achievements of the sprint and upcoming “broad goals” for the upcoming sprints. Less and less time was spent as a team on “planning” and more and more was spent on “delivering”.

Of course, there had to be some level of planning but it was kept to a minimum by restricting planning to a broad and generic goal oriented approach. Goals would be minimally viable, according to priority, deliverable and often user perceivable features such as “auth system” or “subscription system”. This was necessary in order to give the team a sense of general direction and an eventual sense of accomplishment when goals would be met.

3.1 To estimate or not to estimate

A core element of the modified approach to agile scrum was the incorporation of the no-estimates mindset. As is the case in engineering in general, estimation of work effort and/or timeline in software engineering is notoriously unreliable due to the dynamic and complex nature of the work. At the very best it is but a guess. The very invention of the “story point” system in Agile Scrum methodology was to avoid estimating time in favor of estimating complexity. This was originally intended to shift product management’s thinking from “time” to “difficulty”. However, very quickly engineering teams simply forgot the objective behind this and started using story points as indicators for time, i.e. “1 story points half a day of work, 2 story points 1 day of work, etc.”.

Of course, running a business in the real world means restraints in time and budget. Product managers simply cannot make decisions without having any idea how long a project will take. This is where projections come in. Projections in the software engineering world, if implemented correctly, are far superior to estimates. Projections help provide a more accurate sense of investment in time and money required for a certain project, eliminate time wasted “estimating tickets” and provide a sustainable work load for team members without stressing everyone out.

There are several ways of making correct projections for a software project, but all of them steer away from any guess or “estimation” work. Basically, teams first need to plan in terms of features what needs to be built. In our case, we make use of some of the principles of Agile Scrum like epics and stories. Epics are higher level goals such as “users can purchase books” while stories are lower-level sub-goals for epics like “admins can add books to catalog”, “user can browse books”, “integrate payment gateway”, etc. The storyboard, i.e. epics and stories are not fixed and allow for change along the way. We then “count” the number of tickets in our backlog and choose a very small subset to work on for let’s say a month. At the end of the month, we now have a more accurate insight into the team’s capability and capacity and product management can make more informed decisions on how to proceed on the project either by continuing as is, adding more team members or even canceling it all together.

Predicting exact timelines or effort for tasks often proves to be almost useless with extremely high probabilities of deadline overshooting or undershooting, as the inherent variability in problem-solving can skew even the most informed guesses. This has been proven by many studies over the years and could be the answer as to why 80% of all engineering projects fail to meet deadlines. The no-estimation strategy prioritizes the adoption of minimally scoped tickets and abandonment of time-based predictions in favor of metrics derived from actual delivered work. In other words, instead of attempting to estimate delivery times upfront, the team observes progress over several sprints to gather data on the average throughput (or velocity), i.e. the number of tickets completed in a given period. Using this empirical data, the team would be able to approximate delivery dates, continuously updating and tweaking forecast as the project evolves,

The idea here is that teams don’t spend accumulated hours “guessing” a magical number ahead of time so that product managers can have visibility into the future and make decisions early on. Rather, teams do what they do best and focus on building useful features and products for clients and product management can make forecasts and decisions based on immediate previous progress on the project compared to the storyboard. Nothing is perfect, neither is this methodology but in our experience this is a much better projection system. Project managers can finally get the closure and peace of mind regarding the investment they need to make for the project rather than relying on guesses and time wasted on guessing that could’ve been used for development. After all, sprint planning sessions are rather costly from the perspective of project management. Even a one hour session which is on the low end of what teams usually spend per week on sprint planning and story point estimation could mean thousands of dollars on accumulated time and cost of five to ten engineers. If we take a team of ten engineers as an example, that is ten man hours per week that could’ve been spent building software.

To ensure this approach remained effective, the team emphasized on breaking down of tasks (tickets) into the smallest possible scopes. Smaller tickets reduce uncertainty and allow for a more predictable flow of work, aligning with Kanban principles and enabling the no-estimation strategy to function effectively. While the team retained the use of complexity-based story points to capture and have visibility over the relative difficulty of tasks, these were used solely for prioritization and workload balancing rather than as a predictor of time required or consumed. By focusing on the number of tickets completed rather than arbitrary point totals, we achieved a more accurate and adaptable system for project management. This approach not only improved the team's throughput but also fostered a sense of continuous progress, enabling the successful completion of the Core Service project in a manner that minimized wasted time and maximized efficiency.

4. SYSTEM DESIGN & ARCHITECTURE

Knowing that the company was operating in a highly competitive and demanding industry, fintech, the following were the main objectives for the to-be-built macroservice, the ***"Core Service"***:

Main Objectives

- Reduce time-to-market for new SaaS projects by providing production ready core API macroservice for common services such as authentication, authorization, subscription, invoicing, etc.
- Reduce maintenance overhead by centralizing "the core non-business related code (i.e. core) into a single codebase with a single deployment pipeline enabling faster security patches and bug fixes
- Reduce as much as possible latency and speed issues caused by the introduction of the new Core Service

Secondary Objectives

- Divide teams into 2 groups, one very small single team group of engineers for the core API focusing primarily on often neglected or improperly implemented core features such as better auth security, reliable backup mechanism, etc., and another group of multiple teams with engineers focusing on business logic to deliver competitive features and apps
- The Core Service should be client technology agnostic, meaning it can be used with any programming language and/or framework that supports REST API communication

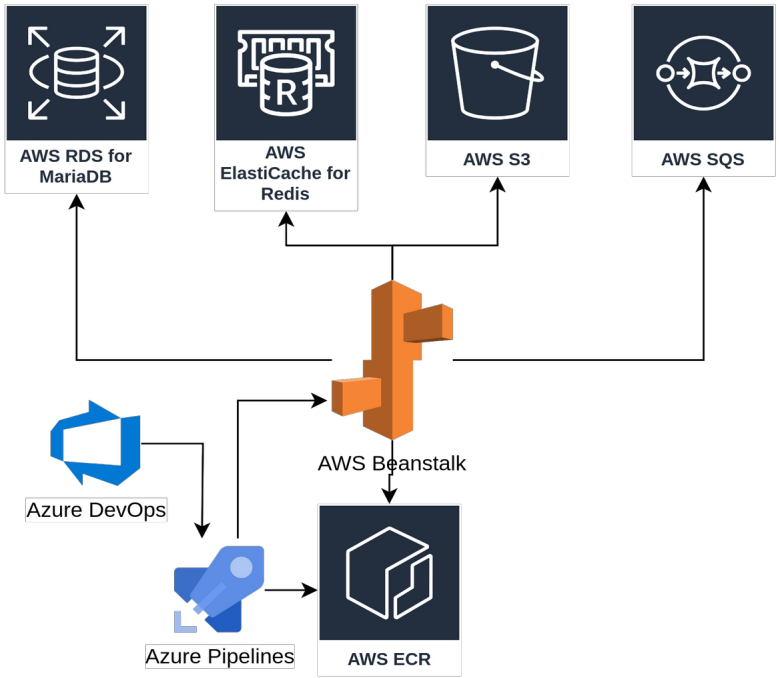


Figure 5: Cloud architecture

4.1 Components

Programming Language

The Core Service was programmed using PHP Laravel mainly due to the team being highly experienced in PHP and Laravel but also due to the fact that Laravel ecosystem allowed for rapid time-to-market and high productivity

Web Server

Instead of using traditional synchronous PHP web servers such as php-fpm over Apache or NGINX, an asynchronous setup with Laravel Octane using PHP Swoole was used. This proved to deliver performance comparable to that of natively asynchronous backend technologies such as Node.js or Go on some endpoints where concurrency is taken advantage of.

Communication Protocol

A simple HTTP REST API protocol was chosen for communication. The main reason for this was to simplify integration efforts for client applications and the simple nature of the Core Service itself did not warrant more advanced microservice protocols such as gRPC and/or message brokers. This does not mean that the communications were necessarily synchronous. Some endpoints where handled asynchronously using queues and jobs

Database

A fully managed and auto-scaled instance of MariaDB from AWS RDS was used for both landlord and tenant databases. AWS RDS also takes care of backups and snapshots out of the box

File Storage

AWS S3 was used for file storage. It is auto-scaled and rather cheap with advanced features such as authorization, temporary access tokens, retention, etc.

Cache

A fully managed and auto-scaled instance of Redis from AWS ElastiCache was used. Laravel ships with database drivers for caching which makes for a good backup system for any reasons such as cost management, unavailability, etc.

Queues/Jobs

Since the main usage for queues & jobs in the Core Service was for rather low-frequency jobs such as creating tenants & running their database migrations, backing up and restoring databases, etc., a database driver was opted for for cost saving purposes. Furthermore, using Laravel meant that switching to a more scalable and efficient solutions such as AWS SQS was simply a matter of changing configuration

variables without the need to change code. This provided cost efficiency immediately with the ability to scale in the future with minimal effort

As for worker nodes, AWS Elastic Beanstalk provides functionality to run worker instances out of the box

Containerization/Virtualization

In the interest of having a unified environment both locally and in the cloud, docker containers were the choice of OS abstraction. This allowed for fewer instance of “it works on my machine” problems. Luckily, Laravel Sail provided easy and integrated abstraction for using docker containers for local development purposes. This also made it easy to deploy the application using fully managed infrastructure such as AWS Elastic Beanstalk. AWS Elastic Container Registry was used to store container artifacts produced by the CI/CD pipelines during the build stage.

Load-balancing & Auto-scaling

Since we opted to use docker containers with AWS Elastic Beanstalk, we did not have to worry about load-balancing & auto-scaling as these were automatically handled for us.

CI/CD Pipelines

Microsoft Azure DevOps was used as the choice for CI/CD pipelines due to previous team experience and personal preferences.

Frontend

Although the Core Service was mostly a headless API system, there was nevertheless a need for a small back-office tool to manage tenants. For this, Laravel Nova was used as it provided battle tested quick tools for building admin panels. The admin panel was to only be used by the Core Service team and not client application teams.

5. METHODOLOGY

Since Laravel was used as the framework of choice for this project, its conventions and best practices were observed throughout the project. These conventions and best practices come from the years of battlefield experience of the Laravel community in general and the team involved in this project in particular.

5.1 Collaboration

Jira was used as the choice of agile project management. Jira is a proprietary product developed by Atlassian that allows bug tracking, issue tracking and agile project management. Jira is used by a large number of clients and users globally for project, time, requirements, task, bug, change, code, test, release, sprint management [4]. Jira supports various agile methodologies, including Scrum and Kanban, allowing teams to create customizable workflows, manage sprints, and visualize project progress through interactive boards. Its features include bug tracking, task management, reporting capabilities, and integration with numerous third-party applications. By providing a centralized platform for collaboration, Jira helps teams improve productivity, enhance transparency, and streamline the development process from inception to deployment.

Our Jira instance was configured to support a hybrid approach between Kanban and Agile Scrum as Jira is highly flexible and configurable to different needs. This allowed for efficient project management and progress tracking for out teams.

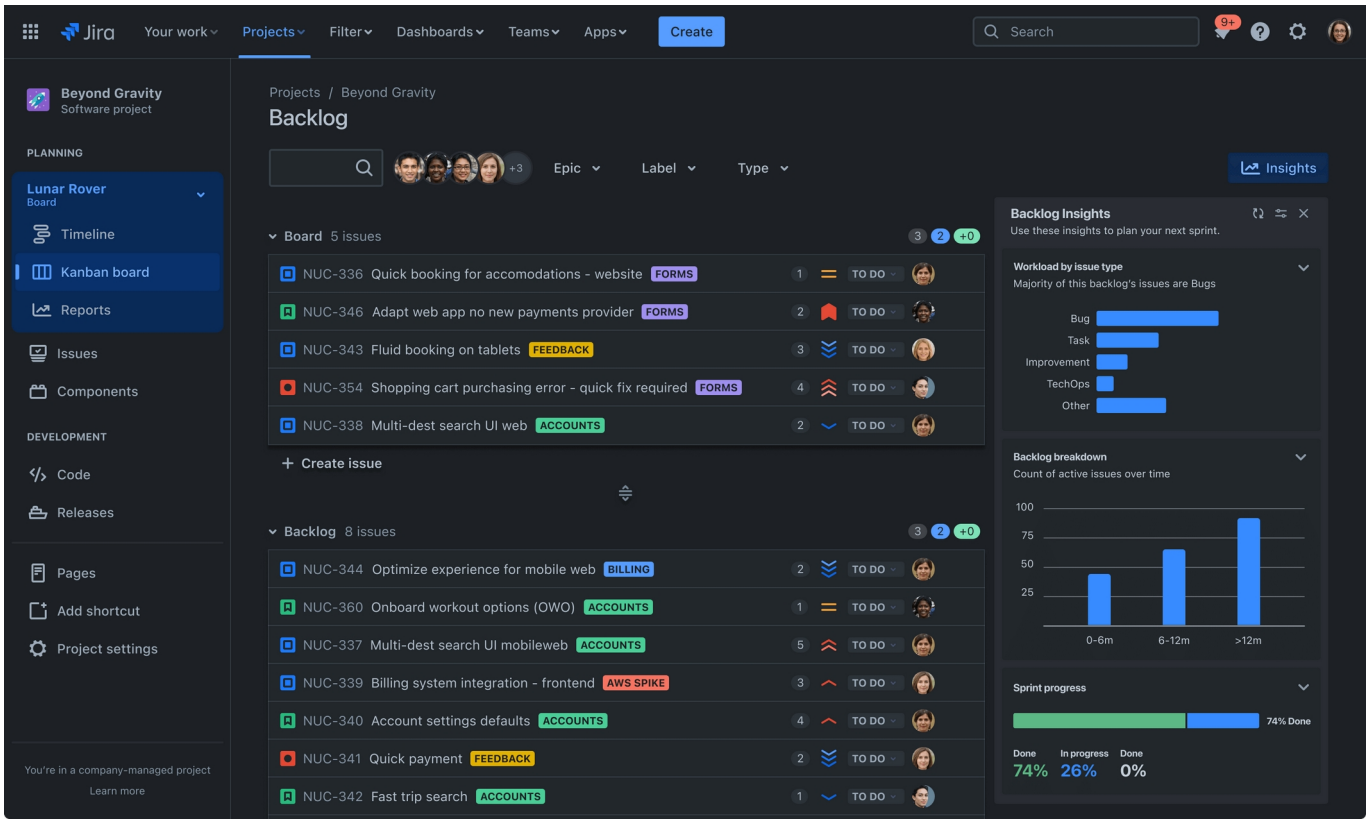


Figure 6: Atlassion Jira user interface [9]

5.2 Continuous Integration (CI)

Microsoft Azure DevOps was used as the choice of git based repository hosting and CI (continuous integration). Azure DevOps provides version control, reporting, requirements management, project management, automated builds, testing and release management capabilities [5]. It also provides a friendly GUI for the git pull-request process with capabilities such as change requests, comments, in-line comments, changes preview, and much more.

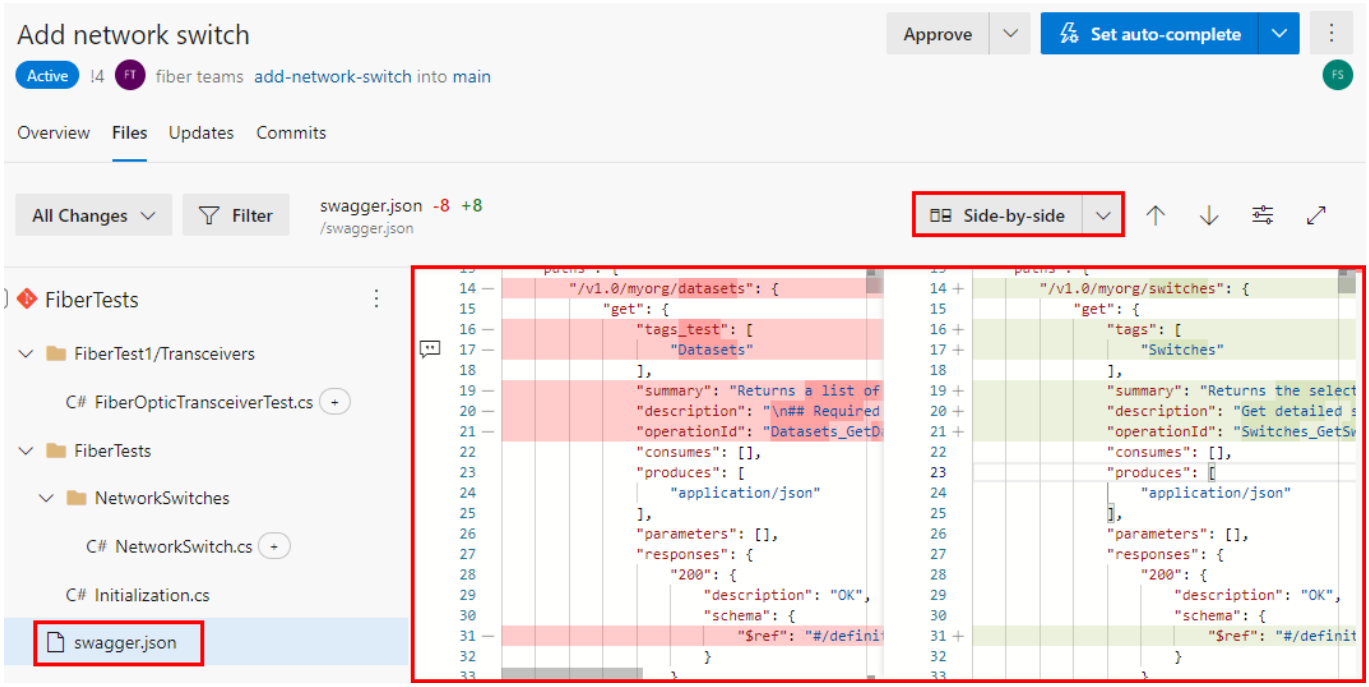


Figure 7: Azure DevOps pull-request changes overview screen [8]

5.3 Continuous Deployment (CD)

Microsoft Azure Pipelines which sits on top of Azure DevOps was used as the choice of CD (continuous deployment) automation tool. Three environments were created for the Core Service to provide robust testing and sandbox environments. These were development, staging and production environments. The CD pipeline had 6 different stages as shown in Figure 8:

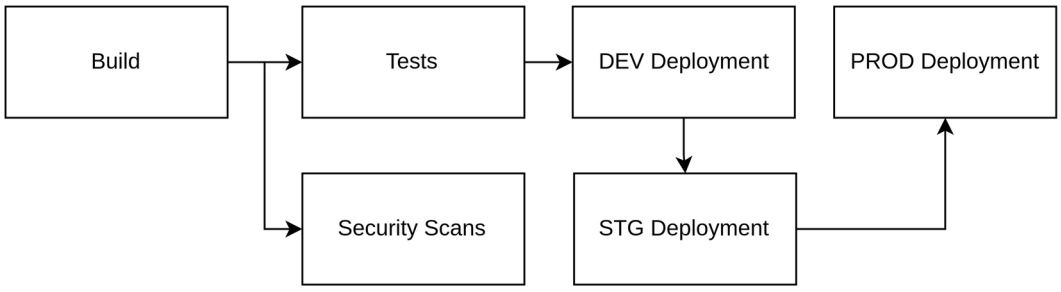


Figure 8: CD pipeline stages

1. Build: In this stage, artifacts such as docker container images are built

2. Tests: In this stage, static analysis, unit, feature, integration and end-to-end tests are run
3. Security Scans: In this stage, the artifacts and code is scanned for security vulnerabilities using tools such as Mend. Failure at this stage does not impede deployment
4. DEV Deployment: In this stage, the new version is deployed to development environment which is meant for internal team testing
5. STG Deployment: In this stage, the new version is deployed to the staging environment where testing is done by members and stakeholders outside of the development team. This stage requires special user permissions to run
6. PROD Deployment: In this stage, the new version is deployed to the production environment and requires elevated user permissions to run

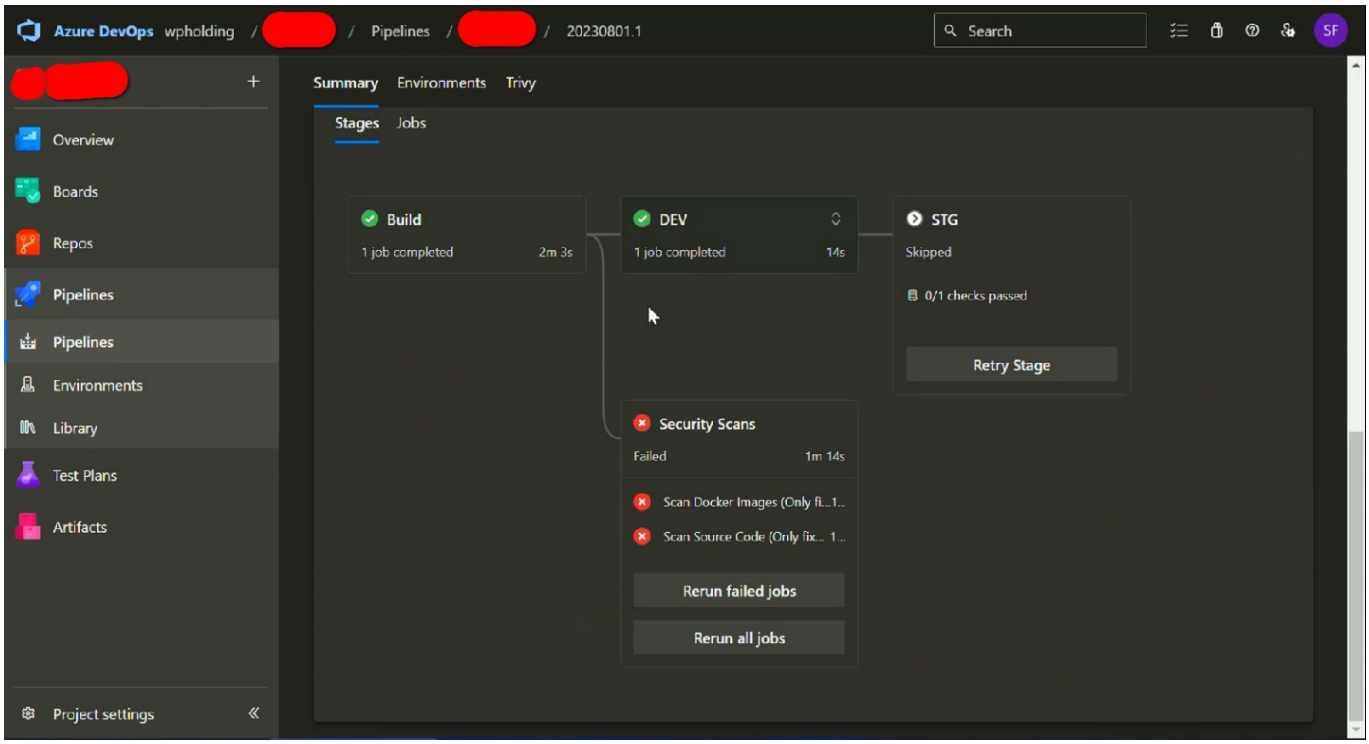


Figure 9: Azure DevOps GUI showing the different CI/CD pipeline stages

5.4 MVC

Model–view–controller (MVC) is a software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements [6]. These elements are:

- the **model**, the internal representations of data
- the **view**, the interface that presents information to and accepts it from the user
- the **controller**, the software linking the two

Since the Core Service is an API only application, the **view** interface would be in JSON rather than a graphical user interface. Using an MVC design pattern ensures a more organized approach to the source code and adds a layer of abstraction and separation of concern between data storage and business logic.

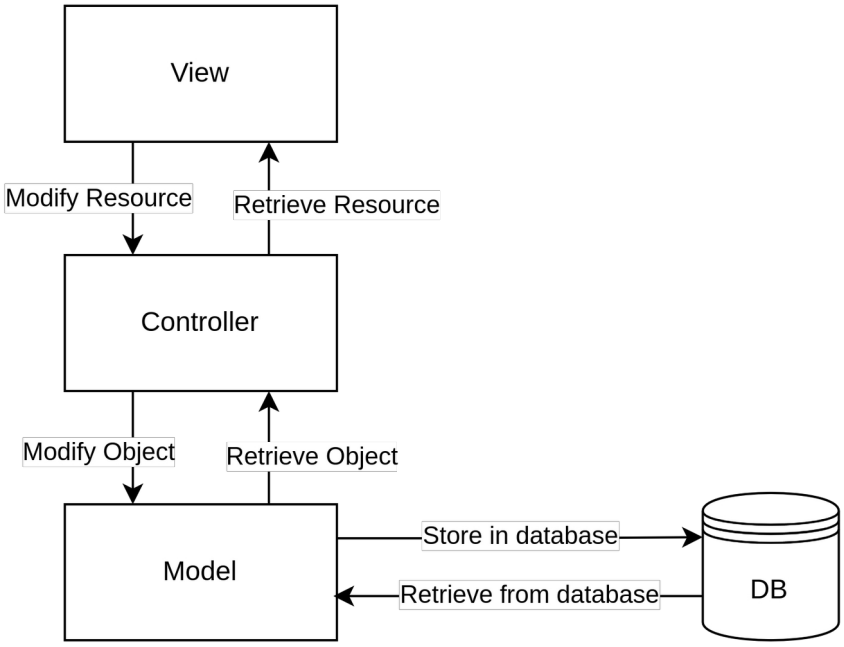


Figure 10: MVC Pattern Visualization

5.5 Async Request Processing

Over the years, one of the main disadvantages of using PHP despite its many advantages was its synchronous approach to request processing. Although solutions such as multi-threaded FastCGI workers attempted to resolve this by using multiple parallel instances and request queues, it still never the less processed each request sequentially in a synchronous manner. This meant that the time spent waiting in idle (doing nothing) for external processes such as DB requests was wasted.

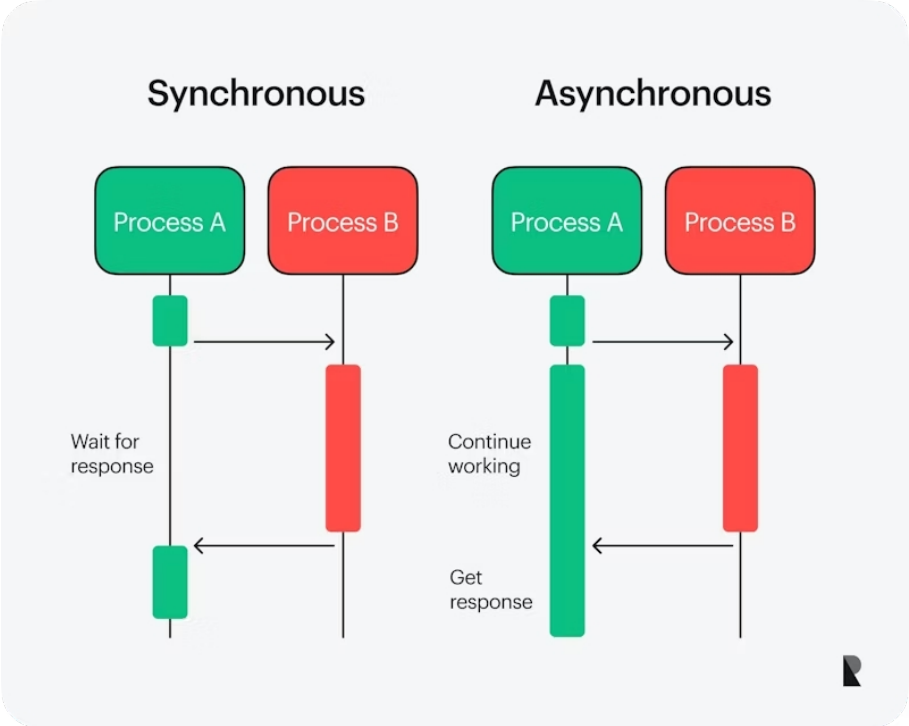


Figure 11: Synchronous vs asynchronous request processing [10]

This as supposed to asynchronous request processing such as the case of Javascript Node.js which uses an event loop to make use of otherwise would-be-wasted wait time as shown in Figure 12

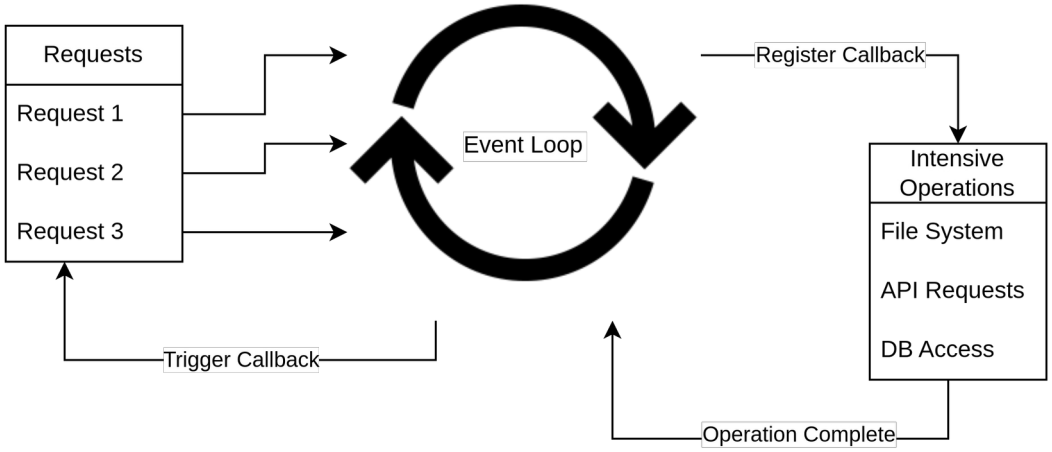


Figure 12: Node.js Event Loop

Another problem was the fact that traditional PHP loads and bootstraps the entire application from scratch for each request. This adds a great amount of repeated work at the beginning of each request. Luckily, recent innovations in the PHP world enabled resolving both problems in rather efficient ways. These include concurrency based application servers such as Swoole, OpenSwoole, FrankenPHP, etc. Laravel Octane which was the web/application server of choice for the Core Server is a native wrapper around

these libraries and brings asynchronous request process to Laravel applications with minimal configuration [7].

However, there are some caveats to using asynchronous application servers with PHP. Since Octane bootstraps the application once in the beginning and reuses it for subsequent requests (albeit by making a copy in memory for every request and not a total reuse), adjustments need to be made when developing applications to ensure no memory leak occurs. For example, static variables would persist between requests and any application service container injections during the boot process should be done using proper resolvers and closures [7]. Furthermore, at the time of writing this report, only Swoole supported running concurrent tasks.

5.6 Multi-tenancy in Laravel

Although there are existing packages to achieve multi-tenancy in Laravel as discussed before, a customized solution was implemented for the purposes of the Core Service. Essentially, there would be two instances of database, log, cache, queue and filesystem connections, Landlord and Tenant. A landlord database is used by the Core Service to store data related to its own internal tenant management processes such as Tenant settings. This is easily done in Laravel using the config files for each item. For example, for the database config file, we assign two database connections as shown in the code below:

```
'connections' => [
    'tenant' => [
        'driver' => 'mysql',
        'url' => env('TENANT_DATABASE_URL'),
        'host' => env('TENANT_DB_HOST', '127.0.0.1'),
        'port' => env('TENANT_DB_PORT', '3306'),
        'database' => null, // tenant specific. Set dynamically by app/Models/Auth/Tenant::use()
        ...
    ],
    'landlord' => [
        'driver' => 'mysql',
        'url' => env('LANDLORD_DATABASE_URL'),
        'host' => env('LANDLORD_DB_HOST', '127.0.0.1'),
        'port' => env('LANDLORD_DB_PORT', '3306'),
        'database' => env('LANDLORD_DB_DATABASE', 'forge'),
        ...
    ],
],
```

And then for each Eloquent Model we make sure to specify which connection it needs to use:

```
class Tenant extends Model
{
    protected $connection = 'landlord';
    ...
}
```

This essentially makes Eloquent Models aware of which connection to use when communicating with the database. A similar configuration is done to the cache system, file system, log system, etc. Of course, there are separate migration files for each database connection:

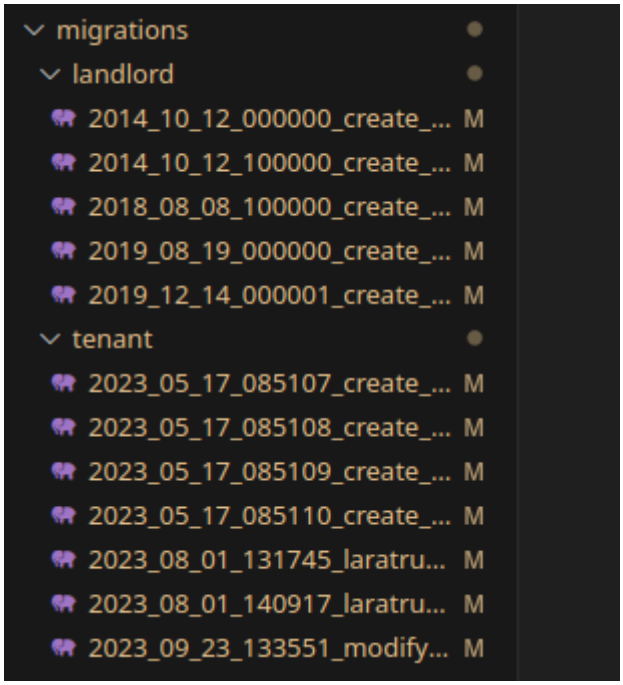


Figure 13: Migrations folder containing separate landlord and tenant migration files

This breaks Laravel Artisan's "php artisan migrate" command, therefore we need to specify the connection and migrations folder "php artisan migrate –database=landlord –path=database/migrations/landlord"

Factories and seeders also need separate files and/or folders for each connection:

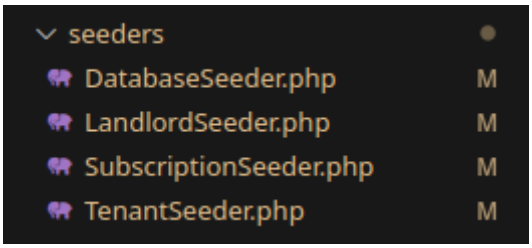


Figure 14: Seeders folder

We can then specify the connection just like we did with the migrate command "php artisan db:seed –database=landlord"

Endpoints that need to know which sub-tenant is being served need to have a prefix route param for this. This param will later be used by the middleware responsible for booting the tenant. E.g.:

```
Route::prefix('/tenants/{tenant_uuid}')->group(function () {
    Route::get('/list-all', [App\Http\Controllers\NotificationController::class, 'index']);
    Route::post('/send-email', [App\Http\Controllers\NotificationController::class, 'sendEmail']);
});
```

5.7 Making Core Service Tenant Aware

Since Laravel by default is not multi-tenant, some modifications were needed to make it tenant aware. Being tenant aware is essentially telling Laravel which tenant at the moment it is processing the request for, what database, cache, queue, etc. to use (since each tenant has a isolated databases, cache, filesystems, etc.). This is done early on in the request life cycle by injecting a middleware called “BootTenant”. This middleware essentially detects based on certain conventions the following items:

- Which Tenant (i.e. application) is making the request? This is determined using a request header “X-APPLICATION”
- Which Sub-Tenant (i.e. tenant’s tenant) is the request for? This is determined using route params “/v1/auth/tenants/08237b02-1528-410e-8a50-155ba1d80dc6/login”
- Which User is carrying out the request? This is determined using an authentication token as a request header

According to these, it then proceeds to configure the app instance of Laravel to use the tenant specific configuration such as database credentials, cache credentials, file-system, queue system, application branding, email servers, etc. This is really where we achieve multi-tenancy and where the magic happens. This setup provides complete data isolation between tenants (i.e. client applications). This middleware needs to be configured in the kernel configuration file so that Laravel uses it:

```
/**
* The application's route middleware groups.
*
* @var array<string, array<int, class-string|string>>
*/
protected $middlewareGroups = [
    'web' => [
        ...
        \App\Http\Middleware\BootTenant::class,
        ...
    ],
    'api' => [
        ...
        \App\Http\Middleware\BootTenant::class,
        ...
    ],
];
```

Figure 15 demonstrates a simplified architecture of an example client app using the core services:

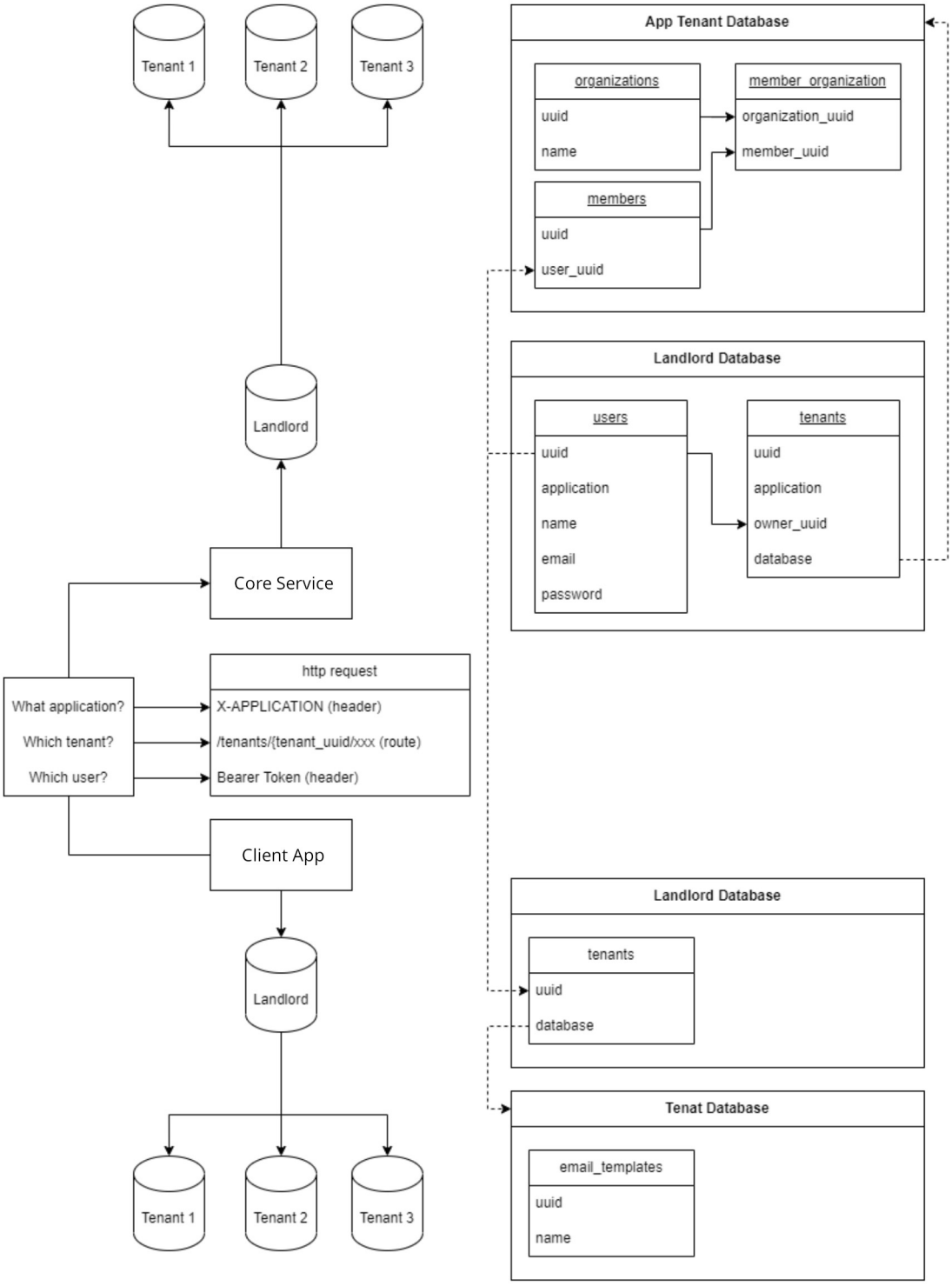


Figure 15: Client app & Core Service architecture

5.8 Auth Service

The Auth service serves 3 essential services, authentication, authorization and user activity. The main list of functionalities provided by the Auth service is shown in Table 1:

Group	Feature	Description
Authentication	Token based login	Login using JWT tokens (mobile apps)
	Cookie based login	Login using cookies (SPA apps)
	Register	Create new user accounts
	Password reset	Reset and/or recover account password
	Email verification	Verify and/or change account email address
Authorization	Permissions	Manage permissions
	Roles	Manage roles and assign permissions to them
	User Management	Manage users, manage access control and determine the roles and permissions of users. It also enables client applications to determine whether users have permissions to carry out certain actions
	OTP	Use OTPs to verify user actions
User Activity	Operations Log	Log user activity and maintain an operation history/log for auditing purposes

Table 1: Auth service features

5.9 Subscription Service

The Subscription service allows client applications to create products, manage coupons and discounts, create packages, process payments, manage periodic recurring subscriptions, etc. The main list of functionalities provided by the Subscription service is shown in Table 2:

Group	Feature	Description
Catalog	Products	Create and manage different products for sale
	Packages	Create and manage packages (groups) of products for sale
Subscriptions	Plans	Create and manage subscription plans
	Subscriptions	Create and manage recurring subscriptions with automated payment processing features
Payment	Vouchers	Create discount and voucher codes
	Taxes	Easily calculate taxes for payment processes
	Online Payments	Process bank card payments easily and securely
Statistics	Statistics	Get insights & statistics on products, packages & subscriptions

Table 2: Subscription service features

5.10 Notification Service

The Notification service allows client applications to asynchronously send omni-channel notifications via various channels such as email, SMS, Whatsapp, database, etc.

Channel	Description
Email	Send and manage emails to single or multiple addresses
SMS	Send SMS to single or multiple phone numbers ensuring minimal cost by automating provider selection
Whatsapp	Send Whatsapp notifications
Database	Manage “database” or “application” notifications for users
Push (websockets)	Subscribe/Publish to websocket channels with authentication and authorization for real time notifications

Table 3: Notification service features

5.11 Database Service

The Database service allows client applications to manage periodic database backups, one-click restores, database exports and more. It internally uses AWS RDS API and MySQL commands to create and manage snapshots

Feature	Description
Backup	Create and manage automatic backups (snapshots) for databases
Export	Export databases to CSV or SQL files

Table 4: Database service features

5.12 Using Core Service

Out of the already existing four SaaS applications in the company, two were migrated to the newly built Core Service. As expected, migrating an existing project to use the Core Service proved to be a tad more complicated than building a new SaaS project from scratch with Core Service in mind. Nevertheless, our teams were able to migrate to the Core Service within a one month from starting the process.

As for new applications built using Laravel, the integration of the Core Service was straightforward. First, we needed to create a new PHP class that resembles and imitates a Laravel Eloquent Model. Luckily, since Laravel Eloquent Models make heavy use of traits and inheritance, we were also able to benefit from those. We first start by creating an abstract class (or a contract) called RemoteModel that implements some Laravel contracts such as Arrayable, Jsonable, JsonSerializable, etc. It makes use of PHP magic methods to provide the same sugar syntax Laravel Eloquent Models provide.

```
abstract class RemoteModel implements Arrayable, Jsonable, JsonSerializable
{
   protected $attributes;
   protected $visible = [];
   protected $hidden = [];

   public function __construct(array $attributes = [])
   {
      $this->attributes = $attributes;
   }

   public function __get(string $key): mixed
   {
      return $this->getAttribute($key);
   }

   public function __set(string $key, mixed $value): void
   {
      $this->setAttribute($key, $value);
   }

   ...
}
```

Then, we need to create a service class to abstract the process of sending HTTP requests to our CoreService APIs. We will call this class CoreService, just like the name of our macroservice. This class will handle application authentication (as the CoreService needs to "authenticate the tenant", i.e. our application), passes other required information such as the current sub-tenant, the user, etc. It also handles any errors thrown by the Core Service.

```
class CoreService
{
    ...

    public function sendRequest($method, $path, $data = [], string | null $use = null)
    {
        $this->client = \Http::acceptJson();

        $headers = [
            'X-APPLICATION' => config('services.core_service.application_identifier'),
            'X-APPLICATION-TOKEN' => config('services.core_service.application_token'),
            'ACCEPT-LANGUAGE' => App::currentLocale(),
        ];

        if ($this->tenantUuid) {
            $headers['X-TENANT-UUID'] = $this->tenantUuid;
        }

        if ($use === 'token') {
            $this->restoreBearerTokenFromSession();

            $headers['Authorization'] = "Bearer {$this->bearerToken}";
        } else if ($use === 'application') {
            $headers['X-ON-BEHALF-OF'] = 'application';
        }

        $this->client->withHeaders($headers);
        $response = $this->client->$method("{$this->endpoint}{$path}", $data)
            ->throwIf(fn (Response $response) => in_array($response->status(), [407, 419]) ||
            ($response->status() >= 500 && $response->status() <= 599));

        return $response;
    }

    public function post($path, $data = [])
    {
        return $this->sendRequest('post', $path, $data, 'token');
    }

    ...
}
```

We are now ready to inherit the RemoteModel class to create our custom models that use the CoreService as data store. Essentially, we wrote a “data persistence driver” for our Model. Instead of persisting the models to the database ourselves, we delegate this process to the Core Service via HTTP requests by using the CoreService class. These classes will have the typical Laravel Eloquent Model methods such as

Model::find(), Model::get(), Model::delete(), etc. We create a User class that extends the RemoteModel and implements the Laravel Authenticatable contract so that it can be used for authentication within Laravel.

```
class User extends RemoteModel implements Authenticatable
{
    ...
    private $coreService;

    public function __construct(array $attributes = [])
    {
        parent::__construct($attributes);

        $this->coreService = new CoreService();
    }

    public static function find(string $uuid, bool $asApplication = false): User|null
    {
        if (empty($uuid)) {
            return null;
        }

        $user = new User();

        $method = $asApplication ? 'getAsApplication' : 'get';

        $response = $user->coreService->$method("/auth/v1/users/{$uuid}", [], false)->throw();

        $user->attributes = $response->json()['data'];

        return $user;
    }

    ...
}
```

We then create a user provider called CoreServiceUserProvider so that Laravel can use it as an auth user provider for authentication purposes. This class implements the Laravel UserProvider contract and provides the functionality required by the framework to perform authentication procedures.

```
class CoreServiceUserProvider implements UserProvider
{
  ...

  public function retrieveById($identifier): Authenticatable|null
  {
    return User::find($identifier, true);
  }

  ...
}
```

Finally, we need to register the CoreServiceUserProvider in the boot function of the framework's auth service provider

```
class AuthServiceProvider extends ServiceProvider
{
  ...

  /**
  * Register any authentication / authorization services.
  */
  public function boot(): void
  {
    Auth::provider('core_service', function () {
      return new CoreServicecUserProvider();
    });
  }

  ...
}
```

Our client app now has a fully functional integration of the Core Service and can authenticate its users via the Core Service APIs.

5.13 Making it multi-level multi-tenancy

Now that we have implemented the Core Service which is a multi-tenant architecture that expects to work with multi-tenant client applications, all we needed to do is implement multi-tenancy in our client applications. This is very similar to what we did for the Core Service except it is a single layer multi-tenancy, i.e. a tenant for each customer. For the majority of our use case and clients, we would market multi-tenancy as "private database", meaning each customer gets their own isolated database. This gave our clients assurance regarding their data privacy and security. Together with the Core Service, we have now achieved a "nested multi-level multi-tenancy" architecture. We could've also taken advantage of existing multi-tenancy libraries we discussed earlier as the way multi-tenancy is achieved in the client applications is of no relevance to the Core Service.

6. RESULTS

Although it is rather difficult to objectively measure results related to productivity increase among SaaS projects after using the Core Service, we have attempted to collect at the very least least more particular measurable and perceived statistics.

6.1 Maintenance Efforts

We have observed a tremendous reduction in maintenance efforts related to the services provided by the Core Service in the sense that the effort is not duplicated. Yes, we still had to fix bugs and security vulnerabilities but now we only do them once in a centralized place. We noticed about 60% to 80% percent reduction in duplicate Jira tickets related to authentication, authorization, subscription and notifications. There was some inevitable duplication related to the implementation and integration of the Core Service in the client applications themselves however.

Overall, in our sprint reviews we saw a steady and continuous increase in work completed related to business logic and features compared to our previous sprint reviews before the introduction of the Core Service.

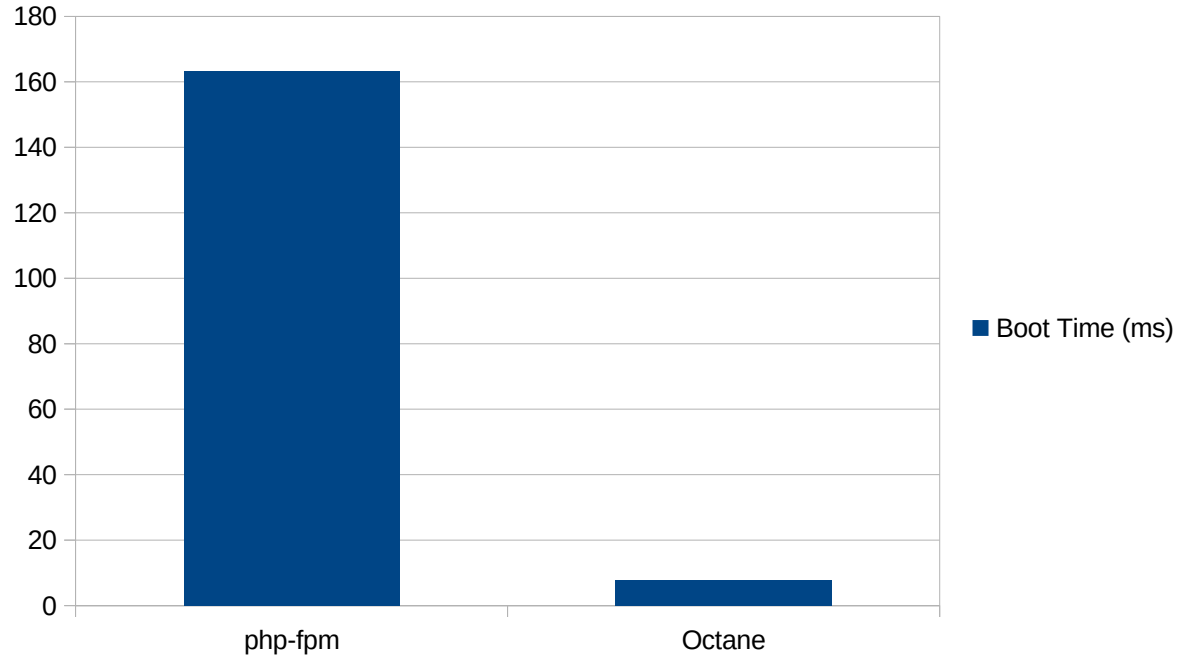
6.2 Time-to-market

It is quite difficult to obtain a fair and objective comparison with regards to time-to-market in SaaS projects built with and without the Core Service. This is due to the big difference in scope of work for each SaaS project, team velocity and capacity, external dependencies, etc. However, we were able to perceive a self reported (on average, by our teams) 80% reduction in time spent developing the initial common modules such as authentication, authorization, subscription, etc., which meant more time is spent developing the actual SaaS application business logic and user features. We also perceived faster and quicker bug and security fixes related to the common functionalities provided by the Core Serivce due to the absence of needing to update and deploy multiple codebases repetitively.

Multiple SaaS applications were built using the Core Service. Compared to previous projects, there was a significant reduction in time-to-achieving the first iteration of the MVP. Our teams were able to develop the domain features at a much faster rate than before. Our projections related to time required were also significantly more accurate with the newly adopted no-estimates methodology. On average, we achieved an error threshold of about 13% compared to 67% before.

6.3 Sync PHP vs Async PHP

Using Laravel Octane proved to provide substantial performance gains due to the not needing to bootstrap the application & framework from scratch with every new request. First, we observed the average time it took to boot the application & framework for each request using Laravel Debugbar. As demonstrated in Graph 1 we see a significant amount of reduction in time consumed booting the application & framework when using Laravel Octane as compared to php-fpm. On average, the boot time dropped from 163.12 ms to 7.76 ms, an impressive 95.24% reduction:



Graph 1: php-fpm vs Laravel Octane boot time

This is a very significant gain in performance for each request. However, in the real world the bottleneck for most application is not the boot time but rather inefficiencies related external factors such as database queries and network latencies. These bottlenecks can be greatly reduced by taking advantage of CPU idle time. This however does require changing our code to make use of concurrency feature provided by Laravel Octane.

To benchmark this, we load tested one of our endpoints that carries 2 database queries and an external API call for every request using Grafana k6. Using 4000 virtual users, we tested how many successful requests we can make to the endpoint within 30s using:

- php-fpm with 40 static workers
- Laravel Octane with Swoole with 40 workers and 60 task workers

All other parameters such as CPU resources, RAM resources, environment configurations, etc. were kept equal for both cases. We first performed the load test on the endpoint before making any use of concurrency:

```php
Route::get('/results', function () {
    $settings = Settings::where('tenant_uuid', app('tenant')->uuid)->findOrFail();

    $users = User::inRandomOrder(50)->get();

    $results = Http::acceptJson()->get($settings->archivedEndpoint)->json();

    return view('results', [
        'users' => $users,
        'results' => $results,
    ]);
});
```

And another test by making use of concurrency:

```php
Route::get('/results', function () {
    $settings = Settings::where('tenant_uuid', app('tenant')->uuid)->findOrFail();

    [$users, $results] = Octane::concurrently([
        fn() => User::inRandomOrder(50)->get(),
        fn() => Http::acceptJson()->get($settings->archivedEndpoint)->json(),
    ]);

    return view('results', [
        'users' => $users,
        'results' => $results,
    ]);
});
```

And a k6 script of:

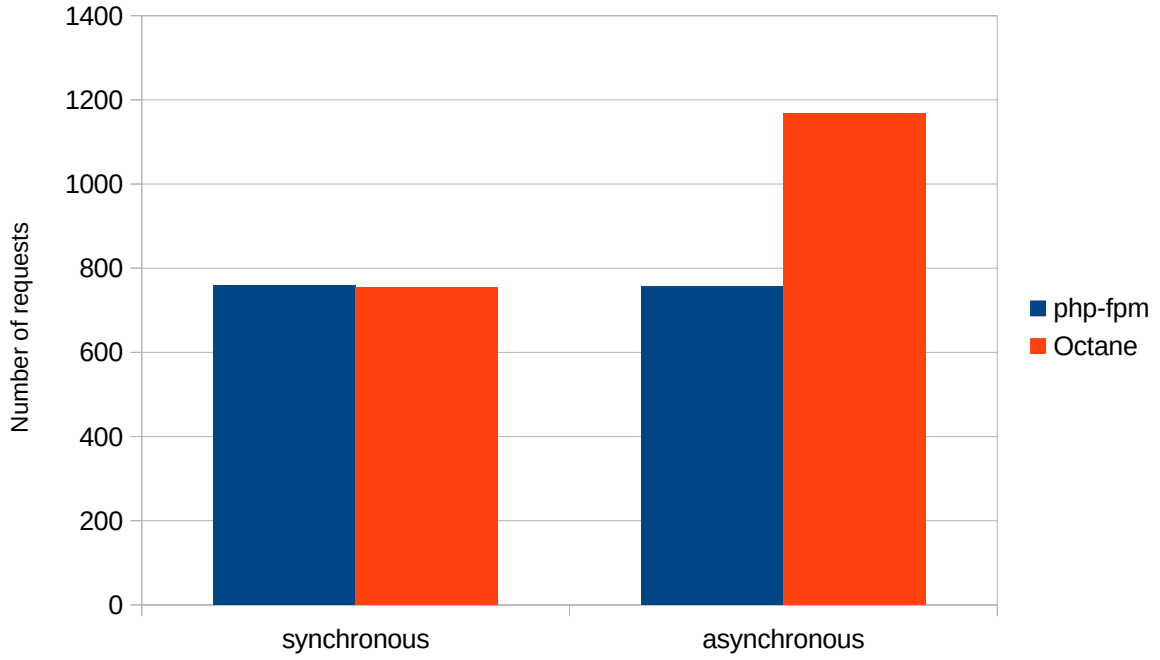
```javascript
import http from 'k6/http';
import { sleep } from 'k6';

export const options = {
    vus: 4000,
    duration: '30s',
}

export default function() {
    http.get('http://localhost:8000/results');
    sleep(1);
}
```

We can observe that when we don't use concurrency, both php-fpm and Laravel Octane perform rather similarly as the 3 intensive operations take an average of 1 second each, rendering the ~150ms boost gained by only bootstrapping the application & framework once rather insignificant. However, when we

run some of these intensive operations concurrently, we see a significant boost of 54% in the number of requests that can be handled as demonstrated in results obtained by the load test shown in Graph 2:



Graph 2: php-fpm vs Laravel Octane load test using concurrency

Needless to say the performance difference between php-fpm vs Laravel Octane is highly dependent on the code and how much of it can be executed asynchronously.

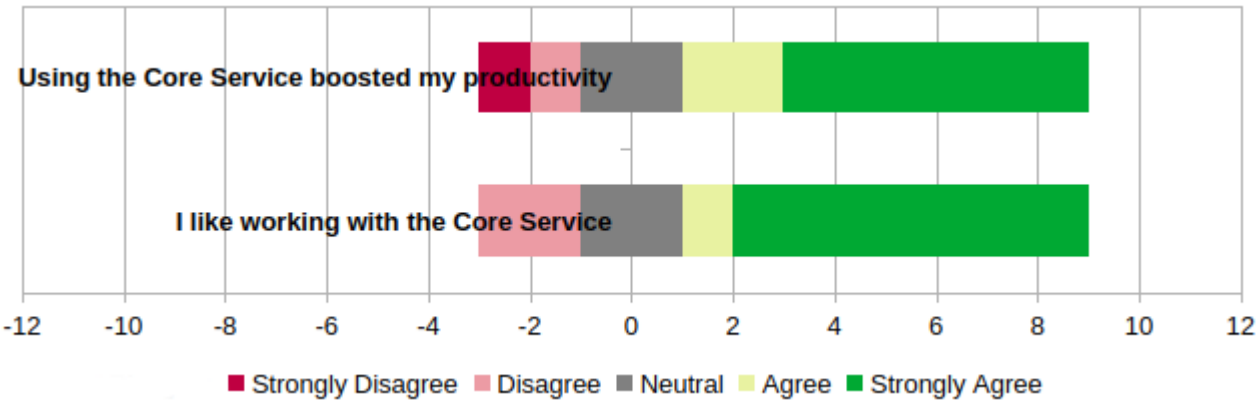
6.4 Observability & Monitoring

Building a multi-level nested multi-tenant core macroservice that served multiple different SaaS applications introduced elevated challenges in observability and monitoring due to the aggregation of logs and metrics. The primary issue stemmed from the sheer volume of logs being generated, which made it difficult to isolate and analyze logs specific to a particular application or sub-tenant during debugging or incident resolution. This forced us to implement robust log tagging and indexing mechanisms to differentiate logs across tenants while ensuring scalability and performance using tools such as Kibana and AWS Elasticsearch. Additionally, the complexity of log aggregation increased as shared resources across tenants often generated interleaved logs, complicating root cause analysis. For example, we had to make some customizations and modifications to our Grafana dashboards to properly show metrics according to each tenant (application). Overall, this increase in complexity was not very difficult to handle and compared to enormous gains in productivity and efficiency, it was well worth the effort.

6.5 DX

Our Software Engineering teams reported high subjective satisfaction levels regarding their experience building new SaaS applications using the Core Service as they could avoid repetitive and routine work, spending more time on the domain of the SaaS application and delivering more features to users.

A random group of 12 engineers of ages 23-33 and experience years ranging from 1-15 years in PHP working at our company were surveyed after working on a real world SaaS project using the newly built Core Service regarding their subjective DX



Graph 3: Core Service subjective survey in number of subjects using a 5-point Likert scale

7. CONCLUSIONS AND FUTURE WORK

In the internship project, we discussed the limitations of synchronous and asynchronous request processing in web applications and demonstrated the performance boost gained by using async PHP. We also discussed the advantages and disadvantages of a multi-tenant architecture and how nested multi-level multi-tenant architecture is built on top of it. We then discussed and demonstrated the applications of nested multi-level multi-tenant architectures by building a Core macroservice that provides common functionalities such as authentication, authorization, subscription, data backups, etc. for multiple client SaaS applications and their own sub-tenants. We also discussed the advantages of using a no-estimates mindset and a hybrid approach between Kanban and Agile Scrum. We then ran some experiments and benchmarks to evaluate the gain the performance achieved by using asynchronous request processing with Laravel Octane that concluded with very promising performance boosts. We also discussed some of the challenges and complications that arose related to observability and monitoring when using such an architectures. We also discussed the difficulty of measuring results when it comes to time-to-market for SaaS projects using the Core Service as each project differs substantially from the other. Finally we surveyed the engineers in order to determine the UX change with using the Core Service we built and whether they perceive a productivity gain in their work as a result of it.

- Using an isolated setup with multi-tenant architectures helps provide data isolation for different customers which in turn helps with enhanced data privacy & security and legal compliance
- Nested multi-level multi-tenancy is an architecture that is based on and improves upon existing multi-tenancy architectures
- Multi-tenant architectures enable maximum benefit and usage of resources by sharing them among clients natively which in turn reduces operating costs for each tenant
- Multi-tenant architecture is not a new concept per se and has existed since the early days of computing when companies shared mainframe computers
- Because a multi-tenant architecture runs on the same codebase, fixing bugs and security vulnerabilities becomes much faster and more efficient
- Multi-tenant architectures can reduce maintenance overhead and long term operating costs
- A core service using a nested multi-level multi-tenant architecture greatly enhances DX and reduces time-to-market for client SaaS projects
- Many engineering teams rewrite the same core functionalities over and over again with every new SaaS project unnecessarily
- Compared to Node.js, traditional PHP process requests sequentially in a synchronous fashion wasting otherwise reusable time awaiting intensive external operations such as API requests and database reads/writes
- Libraries such as Swoole, OpenSwoole, FrankenPHP, etc. bring asynchronous request processing capabilities to PHP projects and yield comparable performance results compared to natively asynchronous technologies such as Node.js and Go

- Laravel Octane is a first-party native async PHP wrapper for Laravel that internally supports multiple drivers such as Swoole, OpenSwoole, RoadRunner, FrankenPHP, etc.
- Laravel Octane brings concurrency and asynchronous request processing capabilities to Laravel projects which greatly boosts performance and maximises resource usage
- Slight adjustments need to be made when working with Laravel Octane as both Laravel & PHP were not built with concurrency in mind
- In synchronous PHP, the application and framework is entirely loaded and bootstrapped from scratch with every request resulting in decreased performance
- In asynchronous PHP, the application and framework is entirely loaded and bootstrapped only once at the start of the server out of which a memory copy is made for subsequent requests
- In asynchronous PHP, static variables can cause memory leaks if not handled properly and are shared between requests
- In Laravel Octane, container injection should not be done on registration but on boot methods to avoid sharing them between requests resulting in unwanted behaviour
- When using traditional synchronous PHP, a webserver such as php-fpm with Apache or NGINX is required to be able to process requests
- When using asynchronous PHP with Swoole, OpenSwoole, FrankenPHP, etc. there is no need for web servers such as Apache and NGINX to handle requests
- In Laravel Octane, the worker needs to be restarted every time there are code changes and/or if the application needs to be bootstrapped over from scratch
- AWS Beanstalk greatly reduces complexity of deployment to cloud by internally handling load-balancing and auto-scaling and can be configured to scale with different requirements and strategies. It also provides abstraction from server maintenance and handles it internally applying security and bug fixes to the server
- AWS Beanstalk provides an easy way to run queue/job worker nodes
- Using docker containers provides OS abstraction so that all servers have the exact same OS setup and configuration in a reusable and rebuild-able way
- Using the same docker containers in local developer environments and on production servers reduces “works on my machine” problems and attempts to solve environment/OS related problems early on before the code is deployed to servers on cloud
- Middlewares in Laravel can be used to “boot tenants” and configure the environment so a multi-tenant Laravel app has proper access to tenant specific database
- Laravel supports multiple database, cache, log, queue, etc. configurations that could be extended to provide isolated access to landlord and tenant specific resources
- Laravel Eloquent Models support configuration that can specify what database connection to use

- Laravel is highly flexible and can be modified to meet different needs and patterns
- The Core Service built within this internship project proved to be very effective at reducing time-to-market for SaaS projects and increase engineer productivity
- Agile Scrum methodologies can be modified to meet team and project requirements in order to enhance productivity and reduce inefficiency
- A hybrid approach between Kanban and Agile Scrum can reduce time wasted planning sprints by providing more independence to team members
- No-estimation methodology aims at resolving engineering estimation error by not relying on guesses but rather factual recent data and small scoped tasks

BIBLIOGRAPHY

[1] “Multitenancy,” *Wikipedia*. Sep. 24, 2024. Accessed: Nov. 19, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Multitenancy&oldid=1247432876
[2] V. Cheng, “Multi-Tenancy Architecture - SaasCEO.com.” Accessed: Nov. 19, 2024. [Online]. Available: https://www.saasceo.com/multi-tenancy-architecture/
[3] “Automatic & flexible multi-tenancy package for Laravel.,” Tenancy for Laravel. Accessed: Nov. 19, 2024. [Online]. Available: https://tenancyforlaravel.com/
[4] “Jira (software),” *Wikipedia*. Oct. 25, 2024. Accessed: Nov. 19, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Jira_(software)&oldid=1253333074
[5] “Azure DevOps Server,” *Wikipedia*. Sep. 26, 2024. Accessed: Nov. 19, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Azure_DevOps_Server&oldid=1247922481
[6] “Model–view–controller,” *Wikipedia*. Sep. 10, 2024. Accessed: Nov. 19, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93controller&oldid=1244967192
[7] “Laravel - The PHP Framework For Web Artisans.” Accessed: Nov. 19, 2024. [Online]. Available: https://laravel.com/
[8] vijayma, “Review and comment on pull requests - Azure Repos.” Accessed: Nov. 19, 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/devops/repos/git/review-pull-requests?view=azure-devops
[9] “Jira | Issue & Project Tracking Software | Atlassian.” Accessed: Nov. 19, 2024. [Online]. Available: https://www.atlassian.com/software/jira?referer=jira.com
[10] “Synchronous vs Asynchronous Programming: Models, Differences, Use Cases | Ramotion Agency,” Web Design, UI/UX, Branding, and App Development Blog. Accessed: Nov. 19, 2024. [Online]. Available: https://www.ramotion.com/blog/synchronous-vs-asynchronous-programming/