Piranha CMS Multi-Tenant Expansion Documentation and Design

Diogo Silva
AS
Student 107647, DETI
Universidade de Aveiro, Portugal
diogo.manuel@ua.pt

Miguel Cruzeiro
AS
Student 107660, DETI
Universidade de Aveiro, Portugal
miguelcruzeiro@ua.pt

Miguel Figueiredo
AS
Student 108287, DETI
Universidade de Aveiro, Portugal
miguel.belchior@ua.pt

Guilherme Amorim
AS
Student 107162, DETI
Universidade de Aveiro, Portugal
guilhermeamorim@ua.pt

June 8, 2025

1 Introduction

The report from the first part of this assignment provided an in-depth analysis and design strategy for extending Piranha CMS to support a multi-tenancy SaaS model. It outlined an analysis of the current Piranha CMS system and its limitations and the vision and strategic goals of the scenario. Domain Driven Design was utilised to model the Business Domain into core, generic and supporting subdomains. An Attribute Driven Design architectural methodology was employed throughout three different iterations to progressively refine the system architecture according to three different sets of quality attributes: Separaton of Concerns, Isolation and Performance & Scalability. Additionally, the report addressed key cross-cutting concerns such as security, observability, error handling and fault isolation, which are essential to ensuring the robustness and maintainability of a multi-tenant system. Finally, the report also emphasized a prioritization of key features essential to the envisioned multi-tenant system and outlined a clear development roadmap to guide its implementation for the next month.

Before dwelving into the contents of this report, it's also important to remember the previously defined concept of multi tenancy, which broadens the traditional concept of multi tenancy of a literal shared infrastructure (compute, storage, and other resources) "to any environment that onboards, deploys, manages, and operates tenants through a single, unified experience" [1].

This final report outlines the implementation phase of the multi-tenant SaaS transformation, starting with a quick recap of the system analysis and limitations of Piranha CMS, along with the strategic goals, key quality attributes and cross-cutting concerns that were achieved/followed in the implementation. It covers the evolution of the system's architecture to address implementation challenges, supported by the respective Architecture Decision Records (ADRs). Moreover, it details the adoption of an Agile/Scrum development methodology, including the epics, stories, sprints and other practices carried out. It also introduces key personas and use cases for this scenario, as well as the implementation details necessary to fulfill the requirements outlined in the first report, such as the CD pipeline with a focus on GitOps practices, isolation and per-tenant observability mechanisms. Despite already having performed a live demo on the latest presentation, evidencies of the implementation will also be provided, including work completed after the demo.

2 Recap

2.1 System Analysis & Limitations of PiranhaCMS

As mentioned in the previous report, the standard Piranha CMS architecture, primarily designed as a single-instance platform rather than multi-tenant SaaS, offers a "multi-site" feature with the following identified limitations: **cannot restrict content types per site**, **cannot restrict users to a single site** and **cannot restrict sites to a single media library**. Thankfully, all of the identified limitations regarding this feature were successfully overcomed in our multi-tenant SaaS platform, ContentRus.

In the last report, it was also outlined that **Piranha CMS is currently only suited for scenarios where a single organization manages multiple sites or brands**, rather than a SaaS platform where each customer expects isolation and control over their environment, with several additional gaps being identified. As previously, all of these gaps have since been addressed with a brief overview of the resolved limitations being now presented:

- The **concept of tenant-level users is now supported**, consisting of a significant improvement from the previous global user management system that existed before.
- A **Tenant Management & Onboarding system/interface is now available**, supporting an automated per-tenant environment proviosining process, tenant lifecycle management (from created until provisioned) and tenant-specific configuration. As it will later be highlighted, this constitutes a completely separate system (residing in the same cluster due to development purposes — local cluster), which also features integrated Stripe functionality for subscription payments (after tier selection).
- **Data & Environment Isolation is now implemented**. True multi-tenancy requires strict guarantees that one tenant cannot access another's data (users, content, media, etc.) or resources (e.g. tenant should not be able to access another tenant's deployment from its infrastructure deployments/pods). As it will later be highlighted, the envisioned architecture employs a fully-siloed deployment model, where the tenant's microservices (and databases) are segregated in separate namespaces. Therefore, cross-tenant namespace communication has to be limited (blocked) in order to ensure true isolation.
- **Per-tenant Observability & resource consumption is now implemented**, replacing the previous monolithic approach. Thus, ContentRus provides granular insights into each individual tenant's operations, encompassing tracing, metrics (e.g. CPU usage, memory consumption, network traffic, and request rates) and dashboards for easy interpretation and management.
- **Microservices Architectural Design is followed**. This allows for autoscaling the application microservices, without having to replicate the entire application instance, reducing resource usage and improving scalability, maintainability and deployment agility. This becomes even more crucial as we need to scale in a per-tenant basis, according to the deployment model. For instance, it's possible with the existing architecture to scale, only for a given tenant, his/her hosted site to accommodate periods of high traffic, without having to scale the manager component of the CMS, which constitutes a separate microservice (i.e. only need to scale HostedSite microservice).

2.2 Achieved Strategic Goals, Quality Attributes and Cross-cutting concerns

2.2.1 Strategic Goals

The vision of this multi-tenant scenario is to deliver a **robust, secure, and cost-effective SaaS platform** that supports multiple distinct clients (tenants) operating concurrently on a shared infrastructure, while preserving complete isolation and ensuring performance and scalability (includes addressing the noisy neighbor problem). To fulfill this vision, the platform is guided by the following **strategic goals** — a brief explanation of how these were achieved is also provided:

- **Achieve strict data & environment isolation:** Guarantee that all tenant-specific data (content, users, configurations, logs, etc.) is logically and technically separated. This is achieved through the implementation of a full stack siloed deployment model and network (at the kubernetes cluster level) and authorization (at the service mesh level) policies preventing cross namespace (cross tenant) communication. Therefore, true isolation is achieved by adhering to principles like

defense in depth (multiple layers of security isolation policies) and isolation by default, as both compute and storage resources are segregated per-tenant by default. Until now we've discussing isolation at our infrastructure level. However, as mentioned in the previous report media contents are stored in a separate azure cloud service called Azure Blob Storage. Thus, is also important to ensure data isolation in that regard as well — this is achieved by using separate azure blob containers with scoped credentials for each tenant (these are created with Argo Workflows as it's showcased next).

- **Simplified and Automated tenant onboarding:** Streamline the onboarding process to support an automated and rapid provisioning of new tenants, including a self-service signup with multiple SaaS tiers and an onboarding flow which triggers pre-designed tenant initialization/provisioning scripts and configurations, promoting consistency across the multiple tenants. This is achieved by implementing Argo Workflows and an ArgoCD pipeline, adhering to Continuous Deployment and GitOps practices. Bear in mind, that Argo Workflows are responsible for creating randomized per tenant credentials (both on the cluster and on the cloud's Azure Blob Service, improving security), as well as triggering argoCD application sets (to actual deploy the tenant environment).

- **Supports tenant-level customization:** Although the means for the user/tenant performing this customization is not available, both the infrastructure and ArgoCD continous deployment pipeline fully supports it, only having to specify a different values.yaml file at the git repository (gitops).

- **Monitoring and analytics:** Maintain a comprehensive monitoring system tha provides both a tenant-aware view of the multi-tenant environment.

2.2.2 Key Quality Attributes

The architectural design of a multi-tenant SaaS platform must be driven by a set of critical **quality attributes** — non-functional requirements that influence key system trade-offs and shape long-term sustainability and user satisfaction. The following attributes are essential for the success of this platform:

- **(Tenant) Isolation:** Applying and implementing measures to ensure that tenant resources are protected against any potential cross-tenant access is of upmost importance. As it's been implemented a fully stack silo model, this mostly refers to deployment-time isolation — in contrast to runtime isolation typically found in pooled resource models — where we have separate microservices and databases for each tenant in separate environments. To achieve isolation, isolation policies (network & authorization policies) were defined at deployment-time to prevent the microservice from accessing any resources that belong to other tenants (at the namespace level).

- **Security:** As previously mentioned in the last topic, the isolation of tenant data is paramamount. Security encompasses not only practices related to isolation, but also a broader set of controls and practices, such as Role-Based Access Control (RBAC), which is already implemented by PiranhaCMS, following the Least Privilege Principle. For ContentRus, we have introduced further security mechanisms, such as tenant-aware access and routing to tenant applications based on hostname (there are specific hostnames — website urls — for each tenant, which include the unique tenant id). It's also important to emphasize the role of argo workflows in the CD pipeline for improving security through the automation of creating scoped credentials stored in the tenant's namespace — this also includes creating separate Azure Blob Storage Container credentials scoped to each tenant. This approach is a viable and effective security strategy as long as the Kubernetes cluster itself remains uncompromised. We've also considered integrating HashiCorp Vault for secret management, but choose not to implement it due to its complexity and the constraints of our limited timeframe.

- **(Tenant Onboarding) Usability:** The process of signing up, self provisioning and configuring the new tenant environment must be fast, intuitive and automated, ensuring a smooth onboarding experience, carefully divided in the following steps proper of a SaaS Multi-Tenant Solution:

  1. Tenant/User registration — The tenant initiates the process by providing essential information (e.g., organization name, admin email) through a self-service portal.
  2. Choosing Subscription Tier — The tenant selects a service plan/tier that defines their feature access, resource limits, and support level.

3. Billing Setup — The tenant must provide billing details for the previously chosen plan.
4. Automated Environment Provisioning — Based on the selected tier, infrastructure and application components are provisioned automatically using reusable automated provisioning scripts and tenant templates, ensuring a consistent and rapid setup.
5. Notify the tenant its environment is ready — Once the tenant's environment has been provisioned notify him/her through email, complemented by a clear visual indicator in the self-provisioning interface.

This onboarding/provisioning flow was meticulously followed and showcased in the live demo of the last presentation.

- **Observability:** Full visibility into the system is crucial for proactive monitoring and issue diagnosis. The architecture should include structured logging, distributed tracing and collect metrics and events, offering dashboards for real-time insights on a per-tenant and global context of the environment and infrastructure.
- **Performance & Scalability:** The system must maintain low latency and fast content delivery even under heavy load/traffic. This is specially important in a multi-tenant environment, where one tenant's resource usage might impact the performance and usability of the others (noisy neighbor problem).
- **Maintainability:** The architecture of the system must consider its ability to be easily modified, extended, and repaired with minimal effort and downtime. Therefore, we must consider the maintainability of both the new control plane (developed by us for the multi tenant SaaS solution) and the application plane, which contains PiranhaCMS developed modules. Due to the existance of only two microservices in the application plane (for each tenant) the correspondant PiranhaCMS modules can be used from the open source project with minimal changes according to the open source rules and licensing terms. However, regarding the ContentRus control plane this consitutes a separate project which must be maintained separately and with a different set of guidances from which we distinguish following best practices, such as infrastructure-as-code — using helm and yaml files form managing the tenant's kubernetes resources — gitops with ArgoCD — using Git repositories as a single source of truth to deliver infrastructure as code (IaC) — version control (hosted on github), and documentation (through reports such as this one).
- **Separation of Concerns:** This principle promotes modularity by dividing the system into distinct sections, each responsible for a single functionality. It enhances clarity, reusability, and maintainability of the codebase, providing a better baseline for application scalability of focused level components based on load, preventing overprovisioning and waste of resources.

2.2.3 Cross-Cutting Concerns

Some aspects span multiple layers of a system and are not restricted to a specific domain. These are referred to as **cross-cutting concerns**, and they must be consistently applied across the software architecture. We've already been discussing cross-cutting concerns throughout the previous subsections (and last report as this is a recap) where strategic goals and quality attributes were discussed, with the following cross-cutting concerns being identified:

- **Security and Tenant Isolation** — Security and authentication are fundamental in any system, specially a multi-tenant environment. It's necessary to extract tenant context and propagate towards the descending services, enabling content-aware routing. Tenant isolation must also be enforced (in our case at deployment time with a fully siloed stack model), blocking any attempt to cross-tenant access. Furthermore, Role-Based Access Control (RBAC) should restrict within each tenant's environment the user's access to resources, according to the Least Privilege Principle.
- **Logging and Observability** — Logs, metrics, and traces are collected within each tenant's scope and stored in isolation to each tenant namespace. Observability covers system health, resource usage, and tenant-specific performance, enabling per-tenant monitoring and dashboards to detect issues like the noisy neighbor problem.

- **Error Handling and Fault Isolation** — Robust error handling is critical in ensuring that failures are contained and do not propagate across tenants. To achieve this we employed containerized service instances, across separate namespaces. Mechanisms should also be put into place so that everytime a error does occur, users receive friendly error messages that do not expose internal details.

3 System Architecture

In the previous report, in order to design the architecture for the multi tenant SaaS system, the architectural design methodology of choise was Attribute-Driven Design (ADD) method because Multi-Tenancy is heavily driven by specific, critical quality attributes — namely Security, Isolation, Scalability, Performance and (Onboarding) Usability. These attributes are essential to the success of a system built on a multi-tenant architecture. Furthermore, Domain-Driven Design (DDD) was also followed with the goal of understanding and modeling the business domain into subdomains, categorized into three different categories: core subdomains, generic subdomains and supporting subdomains. The identified subdomains, including Core Subdomains (Tenant Management, Tenant Provisioning, Onboarding, Content Management), Generic Subdomains (Billing, User Management), and Supporting Subdomains (Media Storage, Notifications), aligned effectively with the final product's development. However, the predefined architecture suffered some changes since that report due to implementation issues, which will be highlighted bellow and documented through the use of Architectural Decision Records (ADRs).

The architecture shown in Figure 1 is the result of applying the ADD methodology across three different iterations in the previous report, focusing on Separation of Concerns, Tenant Isolation and Performance & Scalability, in that order.

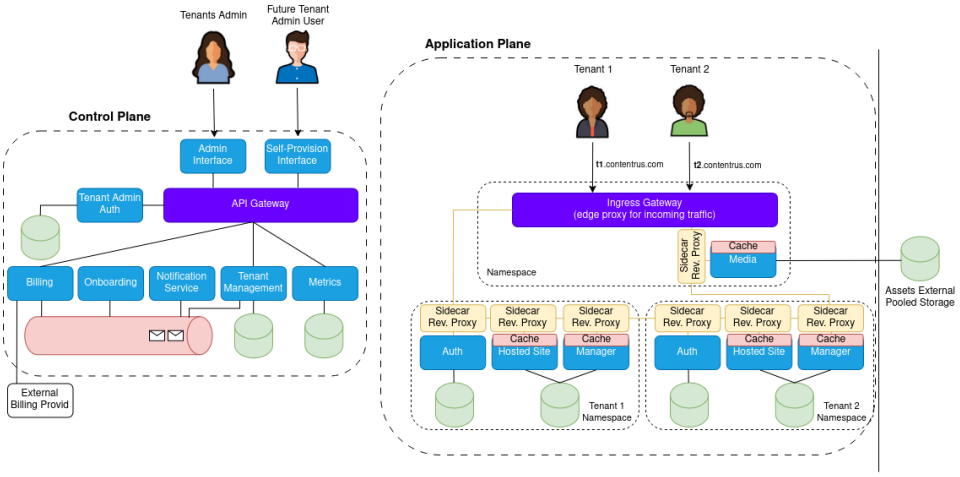

Figure 1: Architecture Diagram achieved through three iterations of ADD methodology

The decisions behind the designed architecture, as well as the respective ADRs have already been discussed in the previous report. Nevertheless, it's important to provide a concise, overall explanation of the designed architecture. Firstly, the architectural design allocates different responsibilities to the Control Plane and the Application Plane. The control plane is responsible for managing and orchestrating all the steps needed to get a new tenant introduced into the SaaS environment, whereas the Application Plane handles tenant-specific operations like content management, hosting the website and media delivery. Furthermore, the control plane is responsible for cluster state and per-tenant observability.

Both sub-architectures of the different planes follow a microservices architecture, where each service is designed with a single responsibility in mind, ensuring a clear separation of concerns and providing scalability, flexibility, ease of deployment and fault isolation. The Control Plane uses an API Gateway as a unified entry point which abstracts the complexity of multiple underlying microservices, providing seamless request routing, robust authentication and authorization, and efficient load balancing... Therefore, the control plane is divided into separate small, independent, and loosely coupled microservices:

- **Self Provision Interface:** User Interface, through which each tenant will configure their account, choose their SaaS plan, confirm the subscription payment and be notified of when their environment is ready.

- **Billing:** Integrates with a Billing External Provider (in this case stripe) to securely process payments, manage invoices and offer seamless subscription management.
- **Onboarding:** Responsible for provisioning the tenant environment by utilizing pre-configured provision scripts and templates.
- **Notification Service:** Responsible for notifying tenants of important events, such as completing the tenant environment setup, through email.
- **Tenant Management:** Responsible for storing all the information about a tenant (company name, identity configuration, tier...), as well as the lifecycle of a tenant.
- **Metrics:** Collects, stores, and analyzes performance and resource usage data across the cluster and multi-tenant environment.

The Control Plane also possesses a message queue for asynchronous communication between the microservices, enabling decoupling and improved scalability. It handles tasks such as sending notifications, triggering onboarding workflows and propagating tenant lifecycle events, improving fault tolerant and allowing for independent scaling of services.

However, the Application Plane architecture employs a Service Mesh using the sidecar pattern with reverse proxies to handle service-to-service communication and tenant-aware routing. This approach provides key benefits such as load balancing, circuit breaking, and other advanced traffic management features, as outlined in the corresponding Architectural Decision Record (ADR) from the previous report, which were central to the decision to adopt a service mesh. The Application Plane segregates the previously monolithic implementation of PiranhaCMS into multiple microservices:

- **Auth:** Manages user authentication within tenant contexts, ensuring secure access.
- **Hosted Site:** Delivers tenant-specific websites, handling runtime hosting tasks.
- **Manager:** Responsible for the management of the CMS platform.
- **Media:** Stores and retrieves tenant-specific media assets, leveraging pooled compute and integrating with external storage solutions (in this case cloud's Azure Blob Storage Service).

The architecture highlighted in Figure 1 reflects the consensus reached through the application of the ADD methodology, across three iterations. However, the sub-architecture of the Application Plane presented significant implementation challenges due to tight coupling between modules in the current PiranhaCMS solution. Therefore, as the ADD methodology supports and encourages iterative refinement a new iteration was performed and documented here to accommodate these new challenges and insights obtained during implementation.

3.1 ADD Fourth Iteration

3.1.1 Establish Iteration Goal and Input Selection

The iteration goal is to design the system's architecture to meet both functional and quality attribute requirements. The most critical quality attribute to address in this iteration is **Maintainability**. The fact that the underlying open-source solution exhibits tight coupling between modules prevents splitting the system into microservices at this stage, as the related implementation problems could not be overcomed. As a result, the team must incorporate this new constraints into the architecture with minimum diviation from the original design and maximizing maintainability by reutilizing the PiranhaCMS modules.

3.1.2 Choose elements of the system

The elements of the system which are subject to change correspond to the microservices that were not possible to segregate due to the tight coupling problems. These microservices correspond to the Auth Service and the Media Service, which were too coupled to the manager service.

3.1.3 Choose Design Concepts that satisfy the iteration inputs

In order to satify the iteration goal and the input architectural drivers the following design concept was followed: **Reusability of the PiranhaCMS Modules**. Instead of replacing existing modules, the architecture leverages and extends the built-in capabilities of PiranhaCMS where possible, ensuring maintainability while reducing rework and minimizing divergence from the open-source foundation.

3.1.4 Instantiate Architectural Elements, allocating responsabilities and defining interfaces

Architectural Elements & Responsibilities

This iteration refines the Application Plane, reducing the initial set of microservices into the following list:

- **Hosted Site:** Delivers tenant-specific websites, handling runtime hosting tasks.
- **Manager:** Responsible for the management of the CMS platform. Also manages **user authentication** within each tenant context, ensuring secure access. Moreover, it stores and retrieves **tenant-specific media assets**, integrating with external storage solutions.

Thus, the manager microservice incorporates the responsibilities of the previously designed auth and media microservices.

Interfaces

The interfaces for each service will not be outlined in this iteration, due to this particular microservice setup not changing the business logic of the service components, comparatively to the PiranhaCMS solution.

3.1.5 Sketch Views and Record Design Decisions

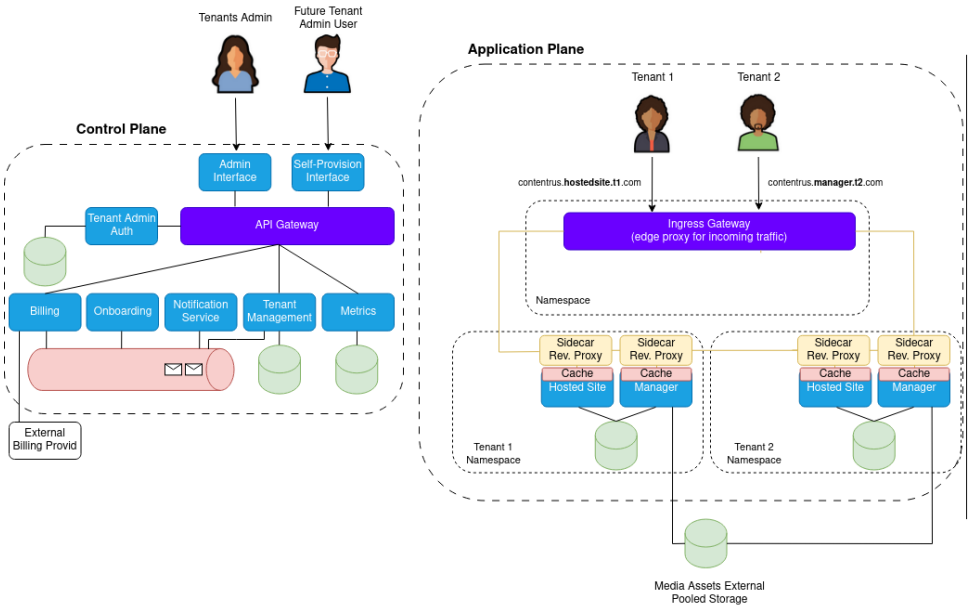

Figure 2: Architecture Diagram achieved through four iterations of ADD methodology

Figure 2 showcases the achieved solution at this iteration. The Control Plane remains unchanged comparatively to the previous architecture, whereas the Application Plane has suffered some changes which will be highlighted now. Each tenant environment/namespace is now only constituted by the hosted site (serving built CMS websites) and manager microservices. The setup of the service mesh and ingress gateway remains similar with tenant url routing for both the hosted site and manager microservices (contentrus.**hostedsite.t1**.com and contentrus.**manager.t2**.com, respectively). The standalone authentication microservice, which previously served only the manager service, has been integrated and merged into the manager service itself. Similarly, the media service has also been integrated into the manager service.

The design decisions and racionale behind this architecture are documented in the following Architectural Decision Record (ADR):

- **ADR: Integration of Auth and Media Services into Manager Service**

  Context — The existent ContentRus system application plane follows a microservices architecture comprised of 4 microservices: Auth Service, HostedSite Service, Manager Service and Media Service. Due to implementation constraints regarding the segregation of the Auth Service and Media Service, measures need to be taken into account to guarantee the functionality of the system and improve maintainability (based on PiranhaCMS open source).

  Decision — To address these constraints while maintaining system maintainability and functional correctness, the Authentication Service and Media Service were integrated into the Manager Service, incorporating their responsabilities and functionalities.

  Consequences

  – Simplifies deployment and reduces service orchestration complexity.
  – Improves maintainability within the current architectural constraints, as it's possible to easily incorporate the respective PiranhaCMS modules with minimal changes.
  – Reduces flexibility and scalability of Auth and Media logic, which are now tightly bound to the Manager Service lifecycle.
  – May require architectural rework in the future if full microservice decomposition becomes feasible or necessary.

3.1.6 Analysis of current design and review of the iteration goal and design objectives

The achieved design in this iteration focused on achieving full functionality and adhering to the Maintainability quality attribute within the app. This was accomplished by integrating the auth and media services into the manager service, after encountering implementation limitations when segregating these two services.

This corresponds to the final iteration and architecture, corresponding to the one that was actually implemented and presented on the presentation demo.

4 Personas & Use Cases

To better understand the different types of users interacting with the multi-tenant Contentrus SaaS platform, we've identified representative personas along with their key use cases. These help clarify how various user groups engage with the platform and what goals they aim to achieve.

4.1 Personas

4.1.1 Business Owner

Name Margaret Carter
Age 52
Role Digital Marketing Expert & Business Owner
Background Margaret owns a digital marketing agency. With extensive experience in launching promotional campaigns, she seeks a solution that enables her to quickly register her business and autonomously provision a CMS environment without technical involvement.

4.1.2 CMS Administrator

Name John Walker
Age 56
Role Manager at Elizabeth's agency
Background James is responsible for managing digital content on the company's CMS - coordinate content campaigns, by creating and updating landing pages.

4.1.3 CMS User

Name Emily Thompson
Age 20
Role Marketing Student & Site Visitor
Background Emily is a university student who frequently browses websites for inspiration and information. She regularly visits campaign landing pages for academic interest and consumer engagement.

4.1.4 Platform Administrator

Name Samuel Wilson
Age 43
Role DevOps Engineer
Background Sam is responsible for maintaining the infrastructure behind the Contentrus platform. He monitors resource consumption, ensures tenant isolation, and handles performance optimization for all environments.

4.2 Use Cases

4.2.1 Tenant Onboarding & Environment Provisioning

Persona: Margaret Carter (Business Owner)

Margaret accesses the Contentrus portal and begins the onboarding process. She **registers an account** by providing her email and creating a secure password. Once logged in, she **fills in the necessary company details**, including the organization's name, address, and country. Afterwards, she selects **the subscription plan** that best fits her agency's budget and proceeds to **complete the payment**. Shortly after, Margaret receives a confirmation email notifying her that the environment has been successfully provisioned. At the portal interface, a success message is displayed alongside direct links to access both the Content Manager and the Hosted Site.

4.2.2 Creating a Marketing Landing Page

Persona: John Walker (CMS Administrator)

John **logs into the Contentrus Manager portal** to manage a new campaign. Once inside the manager interface, he accesses the content editor and begins **building a new landing page** for an upcoming promotional campaign. Then, he **publishes the page**.

4.2.3 Accessing a Campaign Page

Persona: Emily Thompson (CMS User)

Emily visits the agency's hosted site. As **she browses the website**, she discovers the recently published landing page for a new marketing campaign.

4.2.4 Monitoring Resource Usage

Persona: Samuel Wilson (Platform Administrator)

Samuel goes to the system dashboard to monitor infrastructure health. He **checks real-time metrics** related to each tenant, including CPU usage, memory consumption, and network traffic. By doing so, he ensures that resources are fairly distributed and that no tenant is negatively impacting the system's overall performance.

5 Scrum Agile Methodology

To develop the proposed solution, we adopted an Scrum/Agile methodology, which provided a flexible and iterative approach to project management and software development.

We utilized JIRA as our main project management tool to organize and track the progress of our work. Through JIRA, we were able to define user stories, assign tasks, manage sprints, and monitor the status of each feature in real time. This facilitated clear communication within the team and ensured that everyone remained aligned with the project goals.

As part of our Agile practices, we held Daily Scrum meetings to discuss progress, address blockers, and plan the day's work collaboratively. Sprint Planning sessions were conducted at the beginning of each sprint, during which we collectively estimated the complexity of user stories using story points based on the Fibonacci sequence (e.g., 0, 1, 1, 2, 3, 5, 8, 13). This approach allowed us to better assess the effort required for each task and manage workload distribution effectively.

Additionally, we leveraged GitHub for version control, hosting our codebase and enabling collaborative development, ensuring code integrity and traceability throughout the development lifecycle.

5.1 Roadmap Overview

In the first delivery, we defined a detailed roadmap outlining four sprints, each planned to last one week. This structure was designed to facilitate a steady and manageable development pace, allowing us to incrementally build and refine the solution.

However, due to unforeseen time constraints, the originally planned four sprints were condensed into three. Despite this adjustment, we carefully prioritized the most critical features and tasks to ensure that the core functionality was delivered efficiently and within the scheduled timeframe.

This experience highlighted the importance of flexibility within Agile methodologies, enabling us to adapt our plan while still maintaining focus on the key objectives and delivering a valuable product increment.

5.2 Epics and User Stories

To organize and manage the work more effectively, we structured the project around four Agile epics, each representing a major area of functionality:

- **Control Plane Base Infrastructure**: Set up the control plane foundational infrastructure, including the containerization of control plane services and the configuration of all necessary components required for their deployment.
- **Control Plane Core Services**: Design and implement the essential services of the control plane, including the Notifications Service, Billing Service, Onboarding Service, Tenant Self-Provisioning UI, and the Tenant Management Service.
- **Application Plane Base Infrastructure**: Involved refactoring the initial monolithic architecture into a microservices-based design, containerizing the individual services, configuring all necessary components for deployment, and integrating observability features to monitor and analyze service behavior on a per-tenant basis.

- **CD Provisioning**: Automation of the tenant provisioning process through a continuous deployment pipeline, enabling seamless and consistent delivery of resources and services. This is mainly achieved through the use of Argo Workflows and ArgoCD.

Each epic was decomposed into smaller units of work — such as user stories and individual tasks — to enable incremental development and enhance team collaboration.The user stories adhered to a consistent structure to clearly articulate user needs and intended outcomes, following the format: **"As a [user], I want to [action], so that [benefit]"**. Each user story was also supplemented with well-defined acceptance criteria to ensure clarity, guide implementation, and validate completion. These acceptance criteria were defined using the Gherkin syntax: **Given [precondition], When [action], Then [outcome]**. Figure 3 highlights an example of a definition of a user story, namely a user story for picking the multi tenancy SaaS subscription tier and complete the subscription payment.

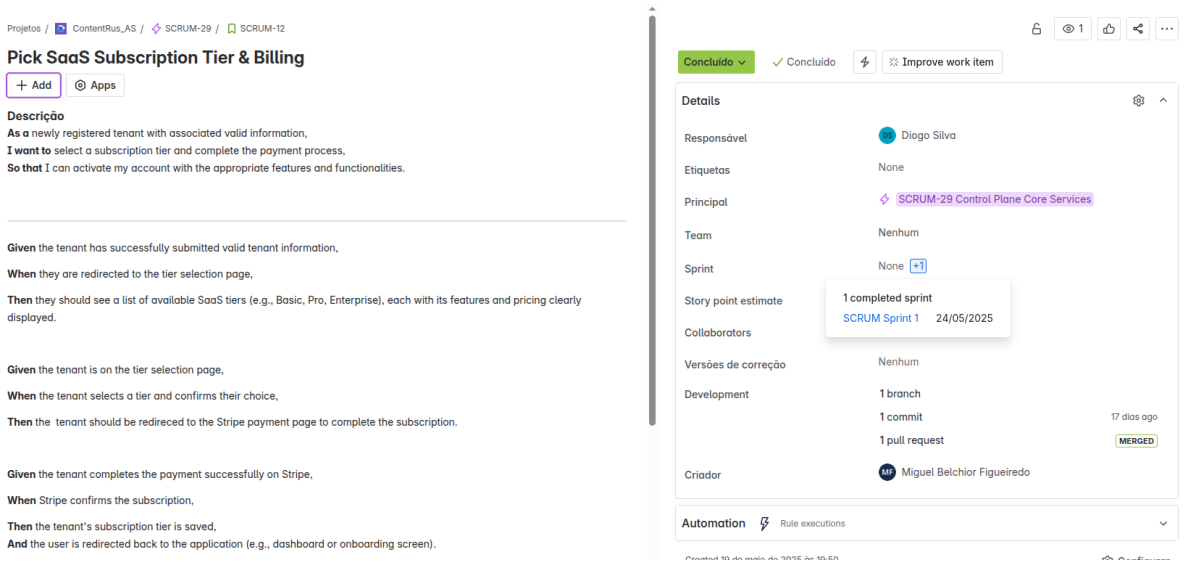

Figure 3: Example of User Story SCRUM-12: Pick SaaS Subcription Tier & Billing

One of the deliverables resulting from this user story is the integration with the stripe subscription page highlighted in Figure 4.

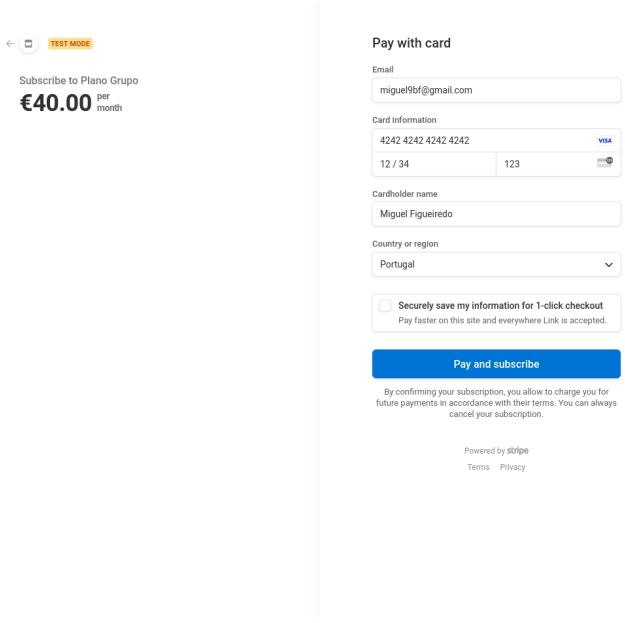

Figure 4: Subscription-based Stripe Integration

5.3 Sprint Review

At the end of each sprint, we conducted a Sprint Review to reflect on the work completed, assess what went well, identify areas for improvement, and define the next steps for the follwing sprint.

5.3.1 Sprint 1

The primary objective of the first sprint was to implement the foundational multi-tenancy features within the control plane, enabling tenant registration, authentication, and environment isolation. This included the initial setup of tenants and the integration of billing functionalities.

Additionally, within the application plane, this sprint focused on implementing segregation and provisioning scripts for the Manager and Hosted Site microservices. It also covered the deployment and configuration of the Istio ingress gateway and service mesh to ensure secure and efficient traffic routing for each tenant's environment.

Together, these efforts established the core infrastructure required to support tenant environments and laid the groundwork for further development in subsequent sprints.

Figure 5 shows the burndown chart for the first sprint. The chart demonstrates that we successfully completed nearly all of the planned work, reducing the remaining story points from 18 to 1 by the end of the sprint

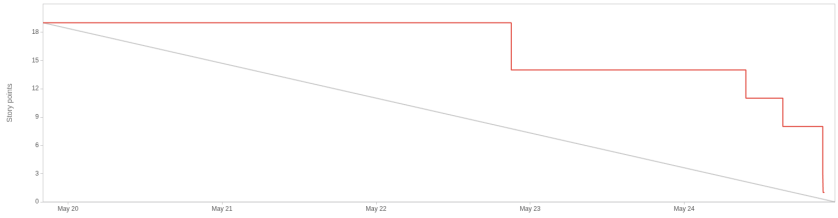

Figure 5: Burndown Chart for Sprint 1

5.3.2 Sprint 2

In this sprint, the focus was on advancing towards a minimum viable product (MVP) by enhancing observability across both the control plane and application plane on a per-tenant basis. Network and authorization policies were implemented to enforce strict tenant isolation. Additionally, we integrated tenant-aware media storage access to Azure Blob Storage, ensuring secure and isolated data management for each tenant.

The tenant onboarding process was further automated using Argo Workflows & ArgoCD, enabling continuous deployment and streamlined management of tenant environments. These improvements collectively strengthened the platform's security, monitoring capabilities, and deployment automation.

Figure 6 shows the burndown chart for the second sprint. The chart demonstrates that the team completed a significant portion of the planned work, reducing the story points from 22 to 9 by the end of the sprint.

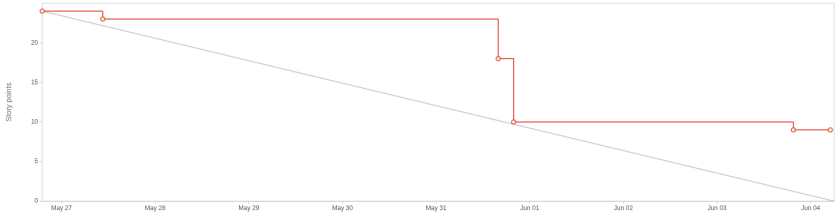

Figure 6: Burndown Chart for Sprint 2

5.3.3 Sprint 3

In this sprint, the goal was to finalize the MVP for the multi tenancy project, incorporating all the unfinished items and implementing all the changes requested during the presentation demo. Therefore, in this

sprint it were added some network and authorization policies for complete isolation. We also successfully rectified the webhook responsible for sending a queue message to trigger email notifications upon successful tenant provisioning and integrated visual confirmation of the tenant environment's successful provisioning status into the self-provisioning interface, as per the professor's requirements.

Figure 7 showcases the burndown chart for the third and last sprint. The chart demonstrates that the team completed the totality of the planned work within the sprint timeframe, accounting for 6 story points. It is also evident from the chart that this sprint was considerably shorter, having a duration of only two working days.

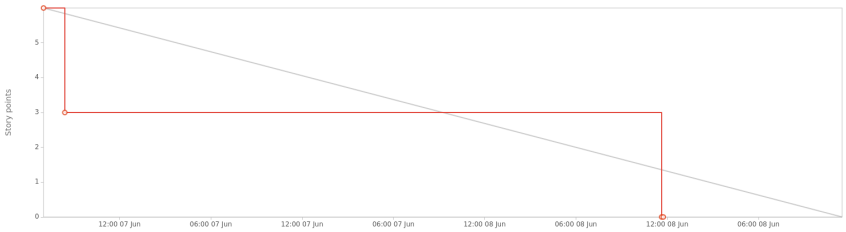

Figure 7: Burndown Chart for Sprint 3

5.4 Pull Requests

To ensure code quality and facilitate collaboration, we adopted a pull request (PR) workflow throughout the development process. Every new feature, bug fix, or improvement was implemented in a separate branch (linked to the corresponding JIRA issue) and submitted as a pull request for review. This process allowed team members to provide feedback, suggest improvements, and catch potential issues before merging changes into the main branch. Before merging a pull request, a corresponding code review must be conducted and approved to ensure code quality and adherence to the project standards. Figures 8 and 9 highlight a practical example of a pull request with a code review, from one of our github pull requests (associated with a JIRA issue).

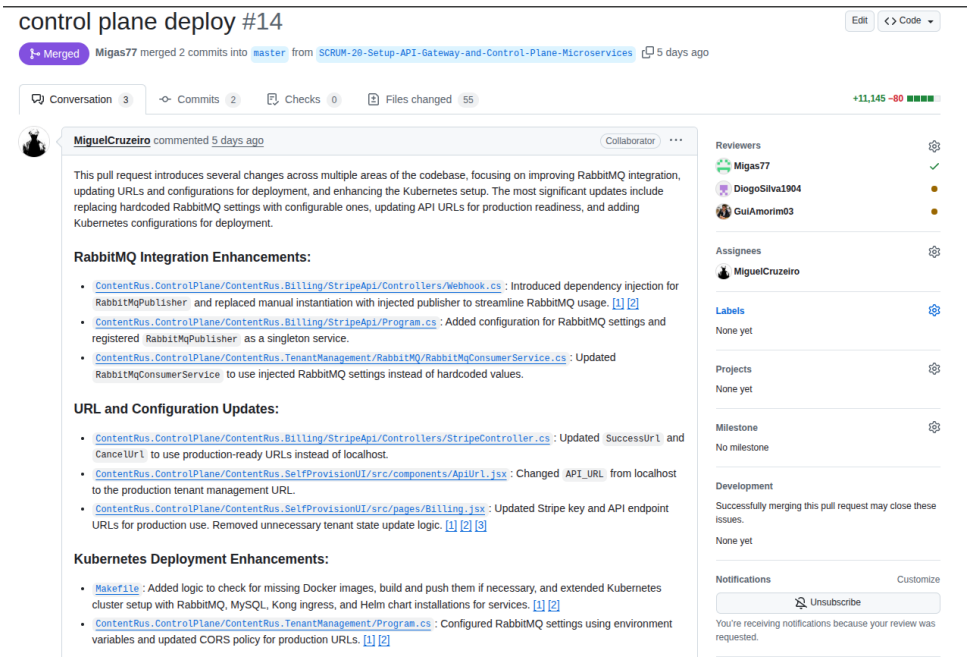

Figure 8: Example of a pull request with code review comments in our development workflow

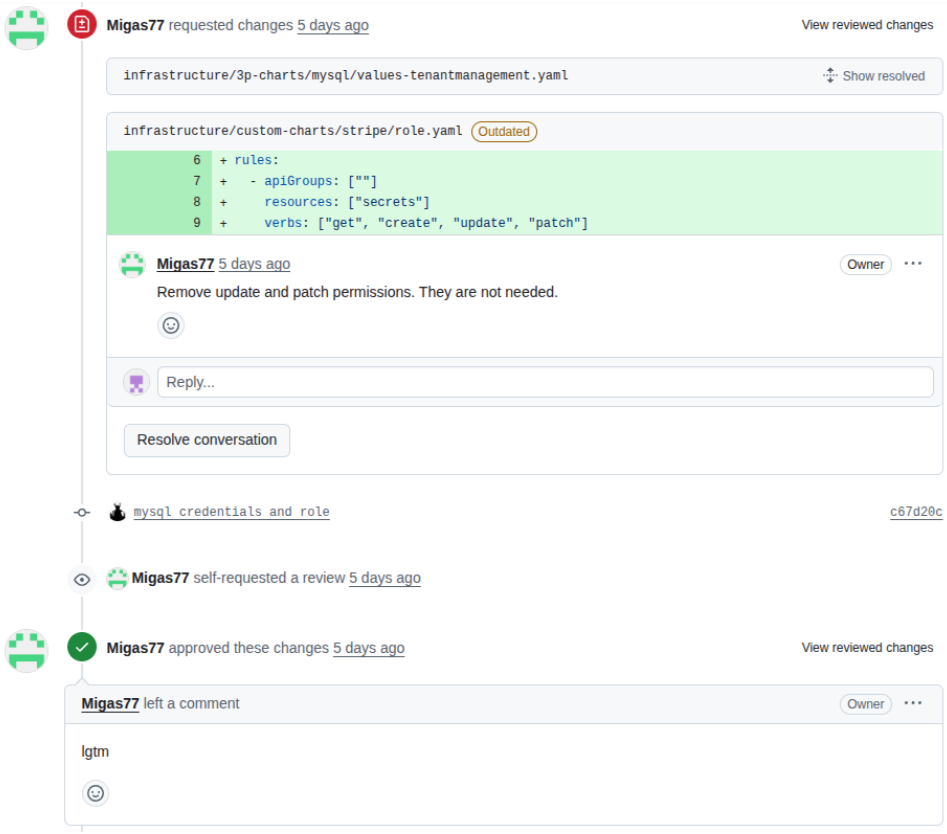

Figure 9: Example of core review comments in a pull request

6 Implementation Details and Practices

6.1 Development Environment with k3d local cluster and registry

To facilitate rapid development, testing, and demonstration, the development of the project was based on a local Kubernetes environment managed by k3d.

The entire setup and teardown process was automated via a Makefile, ensuring a consistent and reproducible development environment for all team members. The Makefile creates the local registry served as the destination for all custom-built container images for the project's microservices. Then, the cluster is created and the script proceeds to deploy the entire baseline infrastructure required by the platform.

This automated setup not only simplified the infrastructure setup process for developers but also ensured that the local environment closely mirrored the architecture and tooling of a production deployment, allowing for realistic testing of features like tenant provisioning, isolation, and observability.

6.2 CD Pipeline

Given ContentRus corresponds to a Multi tenancy SaaS platform, where tenants can self-provision their environments through a self-service interface (after choosing tier and paying for the subscription), a fully automated and scalable deployment mechanism is essential. Supporting self-provisioning requires a pipeline that not only provisions infrastructure on demand but also ensures consistency, reliability, and isolation for each tenant.

Therefore, with the goal of ensuring a rapid, reliable, and consistent delivery of infrastructure ContentRus adopts a **Continuous Deployment (CD)** strategy, based on the **GitOps** pattern of using Git repositories (hosted on GitHub) as the source of truth for defining the desired application state. In order to achieve this, ContentRus uses ArgoCD which is a declarative continuous delivery tool for Kubernetes, which uses (in our case) helm charts hosted on github repos to specify the desired application state. This offers multiple benefits by making all infrastructure changes through Git commits, which

enables automated, auditable, and version-controlled deployments while ensuring consistency and reliability through continuous synchronization between Git and the live environment on the cluster. Argo Workflows has also been used in order to specify all of the steps representing our CD pipeline, which will now be detailed. Here it'll only be discussed the CD pipeline with the full onboarding flow, starting on the self provision interface and ending with this CD pipeline, being highlighted in another section.

When deploying the base cluster an argo workflow template with the description of the deploy workflow is created, which is highlighted in the code snippet bellow and in the following github link [2]. This workflow template takes as input a given `tenantId` for provisioing and performs three sequential steps — bear in mind that the cluster resources created at each step are available for the next one and that if any step fails, the subsequent steps will not start:

- `set-tenant-namespace-and-secrets` — This step is responsible for creating a namespace for the tenant based on its `tenantId`, generating random credentials (for services such as the database), and storing them as Kubernetes secrets within the corresponding namespace in the cluster.
- `set-tenant-azureblob-credentials` — This step is responsible for, using the azure-cli, creating a separate azure blob container named 'tenant{`tenantId`}' within 'contentrus' resource group and 'contentrustenants' storage account. Furthermore, it creates tenant-scoped credentials for that blob container storing them in the kubernetes cluster in the respective namespace. This is of upmost importance to achieve isolation as, despite the storage account being shared by all tenants (due to budget reasons), each tenant only has access to the corresponding azure blob container in the Azure Blob Storage Service.
- `trigger-tenant-application-set` — This last step is responsible for triggering the ApplicationSet used for tenant provisioning. It will be explained next the concept of application sets (argoCD concept), but for now is only important to note that deployments in our ArgoCD setup are triggered by committing changes to a predefined GitHub repository. Thus, this commit to the repo is actually what triggers and makes ArgoCD provision the environment and Kubernetes resources according to the desired state.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: WorkflowTemplate
metadata:
  annotations:
    workflows.argoproj.io/description: 'Deploy Argo Workflow!.'
    workflows.argoproj.io/maintainer: '@belchior'
  name: tenant-provisioning-with-credentials-template
spec:

  # This is the entrypoint of the workflow
  # (the template that will be executed first)
  entrypoint: steps-template
  arguments:
    parameters:
    - name: tenantId
  volumes:
  - name: github-creds-vol
    secret:
      secretName: github-creds
      items:
      - key: ssh-private-key
        path: id_rsa

  templates:

  # Template Invocators
  - name: steps-template
    steps:
    - - name: set-tenant-namespace-and-secrets
        template: tenant-namespace-and-secrets-script
    - - name: set-tenant-azureblob-credentials
        template: tenant-azureblob-credentials-script
```

```
    - - name: trigger-tenant-application-set
        template: tenant-git-commit-script

  # Template Definitions
  # ...
```

This workflow steps are triggered by creating a workflow (instance of this workflow template) using the kubernetes client after the Onboarding Service receives a provisionRequest message through the message queue. An example of the execution of an argo workflow for provisioning a tenant is showcased in Figure 10.

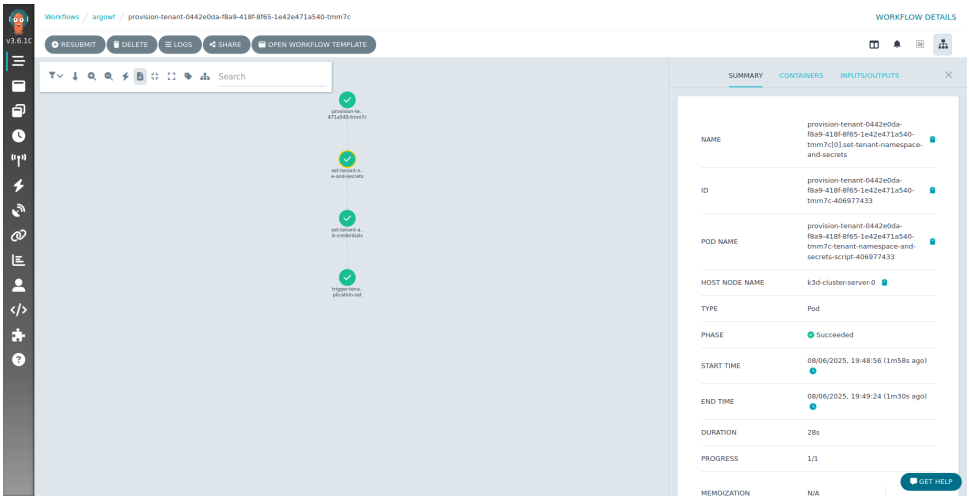

Figure 10: Example of Argo Workflow for creating tenant credentials and triggering ArgoCD with a git commit

Until now we've been discussing the role of Argo Workflows in the CD Pipeline. However, the rest of the provisioning process is handled by ArgoCD and, consequently, it's necessary to grasp the concepts of Applications and ApplicationSets in ArgoCD. In ArgoCD, an Application defines a deployable unit by specifying the source (e.g., Git repo), destination cluster, and namespace, whereas an ApplicationSet automates the creation of multiple Applications based on a template and a set of input parameters (in this case tenant IDs). Therefore, the following application set defines the template for provisioning multiple tenant applications [3]:

```
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: tenant-applications
  namespace: argocd
  annotations:
      notifications.argoproj.io/subscribe.webhook:
spec:
  generators:
  - git:
      repoURL: 'https://github.com/Migas77/contentrus.multiTenancy.
          argocdHelmCharts.git'
      revision: HEAD
      directories:
      - path: tenants/*

  template:
    metadata:
      name: '{{ path.basename }}'
      namespace: argocd
    spec:
      project: project-tenants-contentrus
```

```
sources:
  - repoURL: 'https://github.com/Migas77/contentrus.multiTenancy.git'
    targetRevision: HEAD
    path: infrastructure/umbrella
    helm:
      valueFiles:
      - $values/{{ path }}/values.yaml
  - repoURL: 'https://github.com/Migas77/contentrus.multiTenancy.
      argocdHelmCharts.git'
    targetRevision: HEAD
    ref: values

destination:
  # server from argocd cluster list --grpc-web
  server: https://kubernetes.default.svc
  namespace: '{{ path.basename }}'

syncPolicy:
  automated:
    prune: true
    selfHeal: true
  syncOptions:
  - CreateNamespace=true
  - ApplyOutOfSyncOnly=true
```

The previous application set uses two github repos (HEAD) as sources for our tenant deployments:

- `https://github.com/Migas77/contentrus.multiTenancy.git` contains the baseline infrastructure configuration for a given `tenantId` using Helm Charts. It includes the configuration for the corresponding application plane microservices (HostedSite and manager), as well as the mysql database and necessary network and authorization policies for isolation. All of these resources defined as Helm Charts are aggregated through the use of an umbrella chart on path `infrastructure/umbrella`.
- `https://github.com/Migas77/contentrus.multiTenancy.argocdHelmCharts.git` provides under the path `tenants/tenantId` the `values.yaml` file with tenant specific configuration, enabling customization based on subscription tiers or tailored configurations for individual tenants. These files are the ones that actually trigger the initial provisioning of the environment (by triggering the ApplicationSet), when pushed by an Argo Workflow "bot" during the final step of the previously described workflow.

It's also important to note that in the previous code snipped the applicationSet is configured for the default kubernetes cluster (in k3d), which is also the cluster where the control plane resides. However, this implementation and ArgoCD allow for this two planes to be separated in two different clusters — they just don't are for development reasons.

Additionally, the `syncPolicy` is configured with **automated** synchronization enabled, allowing ArgoCD to continuously apply changes without manual intervention. The `prune: true` option ensures that obsolete resources (no longer defined in Git) are deleted, while `selfHeal`: true automatically corrects any deviation between the live cluster state and the desired Git state. The `syncOptions` include `CreateNamespace=true` to automatically create the target namespace if it does not exist, and `ApplyOutOfSyncOnly=true` to optimize performance by applying only the resources that are out of sync (if not specified by default ArgoCD would apply everything).

Thus, once the `values.yaml` file is committed to `https://github.com/Migas77/contentrus.multiTenancy.argocdHelmCharts.git`, ArgoCD — following a pull based model — will periodically detect the changes in the repository (by default, every 3 minutes). Upon detection, it will begin synchronizing the state and deploying the corresponding Kubernetes resources to match the desired configuration. An example of ArgoCD provisioning a tenant environment is shown in Figure 11.

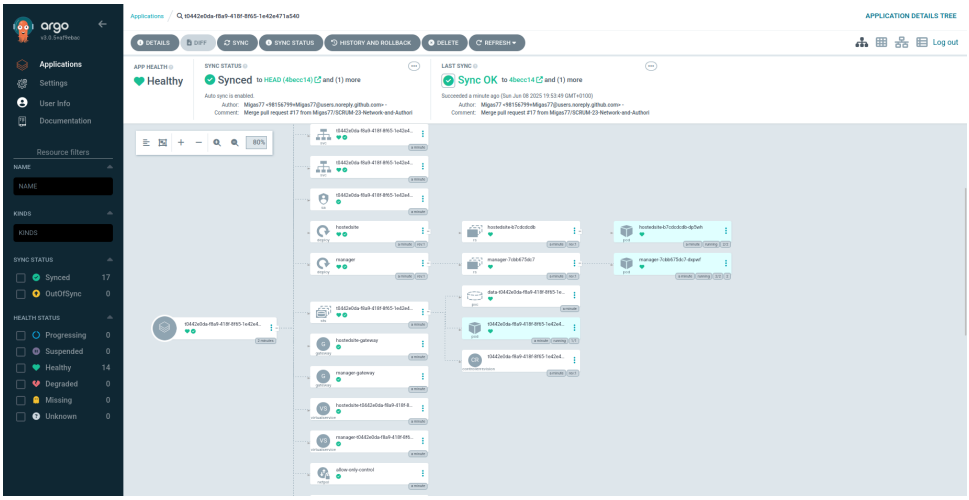

Figure 11: Example of ArgoCD Tenant Provisioning

However, there is still a final step missing, which corresponds to notifying the tenant (both on the interface and through email) of the environment provisioning after it's completed (only after this the tenant's user can use the CMS solution). This is achieved using ArgoCD Notifications in combination with a message queue and our tenant management and notification services, which will be explained in detail bellow. The following code snippet [4] highlights the use of Argo CD Notifications to inform a tenant about the successful or unsuccessful provisioning of their environment. This ConfigMap configures Argo CD so that, when the `on-deployment-finished` condition is triggered — more specifically, when an Argo CD application reaches a Synced state and is either Healthy or Unhealthy — a POST request is sent to the webhook endpoint `http://onboarding.control.svc.cluster.local/webhook/deployment-status`. This endpoint belongs to the onboarding service deployed in the control(-plane) namespace. Upon receiving this request, the onboarding service publishes a notification event to a message queue, which is then consumed by the tenant management service, containing the Argo CD application name, its sync and health status, the tenant namespace (including the `tenantId`), and the timestamp at which the deployment status was registered. The tenant management microservice processes this information, updating the tenant's status accordingly and emitting another message through the message queue to the notifications service, which then sends an email notification informing the tenant of the successful provisioning of their environment. Moreover, once the tenant management service updates the tenant state, the self-provisioning interface is also refreshed to reflect the successful deployment, containing hyperlinks for the manager and hosted sites of the corresponding tenant.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-notifications-cm
  namespace: argocd
data:
  service.webhook.deployment-status: |
    url:  http://onboarding.control.svc.cluster.local/webhook/deployment-status

    headers:
      - name: Content-Type
        value: application/json

  template.deployment-finished: |
    webhook:
      deployment-status:
        method: POST
        body: |
          {
              app : {{ .app.metadata.name }},
              status : {{ .app.status.sync.status }},
              health : {{ .app.status.health.status }},
```

```
            tenantNamespace : {{ .app.spec.destination.namespace }},
            timestamp : {{ now | date 2006-01-02T15:04:05Z07:00 }}
        }

  trigger.on-deployment-finished: |
    - name: on-deployment-finished
      description: Trigger when app is synced and healthy or unhealthy
      send:
        - deployment-finished
      when: app.status.sync.status == 'Synced' and (app.status.health.status ==
          'Healthy' or app.status.health.status == 'Unhealthy')

  subscriptions: |
    - recipients:
        - deployment-status
      triggers:
        - on-deployment-finished
```

In conclusion, ContentRus implements a fully automated and scalable tenant self provisioning pipeline by leveraging a GitOps-based Continuous Deployment strategy with ArgoCD and Argo Workflows. This approach ensures consistent, reliable, and auditable infrastructure delivery for each tenant. By integrating dynamic Argo Workflows to automate one-time actions such as namespace setup and credential generation, security and isolation are achieved by securing the creedntials as secrets in the intended tenant's namespace. This workflow also triggers the ApplicationSets for automated scalable multi-tenant provisioning, with ArgoCD Notifications being used to communicate the deployment status to the end user/tenant.

6.3 Isolation

In our multi-tenant implementation of PiranhaCMS, tenant isolation is one very important concern. Its crucial that each tenant has his data and resources segregated and protected from anouthorized access from other tenants. We achieve this by applying a strategy that combines namespace-level segregation with network and authorization policies. Because each tenant has its own dedicated namespace, at the time of provision, authorization and network policies are applied to each tenant specific namespace.

6.3.1 Kubernetes Network Policy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-only-control
  namespace: {{ .Release.Namespace }}
spec:
  podSelector: {}
  policyTypes:
    - Egress
    - Ingress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: control
    - from:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: argocd
    - from:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: istio-system
    - from:
        - namespaceSelector:
```

```
            matchLabels:
              kubernetes.io/metadata.name: common
    - from:
        - podSelector: {}
  egress:
    - to:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: control
    - to:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: common
    - to:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: argocd
    - to:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: istio-system
    - to:
        - podSelector: {}
    - to:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: kube-system
      ports:
        - protocol: UDP
          port: 53
        - protocol: TCP
          port: 53
```

The first layer of protection is at the networl level (OSI Layer 3) using this kubernetes network policy. This policy controls inter-namespace traffic on a default-deny basis, allowing communication only when explicitly permitted.

At ingress level (every traffic that arrives to that namespace), we defined as legitimate incoming traffic from the control namespace (where the provision services are located), argocd (required for the deployment of tenants), istio-system (for service mesh operations), common (where the opentelemetry services reside) and pods within the same tenant namespace.

For egress, it restricts outgoing traffic to the same set of allowed namespaces, pods within the same namespace and also to the kube-system namespace which is required for dns resolution.

As this policy operates as a default-deny basis, every other communication attempt is blocked.

6.3.2 Istio Authorization Policy

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-same-namespace
  namespace: {{ .Release.Namespace }}
spec:
  action: ALLOW
  rules:
    - from:
        - source:
            namespaces: ['istio-system']
    - from:
        - source:
            namespaces: ['{{ .Release.Namespace }}']
    - from:
        - source:
            namespaces: ['common']
```

```
- from:
    - source:
        principals:
        - cluster.local/ns/common/sa/istio-ingress
```

To improve isolation, we added a second layer of security at the Application level (OSI Layer 7) using an Istio Authorization Policy. This policy operates within the service mesh and doesn't strict only to network addresses but also identities (principals). This policy also acts as a default deny model, where any request not matching the defined rules is denied.

The first three rules allow traffic based on the source namespace alligning with the already permitted rules defined in the network policy. The final rule is more specific and allows traffic based on the source principal, which is the service account identity running the pod. This rule explicitly allows traffic from the istio-ingress service account which runs the ingress gateway responsible for routing external traffic.

By layering these two policies, we follow the defense in depth principle, improving our isolation objective through multiple isolation policies inforced at different layers of the OSI model. This multi-layered approach creates a secure and resilient multi-tenant system where isolation is enforced at both the infrastructure and application layers.

6.3.3 Media Data Isolation in Azure Blob Storage

Furthermore, tenant isolation goes beyond the kubernetes cluster level, with a critical service to isolate being the media asset storage. Media contents are stored in a separate azure cloud service — Azure Blob Storage. In order to ensure that data is segregated, our strategy was to separate azure blob containers with scoped credentials for each tenant.

As stated in the previous section, this process is fully automated at the time of tenant creation using an argo workflow. When a new tenant is onboarded, the workflow creates a unique blob container for each tenant within the same storage account, as highlighted in Figure 12. Then a SAS token is generated, which is scoped exclusively to the tenant's specific container. It cannot be used to list, read, or modify content in any other tenant's container, effectively locking down access to a single tenant's data.

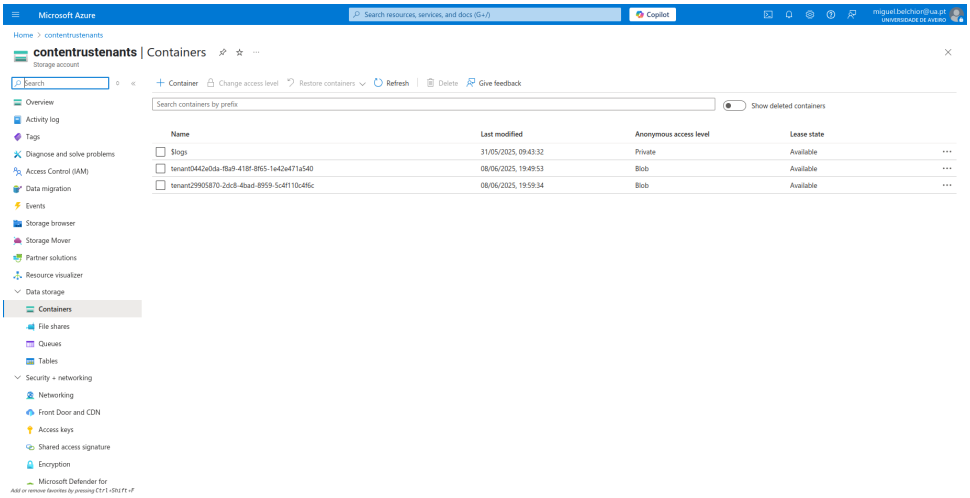

Figure 12: Per-tenant Azure Blob Storage containers in Azure Blob Storage Service

After that, the workflow creates a kubernetes secret with the azure credentials within the tenant's dedicated namespace, meaning that only the pods running inside that tenant's namespace have the ability to mount and use these credentials.

This final step links the external storage security back to the in-cluster isolation model, preventing Tenant A's application from ever accessing the credentials needed to connect to Tenant B's blob container, ensuring data confidentiality is maintained.

6.4 Observability

In modern software systems, observability has become a critical aspect of maintaining system health, performance, and reliability. Observability refers to the ability to measure the internal state of a system

based on the data it produces, such as logs, metrics, and traces. It enables developers, operators, and DevOps teams to detect issues, understand root causes, and optimize system behavior.

This section outlines the implementation of observability within our system, detailing the technologies employed and their respective roles in monitoring, tracing, and ensuring system reliability.

To implement observability in our project, we used OpenTelemetry in conjunction with Grafana, Jaeger, and Kiali (for the service mesh). This setup enabled us to achieve tenant-level observability, allowing for detailed monitoring and tracing per tenant.

We've deployed a single OpenTelemetry Collector in the common namespace to centralize telemetry data collection from all tenant namespaces. It receives telemetry data (metrics and traces) from each tenant, forwarding the processed metrics to Prometheus and traces to Jaeger for storage and visualization.

To achieve per-tenant observability, we configured the OpenTelemetry Collector to extract and attach Kubernetes metadata to telemetry data. Specifically, we use the k8sattributes processor in the Collector's pipeline to add metadata like the namespace as custom tags. Because each tenant has his own namespace, we can isolate each tenant's telemetry data in jaeger and grafana by filtering using each tenat unique namespace.

As part of our observability stack, we integrated Kiali to provide service mesh observability, leveraging Istio to visualize service-to-service communication within and across namespaces.

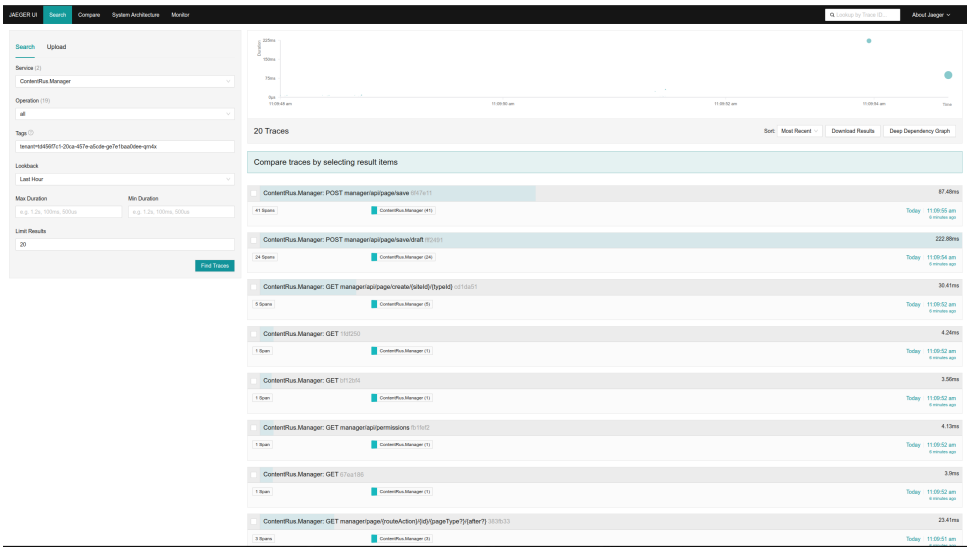

Figure 13: Jaeger Traces

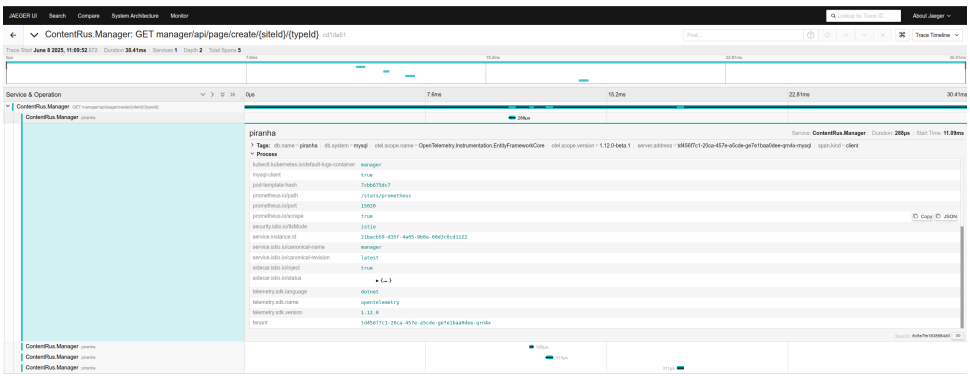

Figure 14: Jaeger Traces for GET endpoint

Figures 13 and 14 illustrate the process of filtering traces by tenant in Jaeger. As shown in Figure 14, each trace includes a tenant tag, derived from the k8s.namespace.name metadata added by the OpenTelemetry Collector's k8sattributes processor, which uniquely identifies each tenant.

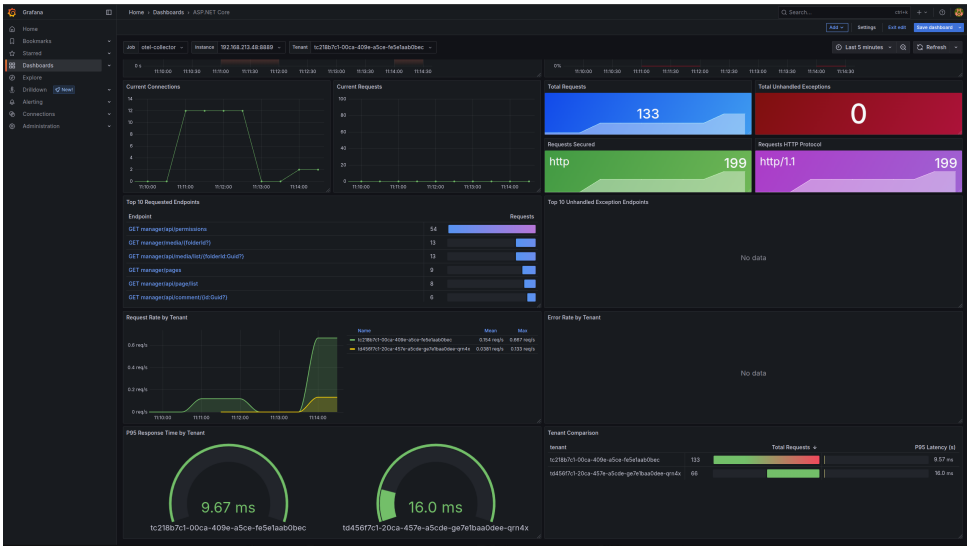

Figure 15: Dashboard with tenant specific metrics

Figure 16: Dashboard with tenant specific metrics

In Figure 15, we can observe tenant-specific metrics, including the top requested endpoints and a comparison of total requests and request latency across each tenant. A dropdown menu at the top enables selection of a specific tenant. Similarly, Figure 16, provides us with some tenant-specific metrics with the same namespace-based filtering capability.

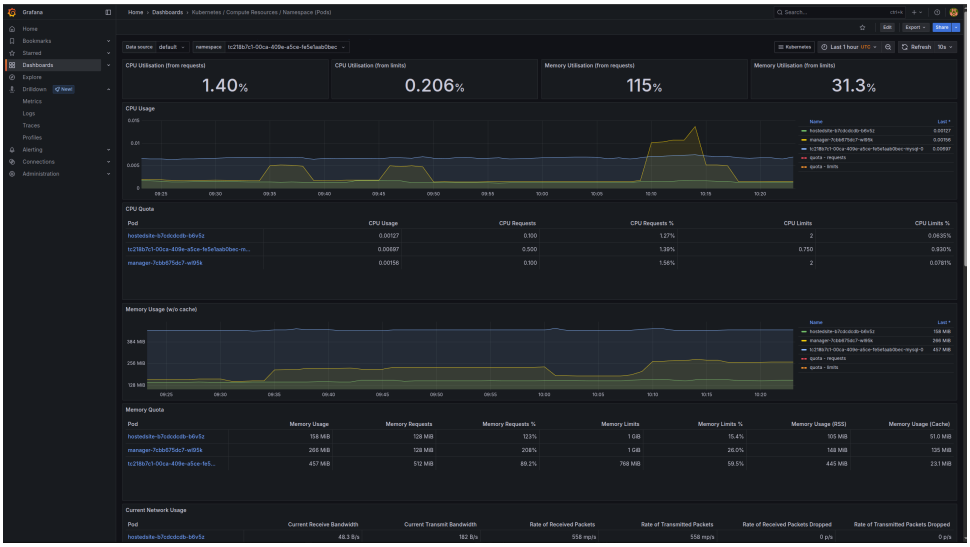

Figure 17: Grafana Dashbaord with resource usage per tenant

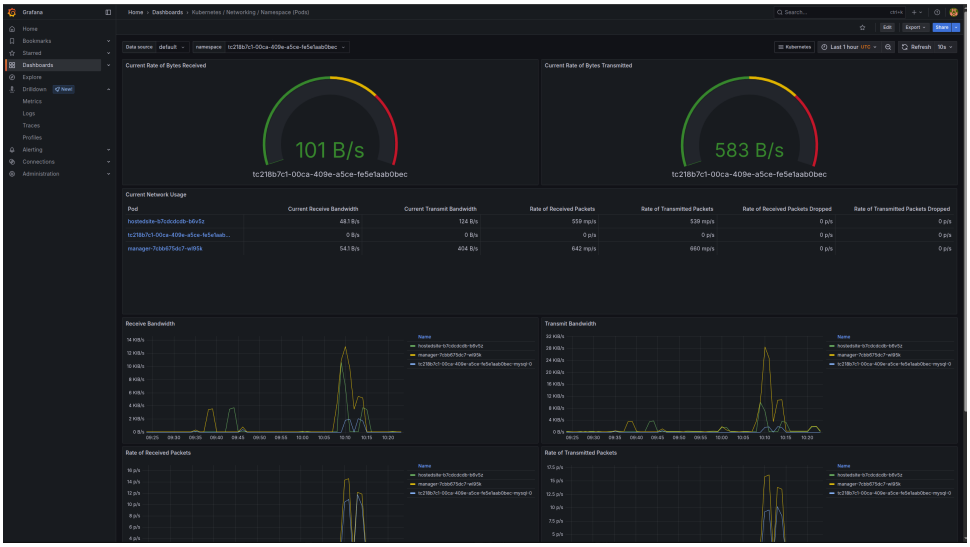

Figure 18: Grafana Dashbaord with networking usage per tenant

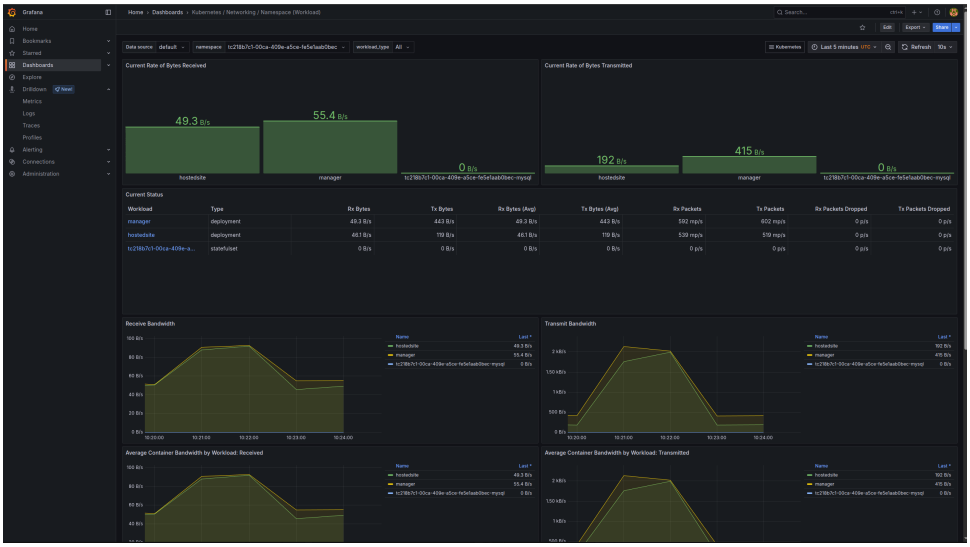

Figure 19: Grafana Dashbaord with resource usage per tenant

Figure 20: Grafana Dashbaord with resource usage in control namespace

Figures 17, 18, 20 and 19 provide some useful metris about the resources and networking of services within the cluster.

By observing Figure 17, we can observe some important metrics like the cpu and memory usage and limits by each pod (manager, hostedsite, and mysql database). This allows us to monitor each tenant resource consumption, check if they are approaching their CPU or memory limits and possible bottlenecks.

Figures 18 and 19 showcase the bandwidth received and transmitted from each pod in each tenant namespace. These metrics provide insights into network performance, allowing us to monitor data transfer rates and identify potential network congestion or anomalies specific to a tenant. Similarly, Figure 20 displays the same bandwidth metrics but for the control namespace, where services for self-provisioning are deployed.

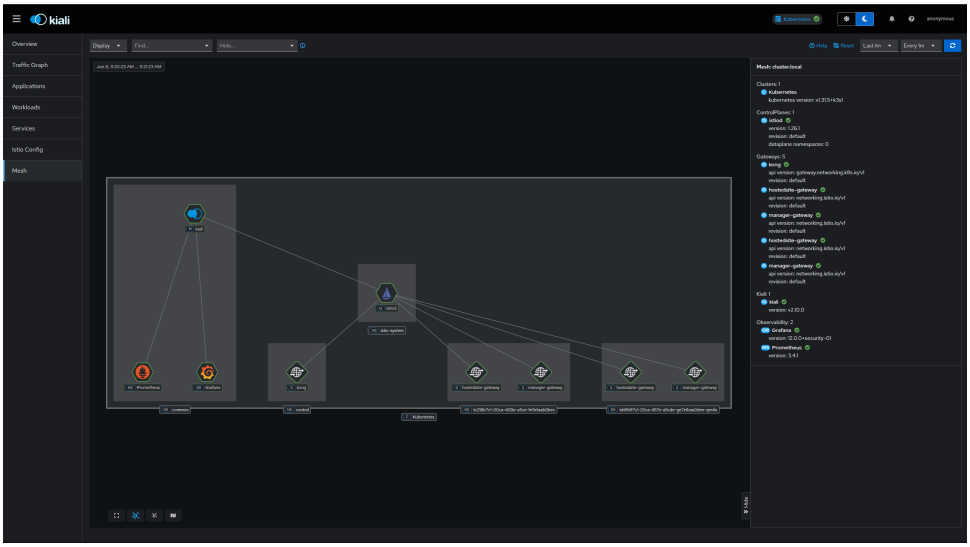

Figure 21: Kiali

In Figure 21 we can observe the service mesh with Kiali. The graph view highlights the interactions between various components, including the Istio system, Prometheus, Grafana, and tenant-specific services — hosted-site-gateway and manager-gateway.

6.5 Full Provisioning Flow

As it is important to systematically document the implemented full onboarding flow of a tenant's environment the diagram showcased in Figuere 22 was created.

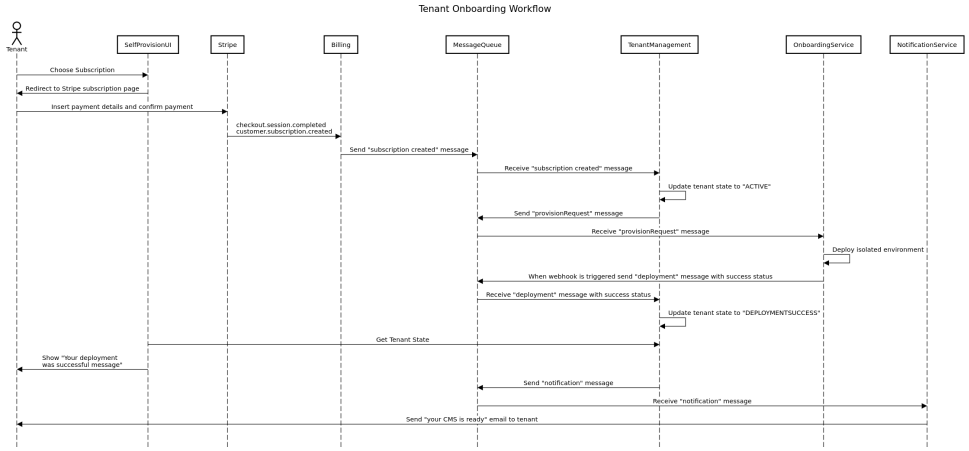

Figure 22: Onboarding Flow Sequence Diagram

Initially, the tenant interacts with the self-provisioning interface to choose a subscription plan. Once selected, the user is redirected to a Stripe-hosted page to complete the payment. After confirming the payment, Stripe triggers the checkout.session.completed and customer.subscription.created events, which are handled by the billing service webhook. This service then publishes a "subscription created" message to the message queue.

The tenant management service consumes this message, updates the tenant state to "ACTIVE", and emits a "provisionRequest" message, which is received by the onboarding service. This service is responsible for deploying an isolated environment for the tenant, which is triggered through the C# kubernetes client. Once the deployment completes, Argo CD triggers a webhook to notify the onboarding service of the deployment status.

Upon receiving this webhook, the onboarding service publishes a "deployment" message with a success status to the message queue. The tenant management service consumes this message, updates the

tenant's status to "DEPLOYMENTSUCCESS", and emits a "notification" message. Finally, the notification service consumes this message and sends an email to the tenant indicating that their CMS environment is ready. Simultaneously, the self-provisioning interface reflects the updated status and displays a confirmation message to the tenant.

7 Work Completed After Presentation & Demo

This section highlights the functionalities that were implemented after the final presentation, along with evidence demonstrating that they were in fact developed and are working as intended.

7.1 Documentation

The project's reports constitute the documentation for the architectural decision and important implementation details. Furthermore, we've also documented our APIs using swagger, as showcased in Figure 23 for the Tenant Management Service.

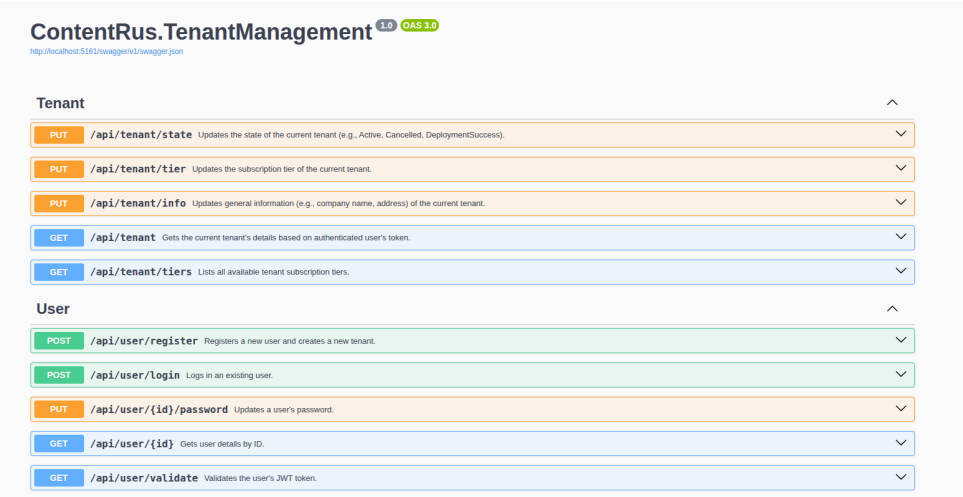

Figure 23: Swagger with API Documentation for the Tenant Management Service

7.2 Network and authorization policies

As mentioned in Section 6.3, both a network and an authentication policy have been created in order to isolate each tenant namespace and prevent unauthorized access from other tenants.

7.3 Tenant Provisioning Webhook and Email Notification

As previously described, when the provisioning is successful, Argo CD sends a POST request to a webhook exposed by the onboarding service. This service then publishes a notification event to the message queue, which is consumed by the tenant management service. After updating the tenant's status accordingly, the tenant management service emits a new message to the notifications service, which in turn sends an email to inform the tenant that their environment has been successfully provisioned. Figure 24 shows the email that is sent by the notifications service.

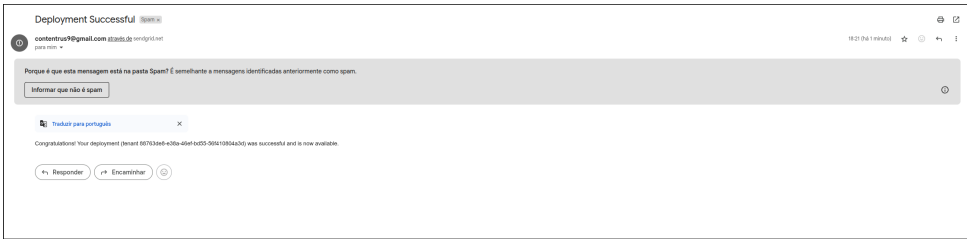

Figure 24: Email sent by notification service

Furthermore, as the tenant's state is now updated, the self provision interface is refreshed to inform the user of the successfull tenant environment deployment status with hyperlinks for the manager and hosted sites, which is showcased in Figure 25.

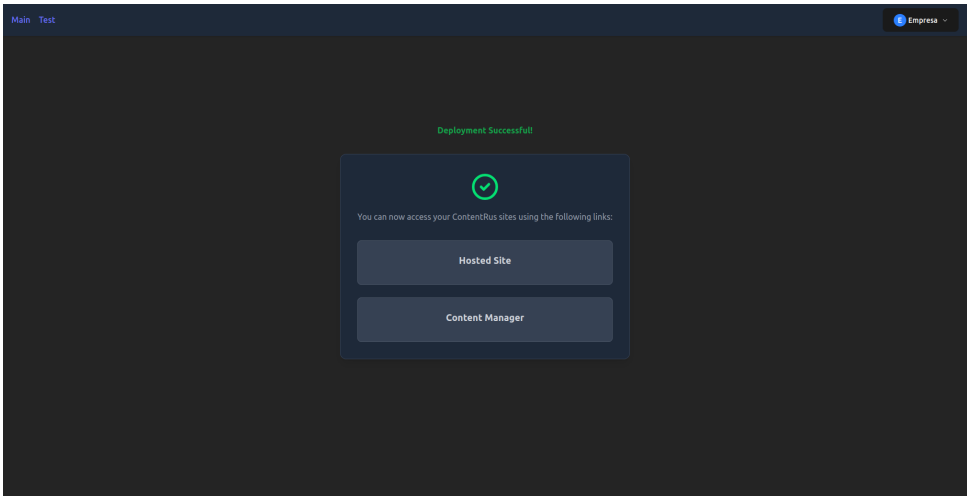

Figure 25: Tenant Environment Deployment successful page

8 Future Work

Future work will focus on enhancing security and scalability to support continued growth and operational robustness. One key area is the integration of a dedicated key management system, such as HashiCorp Vault, for tenant credential management. This would replace the process of simply having credentials stored as in-cluster secrets with a more mature and secure solution, enabling better secrets lifecycle management, access control, and auditability.

Another important improvement involves implementing Horizontal Pod Autoscaling (HPA) to dynamically scale resources based on each tenant's workload. The existing pipeline and configuration architecture is already designed to support multi-tenant scaling, and only minimal script adjustments are needed to enable HPA for each tenant environment. These enhancements aim to improve system resilience, security, and performance in a production-ready setup.

9 Conclusion

This report detailed the transformation of PiranhaCMS into a robust, scalable, and secure multi-tenancy SaaS platform. Through the application of Attribute-Driven Design, the project addressed the initial limitations of Piranha CMS, achieving strict tenant isolation, automated onboarding, and per-tenant observability.

The adoption of a microservices architecture, enhanced by a service mesh with Istio, and the implementation of a GitOps-based Continuous Deployment pipeline with Argo Workflows and ArgoCD, ensured a resilient and maintainable system. The Agile/Scrum methodology facilitated iterative development, enabling the team to adapt to challenges such as tight coupling in the application plane, ultimately integrating authentication and media services into the Manager microservice for improved maintainability.

Key features, including tenant-specific namespaces, network and authorization policies, and integration with Azure Blob Storage, fulfilled the strategic goals of usability, isolation, and performance. Post-demo enhancements, such as refined provisioning webhooks and email notifications, further validated the platform's readiness. This project lays a solid foundation for future improvements, including advanced security with HashiCorp Vault and dynamic scaling with Horizontal Pod Autoscaling, positioning ContentRus as a viable multi-tenant SaaS solution.

References

[1] T. Golding, *Building Multi-Tenant SaaS Architectures*. O'Reilly Media, Inc., April 2024. [Online]. Available: https://learning.oreilly.com/library/view/building-multi-tenant-saas/9781098140632/

[2] ContentRus, "Argo workflow template — tenant provisioning with credentials," https://github.com/Migas77/contentrus.multiTenancy/blob/master/infrastructure/argo/argowf/tenant-provisioning-with-credentials-template.yml, 2025, accessed: 2025-06-08.

[3] ——, "Argocd application set for tenant provisioning," https://github.com/Migas77/contentrus.multiTenancy/blob/master/infrastructure/argo/argocd/tenants-application-set.yaml, 2025, accessed: 2025-06-08.

[4] ——, "Tenant provisioning argocd notification," https://github.com/Migas77/contentrus.multiTenancy/blob/master/infrastructure/argo/argocd/deploy-status-notification.yaml, 2025, accessed: 2025-06-08.