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

May 9, 2025

1 Introduction

This report provides an in-depth analysis and design strategy for extending Piranha CMS to support a multi-tenancy SaaS model. It outlines an analysis of the current Piranha CMS system and its limitations and the vision and strategic goals of this assignment and scenario. Domain Driven Design is utilised to model the Business Domain into core, generic and supporting subdomains. An Attribute Driven Design architectural methodology is 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 addresses 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 emphasizes the prioritization of key features essential to the envisioned multi-tenant system and outlines a clear development roadmap to guide its implementation for the next month.

Before dwelving into the content of this report first we must define the concept of multi tenancy in a SaaS environment. The Multi-tenancy concept is generally used to convey the idea that some resource (compute, storage, and other resources) is shared by multiple tenants, with each tenant's data being isolated from, and invisible to, the other tenants sharing the application instance, ensuring data security and privacy for all tenants. In fact this is a perfectly valid notion of multi-tenancy.

However, "using the literal shared infrastructure definition of multi-tenancy doesn't seem to map well to the various models that can be used to deploy tenant infrastructure" [1]. For instance, in a full-stack silo model, each tenant may have dedicated microservice instances, meaning there's no resource being shared at the service level. Yet, these tenants are still onboarded, managed, and operated through a common system and interface. Therefore, throughout this report, multi-tenancy will be defined in a broader sense and "will refer to any environment that onboards, deploys, manages, and operates tenants through a single, unified experience" [1].

2 System Analysis & Project Vision

2.1 System Analysis

The standard Piranha CMS architecture is primarily designed for a single-instance, not a multi-tenant SaaS model. However, the authors of Piranha CMS highlight its support for "multi-tenancy", in the sense that it's possible to set up a single Piranha CMS instance serving multiple websites at the same time. This definition and feature, although usefull to the overall goal of this project, does not align with

the previously defined concept of multi-tenancy coming with the following limitations (outlined by the developers in their documentation [2]):

- **You cannot restrict content types per site** - This means that if you had a single Piranha CMS site with two companies (Company A and Company B) then both companies would share the same content types.
- **You cannot restrict users to a single site** - This means that you cannot have a user who is only able to edit one site, providing no real isolation capabilities.
- **You cannot restrict sites to a single media library** - This means that all sites will share the same images, providing no real isolation capabilities.

Therefore, **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. Moreover, there are other gaps in the Piranha CMS that further outline the divergence of the multi-tenant scenario:

- **User Management**: User management is global to the CMS and does not inherently support the concept of tenant-level users.
- **Tenant Management & Onboarding**: As there is no conecpt of tenants, there is no built-in functionality for automated tenant provisioning, lifecycle management (suspension, deletion), or tenant-specific configuration. Onboarding would currently be a manual process of setting up new sites or instances in separate infrastructure.
- **Data Isolation**: While sites might separate content logically, true multi-tenancy requires strict guarantees that one tenant cannot access another's data (users, content, media, etc.). One example of this is related to Piranha typically using a single database per instance, lacking built-in mechanisms for strict data partitioning within that database or across databases managed by a single application instance.
- **Observability & Resource Management**: PiranhaCMS doesn't provide mechanisms to meter manage or resource consumption on a site, user or per-tenant basis, which is important for a SaaS offering, specially one emphasizing multi-tenancy.
- **Monolithic Architecture & Scalability Concerns** : Although the PiranhaCMS provides a plugabble and extensible architecture, the architecture of the solution is monolithic running within a single process. This design can limit scalability and flexibility, particularly in high-traffic scenarios or complex enterprise environments. Scaling would involve replicating the entire application instance, which can lead to inefficient resource usage and difficulty in isolating or upgrading individual services. As demand grows, adopting a more modular or microservices-based architecture may be necessary to achieve better scalability, maintainability, and deployment agility. This becomes even more important as we need to scale in a per-tenant basis (according to the deployment model).

2.2 Project Vision and Strategic Goals

2.2.1 Vision

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).

2.2.2 Strategic Goals

To fulfill this vision, the platform is guided by the following strategic goals:

- **Achieve strict data isolation:** Guarantee that all tenant-specific data (content, users, configurations, logs, etc.) is logically and technically separated, ensuring both security and compliance requirements (e.g., GDPR, HIPAA).

- **Simplify tenant onboarding:** Streamline the onboarding process to support an automated and rapid provisioning of new tenants, including a self-service signup and onboarding flow which triggers pre-designed tenant initialization/provisioning scripts and configurations, promoting consistency across the multiple tenants.
- **Ensure horizontal scalability:** Architect the system to scale horizontally, supporting a growing number of tenants with minimal performance impact. This includes stateless service design, efficient load balancing, and tenant-aware caching mechanisms.
- **Support tenant-level customization:** Allow individual tenants to configure themes, roles, features, and policies without affecting other tenants. This improves adoption by meeting specific client needs.
- **Monitoring and analytics:** Maintain a comprehensive monitoring system tha provides both a global and tenant-aware view of the multi-tenant environment. This should include real-time tracking of the system health and resource consumptions, as well as usage patterns in a per-tenant basis, in order to provide insights on preventing the noisy neighbor problem. Moreover, it should also provide alerting mechanisms for identifying breaches of isolation in the multi-tenant environment, in the eventuality that they occur.
- **Maintain high availability and fault tolerance:** design the system with redundancy and failover capabilities to ensure continuous service delivery, even in case of node or service failures.

2.2.3 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. This refers to deployment-time and runtime isolations mechanisms:
  - Deployment-time isolation — apply your isolation policies when your resources are deployed and configured, usually through DevOps practices. For instance, in a fully stack silo model, where we have separate microservices and databases for each tenant in separate environments, an isolation policy has to be attached to the compute of each microservice that prevents that microservice from accessing any resources that belong to other tenants.
  - Runtime Isolation — For pooled models with shared resources (for instance, a common database) it's necessary to, at runtime, utilize the tenant context to populate a policy and acquire tenant-scoped credentials that will be used to access downstream resources.

  Furtermore, given that different clients/tenants may have different compliance obligations this isolation requirements becomes more significant.
- **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:
  - Role-Based Access Control (RBAC) mechanims, where permission claims (such as read, write or delete of certain resources) are assigned to roles, which are then assigned to users. Thus, the user's access to resources is determined by the roles assigned to it, following the Least Privilege Principle. This is already implemented at the application level by the existing Piranha CMS solution.
  - Access to the API and application instances must not only follow Role-Based Access Control (RBAC), as described previously, but must also be tenant-aware, which may be done at the application instance or in other component of the architecture depending on the deployment model. Tenant access to storage instances and databases must also be part of this debate.
- **Scalability:** The platform must scale horizontally to support an increasing number of tenants and user traffic without impacting performance, employing strategies, such as a Stateless service design for easy replication, with distributed caching mechanisms. Automatic Load Balancing and auto-scaling infrastructure (provided by standards such as Kubernetes) is also an important concern.

- **(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. Redirect to Tenant Admin Interface — Once provisioning is complete, the tenant admin user is automatically redirected to the tenant-specific admin dashboard, where they can analyse and possibly configure their environment.

- **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). In order to achieve this, aspects such as autoscaling policies, caching, asynchronous task processing and performance and resource consumption monitoring/alerting must be considered.

- **Reliability:** As a SaaS offering, the system must ensure high availability and fault tolerance, obeying to the following considerations:

  – Minimize single points of failure.
  – Incorporate health checks and failover mechanisms.
  – Implement graceful degradation strategies under partial failure.

- **Maintainability:** The architecture of the system must consider its ability to be easily modified, extended, and repaired with minimal effort and downtime. This can be achieved through best practices, such as infrastructure-as-code — specially important in this multi-tenant deployment scenario — version control, documentation, etc. Given that Piranha CMS is an existing open source project, further consideration is necessary about whether to adhere/contribute to their project and practices or to fork the project due to incompatibilities of the envisioned multi-tenancy solution when compared to the original.

- **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.

- **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.

3 Choosing an Architectural Design Methodology

The selected architectural design methodology that will be applied for the multi-tenant scenario is 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. While the Architecture-Centric Design Method (ACDM) also offers a structured and intuitive approach, it is more process-oriented and lacks the explicit prioritization of these key attributes.

ADD, however, begins by identifying the most important quality attributes for the system and uses them as the basis for all architectural decisions. This ensures that design choices are not made solely based on functionality or intuition but are instead directly aligned with the system's critical quality

goals. This attribute-first focus makes ADD particularly well-suited for evolving an existing system like Piranha CMS, where new requirements, quality attributes, and constraints must be identified and integrated into the architecture iteratively, in the goal of achieving multi-tenancy.

Despite its robustness as an enterprise architecture framework, TOGAF Architecture Development Method (ADM) was not picked as our architectural design methodology. TOGAF ADM is intended for large-scale enterprise contexts and introduces a level of complexity, coordination, and documentation that is disproportionate to the needs of this project. In this context, it would have introduced unnecessary overhead and slowed down our progress, rather than offering meaningful architectural clarity. Therefore, ADD emerged as the most appropriate choice for our objectives. The methodoly steps of the ADD approach are outlined in Figure 1.

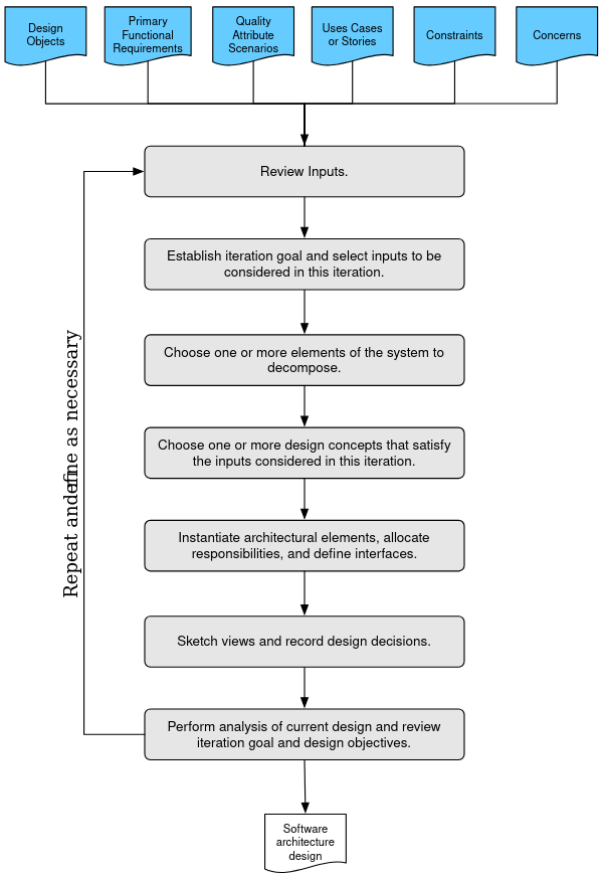

Figure 1: Attribute Driven Design (ADD) Methodology

The selected Attribute Driven Design (ADD) Methodology will be applied in Section 5, following the identification of the core domains of the multi-tenant system.

4 Identify and Model Core Domains — Domain Driven Design

Domain-Driven Design (DDD) is a software design philosophy that emphasizes the importance of understanding and modeling the business domain into subdomains, categorized into three different categories: core subdomains, generic subdomains and supporting subdomains. Firstly, we must identify the domain of PiranhaCMS. Piranha CMS operates in the business domain of digital content management (CMS — Content Management System), enabling organizations to create, organize, and deliver web content efficiently, without needing deep technical expertise. The platform supports multi-site, multi-language publishing for businesses targeting diverse markets. With this business context established, we can now begin identifying and categorizing the subdomains of Piranha CMS:

4.1 Core Subdomains

These are the activities the company is implementing and performing differently from its competitors and from which it gains its competitive advantage:

- **Tenant Management** — This is the absolute core of the multi-tenant scenario. It contains all the information about a tenant (company name, identity configuration, tier...), as well as the lifecycle of a tenant. The lifecycle of a tenant might include the following states: TENANT_CREATED, TENANT_PROVISIONED, BILLING_INITIALIZED, USER_CREATED, and TENANT_ACTIVATED.
- **Tenant Provisioning** — Corresponds to the provisioning of any tenant resources that are required, possibly depending on the tenant tier and deployment model (e.g. full stack silo model, full stack pool model, mixed mode model, etc.). This includes aspects such as adding new tenant routing entries, provisioning compute and/or storage units (depending on the deployment model), among others.
- **Onboarding** — Responsible for the orchestration and the full lifecycle of the onboarding process, managing and ensuring that all steps in the process are completed successfully.
- **Content Management (Core CMS Functionality)** — Although not as relevant in the multi-tenancy process as the previously mentioned subdomains, we consider it is still a critical part of the platform and one of the deciding factors in user adoption. Therefore, we decided to include it in the core subdomains.

4.2 Generic Subdomains

These are areas common to many software systems or CMS platforms. While necessary, they do not represent a unique competitive advantage for Contents'r'us in the multi-tenant context:

- **Billing** — Related to managing tenant-specific subscriptions. This will require adoption of established pre-existing payment providers.
- **User Management** — The pre-existing system already provides user management features with authentication capabilities through ASP.NET Core Identity and claims-based authorization. The authorization process follows a Role-Based Access Control (RBAC) mechanism at the application level, where a set of permissions are assigned to user roles, which are then assigned to the users.

4.3 Supporting Subdomains

These domains support the core business and are typically simpler with obvious solutions, not directly providing any competitive advantage:

- **Media Storage** — Handles the storage, retrieval, and delivery of media assets such as images, videos (static assets), etc. This will rely on standard, well-established cloud storage solutions — in this case, Azure Blob Storage as it's already implemented by the framework — and integrates with content delivery networks (CDNs) for scalability and performance. This aspect gains importance as we discuss multi-tenancy due to efficient, secure, and isolated handling of tenant-specific media assets, ensuring data segregation and access control mechanisms (e.g. bucket policies).
- **Notifications** — Assist core business functionalities by facilitating notifying customers of events, such as a completed onboarding process. These notifications will be implemented using off-the-shelf external services.

5 Applying Attribute-Driven Design — ADD

This section will focus on applying the ADD methodology for the multi-tenancy scenarios. This is an iterative process and therefore various iterations of the approach will be outline throughout this report.

5.1 Review Inputs

According to Figure 1, the inputs necessary for this step of the ADD methodology are as follows: Design Objectives — high-level goals or desired outcomes for the system's design, guiding the overall architecture and design decisions; Primary Functional Requirements — core functionalities the system must provide; Quality Attribute Scenarios — specific requirements related to the system, often addressing non-funcional aspects; Constraints — limitations or restrictions on the design, such as time, budget, technology stack, or regulatory requirements; Architectural Concerns — key architectural issues or challenges that need attention; Existing Architecture – the current system's architecture as initially developed for the Piranha CMS.

In practice, the previous list of high-level inputs maps into the following specific inputs:

- **User Stories**
  - **System Admin Dashboard - Multi-tenant / Per-tenant View**:
    As a system administrator, **I want** to access a comprehensive dashboard that provides both system-wide and per-tenant analytics, **so that** I can identify anomalies, optimize resource usage, and maintain operational efficiency across all tenant environments.
  - **Tenant Admin Dashboard - Per-tenant View**:
    As a tenant admin who has already provisioned its environment, **I want** to access a comprehensive dashboard that provides both system-wide and per-tenant analytics, **so that** I can monitor usage metrics, track performance and identify anomalies.
  - **Disabling a Tenant from System Admin Dashboard**:
    As a system administrator, **I want** to temporarily disable a tenant, **so that** the tenant's environment becomes inaccessible without deleting or altering any of its data and configurations.
  - **Tenant Self-Provisioning**:
    As a new user, **I want** to sign-up, choose a subscription plan, provide billing information and create my own tenant environment automatedly **so that** I can start using the CMS plaftorm autonomously.
  - **Notify Tenant Admin when Tenant Environment is Ready**:
    As a tenant admin who has readly completed self-provisioning, **I want** to be notified when my tenant environment is ready, **so that** I can begin working with the CMS platform.
  - **Disabling a Tenant Automatically When Billing Becomes Invalid**:
    As a system administrator, **I want** the platform to automatically disable a tenant when their billing information is/becomes invalid, **so that** we can prevent unpaid usage and encourage timely billing updates.
  - **Tenant Lifecycle Management Dashboard**:
    As a system administrator, **I want** to access a dashboard to track the tenant lifecycle, **so that** I can manage tenant states and transitions effectively.
- **Functional Requirements**
  - **FR1** The system shall provide a System Admin Dashboard for system administrators to view a list of all tenants and filter them individually, showcasing relevant tenant information such as plan type, tenant state (active, disabled, provisioning) and last activity date.
  - **FR2** The system shall provide a System Admin Dashboard for system administrators to view both system-wide and per-tenant analytics of the environment at real time, including resource usage metrics and usage patterns.
  - **FR3** The system shall provide a Tenant Admin Dashboard for tenant administrators that displays per-tenant analytics including usage metrics and performance analytics.
  - **FR4** The system shall allow system administrators to temporarily disable a tenant environment from the System Admin Dashboard, ensuring data and configurations remains intact during the disabled state, for a given period of time (grace period).
  - **FR5** The system shall provide a self-provisioning interface from where the users (soon to be tenants) can configure their environments according to the following order:
    1. Register an account.

2. Select a subscription plan.
3. Enter organization and billing details.
4. Trigger automatic provisioning of a new tenant environment upon successful payment.
5. Redirect to Tenant Admin Dashboard.

- **FR6** The system shal automatically disalbe a tenant, when their billing information is/becomes invalid. It should also apply a grace period as mentioned in **FR4**.
- **FR7** The system shall provide, at the Tenant Admin Dashboard, a temporary page informing users that their environment is being prepared until provisioning completes.
- **FR8** The system shall send a notification (email) to the tenant administrator once the tenant environment is successfully provisioned.

- Quality Attribute Requirements — The specification of the quality attributes was already outlined in Section 2.2.3. We prioritized these attributes based not only on their overall importance to multi-tenancy, but also on the ease with which we could iteratively address them, considering our limited experience in designing the architecture of a multi-tenanct solution:

  1. Separation of Concerns
  2. Isolation
  3. Scalability & Performance

- Constraints — In designing the architecture for our multi-tenancy solution, we identified the following constraint:

  1. **Budget** — It's common for SaaS multi-tenant solutions to move towards the cloud and serverless solutions. However, due to budget limitations, we were unable to leverage cloud providers or serverless architectures, using services such as AWS or Azure. As a result, (and as you'll later see) we opted for a containerized deployment model orquestrated with Kubernetes, where we limit the use of Azure Cloud for what was strictly necessary and what would save us time upon implementation (already implemented): Azure Blob Storage.

- Existing Modules — As the open-source Piranha CMS does not provide a clear description nor diagram of its architecture, we analysed the available documentation, code structure, and its key components and interactions. The implemented solution follows a monolithic architecture that couples all business functions in one codebase or application. The following diagram (Figure 2) illustrates the most important modules/services of the monolithic system that will be taken into account as the ADD methodology is applied:

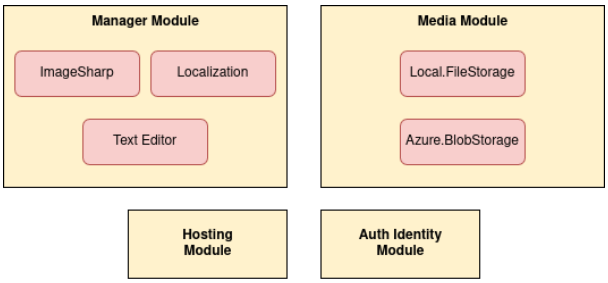

Figure 2: Piranha CMS Modules

The Manager Module handles core administrative and editing functionalities, with subcomponents for image processing, localization for translation and editing text being highlighted. The Media Module is responsible for storing static assets either locally on the filesystem or using a cloud solution, which in this case is Azure Blob Storage. The Hosting Module is responsible for managing the service aspects of the websites and runtime hosting capabilities. The Auth Identity Module is responsible for user authentication and identity management, including login, roles, permissions, and user profiles. The authorization process follows a Role-Based Access Control (RBAC) mechanism at the application level, where a set of permissions are assigned to user roles, which are then assigned to the users.

5.2 First Iteration

5.2.1 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 the **Separation of Concerns**, ensuring that the system is segregated in a way that clearly separates different responsabilities into different components. This requirement is closely coupled and contributes to improve scalability/performance, as by, separating different responsabilities to different components it's possible to apply autoscalling rules to target components, optimizing resource usage and avoiding unnecessary overprovisioning.

5.2.2 Choose elements of the system to decompose

The default Piranha CMS showcases a monlithic architecture, where all functionalities run within the same process. The **full system will undergo a major refactor** to introduce separation of concerns, allowing different responsibilities to be isolated into distinct components or services, which improves other non-functional requirements such as scalability, maintainability, and deployment flexibility.

5.2.3 Choose Design Concepts that satisfy the iteration inputs

In order to satify the iteration goal and the input architectural drivers the following design concepts can be used:

- a **microservices architecture** with an API-First design for communication between microservices and, the microservices and the UI.
- employing a **asynchronous messaging system for asynchronous communication between microservices**. This is specially important due to dependencies to external services, which may not be always available. Moreover, it will provide a robust mechanism for decoupling services and handling communication at scale, allowing for better fault tolerance and system resilience.

5.2.4 Instantiate Architectural Elements, allocating responsabilities and defining interfaces

In this first iteration focusing on **Separation of Concerns**, we instantiate the architectural elements by decomposing the monolithic Piranha CMS into distinct microservices with clear responsibilities and well-defined interfaces.

Architectural Elements

The monolithic Piranha CMS is restructured into two distinct planes to separate administrative and operational functions. The Control Plane oversees tenant management, provisioning and administrative tasks, while the Application Plane handles tenant-specific services. The elements are organized as follows:

- **Control Plane:** Includes Tenant Admin Auth, Billing, Onboarding, Notification Service, Tenant Management, Metrics, and API Gateway, as well as two dasboards for admin and self-provisioning purposes, respectively. The Control Plane also includes a Message Queue.
- **Application Plane:** Comprises Auth, Hosted Site, Manager, Media, and API Gateway.

This division establishes a clear separation of concerns, allowing independent scaling and development of each plane.

Responsibilities

The following section describes the responsibilities allocated to each service to meet functional and non-functional requirements.

- Control Plane Services

- **Tenant Admin Auth:** Authenticates tenant administrators, ensuring secure access to management functions.
- **Billing:** Integrates with a Billing External Provider to 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 tenant environment.
- **API Gateway:** The API Gateway serves as a central access point routing requests between admin interfaces and backend microservices while mediating communication between services.

- Application Plane Services

- **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, integrating with external storage solutions.
- **API Gateway:** It's similar to the API Gateway from the Control Plane Services. However, it also routes tenant-specific requests to the appropriate services, ensuring seamless interaction.

Thus, the identified services segregated by the outlined service responsibilities support a modular and tenant-aware architecture.

Interfaces

The following class diagram 3 outlines the interfaces of the control plane services to which we previously attributed responsabilities. We didn't include such diagram for the Application Plane has this functionality is already implemented by the default Piranha CMS. The diagram includes a general high level view of the actual functionalities of each component/class. Bear in mind that the API Gateway and message queue are not present in the diagram and therefore, the interactions between microservices appear as being point to point (not actually true). The metrics microservice will also not be included. We also didn't include specification of the methods of the interfaces/dashboards.

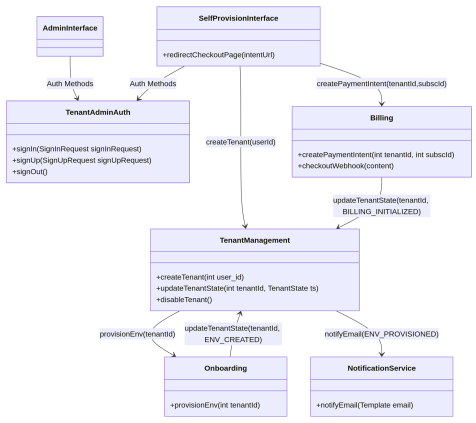

Figure 3: Class Diagram — Control Plane Services

5.2.5 Sketch Views and Record Design Decisions

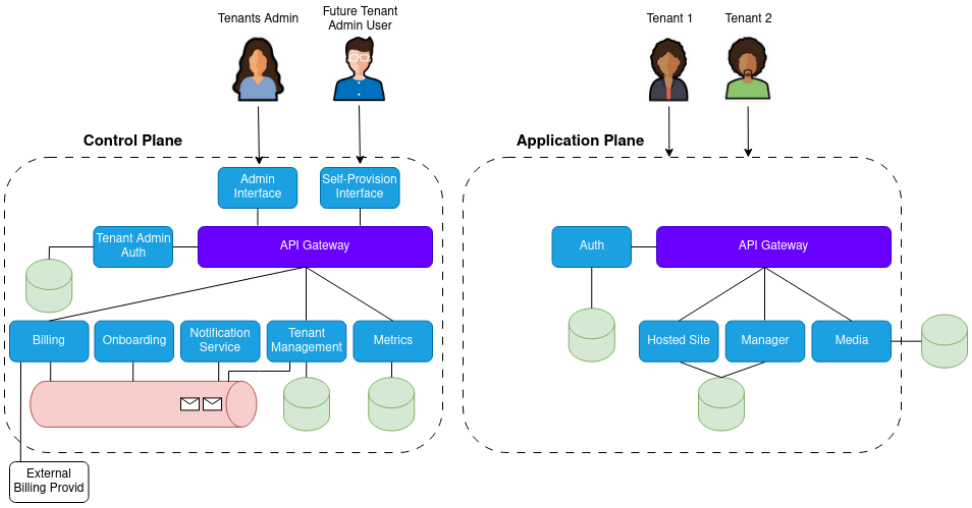

Figure 4: Architecture Diagram — Step 5 Iteration 1

Figure 4 showcases the designed solution at this step. At the diagram we can clearly observe the existance of two different planes and sets of services according to the predefined responsabilities. In both planes, backend microservice communication is handled by the API-Gateway. In the case of the Control Plane, a message queue is included for asynchronous communication, which is regularly used to communicate tenant state changes from one service to another. Moreover, it's clear the existance of two different external services, namely an external billing provider on the control plane and a Media Blob Storage solution on the application plane.

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

- **ADR: Adopting a Microservices Architecture**

  Context — The existent Piranha CMS architecture is monolithic with all functionalities tightly coupled within a single process which limits scalability, as scaling requires replicating the entire application, leading to inefficient resource usage. It also restricts isolation, as tenant-specific data and operations are not strictly segregated.

  Decision — Adopt a microservices architecture, refactoring the Piranha CMS architecture into distinct, independently deployable services. This introduces modularity to the system, enabling clearer delineation of responsibilities between tenant management (Control Plane) and tenant-specific operations (Application Plane). Each microservice will have a single responsibility (e.g., Tenant Management, Auth, Media) and will communicate via REST APIs or a message queue.

  Consequences

  - Promotes separation of concerns by decomposing the monolithic system into modular services with clear responsibilities.
  - Achieves isolation by structuring services to operate within distinct planes, enabling tenant-scoped segregation.
  - Supports scalability by allowing individual services to be scaled independently based on demand, avoiding overprovisioning of the entire system.
  - Increases operational complexity due to the need to manage and coordinate multiple services.
  - Introduces communication overhead from inter-service interactions via REST APIs and message queues.
  - Poses challenges in ensuring data consistency across microservices, necessitating careful design of event-driven workflows via the message queue.
  - Reduces maintainability due to deviating from the original architecture of PiranhaCMS. Thus, contributions to this project will have to be made separately (fork of PiranhaCMS). However,

we strongly believe that the pre-existing architecture wouldn't allow for granular scalability, which is a core concern. (we will later disscuss this issue)

- **ADR: Using an API Gateway as an abstraction layer, which provides context for downward services**

  Context — The system is multi-tenant, with distinct control and application planes, following a microservices architecture. In the application plane, each authenticated request must be mapped into a tenant context for each of the downward services to operate correctly.

  Decision — Introduce an API Gateway as a single entry point for both the control and application planes. The gateway will handle authentication, enrich requests with the corresponding tenant context, and route them to the appropriate backend services.

  Consequences

  - Simplifies service implementation by centralizing responsibilities such as authentication and context handling to a single layer, improving consistency and maintainability.
  - Provides a centralized point for monitoring, rate limiting, and logging.
  - Introduces a single point of failure and potential performance bottleneck if not properly scaled and managed (this depends heavily on how and through which URL each tenant accesses the system — this will be further discussed later on a future iteration).
  - Adds operational complexity in terms of deployment and configuration.

- **ADR: Separate Authentication Services for Tenant Admin and Tenant Users**

  Context — In a multi-tenant environment following a microservices architecture, ensuring strict isolation between tenants is paramount to prevent cross tenant unauthorized access to data or resources. A decision must be made between having a single or separate authentication components for the tenant admins and the users of a tenant.

  Decision — Implement separate authentication services for the Control Plane (Tenant Admin Auth) and the Application Plane (Auth) to enforce strict tenant isolation. The Tenant Admin Auth service will authenticate tenant administrators, restricting their access to Control Plane functions (e.g., tenant management). The Auth service will authenticate tenant users, scoping their access exclusively to their tenant's context within the Application Plane (e.g., content editing, media management, etc.).

  Consequences

  - Enhances Isolation by ensuring tenant administrators and tenant users are authenticated through distinct services, preventing cross-tenant access and reducing changes of misconfiguration.
  - Supports compliance requirements by enforcing strict data and access segregation between tenants.
  - Increases operational complexity due to managing two separate authentication services, requiring distinct configurations and monitoring.
  - May introduce challenges regarding the creation of the first manager user of the CMS platform on the Application Plane (possibly solved through the use of a job). Bear in mind, this CMS manager user should not have any connection to the tenant admin user, apart from belonging to the same tenant (as well as all the other users of that same tenant).

- **ADR: Message Queue**

  Context — The system is multi-tenant, following a microservices architecture with an API Gateway for microservice-to-microservice communication. The Billing Service utilizes an external billing provider to achieve its business purposes, which may not always be available. Services also need to trigger asynchronous workflows for communicating changes of tenant state as well as environment provisioning. Depending on the operation retry logic may also be necessary.

  Decision — Introduce a Message Queue to support asynchronous communication between microservices. Services will publish events or messages to queues or topics, and subscribing services will consume and process these events independently. A retry and dead letter exchange may be

configured for retrying mechanisms. Retrying mechanisms are specially important as there are dependencies to external services and unavailability of those services corresponds to the unavailability of the onboarding process. The fault tolerant integration with an External Billing Provider and retrying logic can be showcased in the following image:

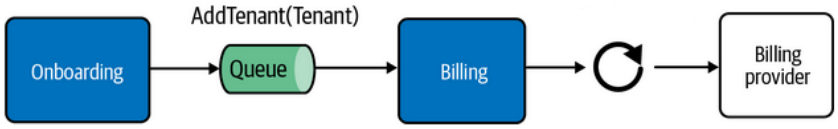

Figure 5: Fault Tolerant integration With an External Billing Provider

Consequences

- Fault Tolerance and Resilience: Services will be more resilient to failures of downstream dependencies. The retry logic and dead letter queues greatly enhance robustness, ensuring that temporary issues with external services do not bring the system to a halt.
- Operational Overhead: Monitoring, scaling, and maintaining the message broker infrastructure will add to DevOps responsibilities, including ensuring high availability and proper failover mechanisms.
- Observability Needs: Tracing asynchronous flows across services requires robust observability tools to ensure proper monitoring, logging, and alerting.
- Asynchronous Processing: Workflows that do not require immediate responses can be handled asynchronously, reducing coupling between services and improving scalability.

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

The achieved design in this iteration focused on achieving Separation of Concerns within the app. This was accomplished by clearly delineating responsibilities between 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. In this case, the control plane has adicional responsabilities regarding administrative tasks with an overall view of all the tenants and their resources. This could be further refactored into a separate environment, but will remain in the control plane for efficiency.

Conceptually, this architecture seems to comply with all of the necessary steps that are required to provision a new tenant. However, further optimization and discussion regarding this architecture is necessary in order to achieve the isolation that is required of a SaaS multi-tenant system.

5.3 Second Iteration

5.3.1 Establish Iteration Goal and Input Selection

The iteration goal is to refine the system's architecture to meet both functional and quality attribute requirements, with special emphasis regarding **isolation**. Therefore, this iteration must outline mechanisms to ensure that, for the current tenant, you are restricting access to those resources of only that tenant. Bear in mind that, for instance, filtering the users by tenantId isn't ensuring isolation. Isolation mechanisms have to be put in place in order to, independently of what a developer does in their code, any effort to access another tenant's resources will be blocked/prevented.

5.3.2 Choose elements of the system to decompose

On one hand, the control plane corresponds to a unique plane of glass for the SaaS environment, providing the team with a purpose-built collection of features and capabilities that are essential to operating the multi-tenant environment. The application plane, on the other hand, is responsible for hosting the actual business logic in tenant-specific environments, ensuring data segregation and consistent performance, according to the deployment model. Therefore, **the components present on the application plane are chosen for refinement** in this iteration.

Bear in mind that, as we progress, the line between architecture diagram and deployment model becomes blurier, but this is common given the multi-tenancy context of the scenario.

5.3.3 Choose Design Concepts that satisfy the iteration inputs

In order to satify the iteration goal and the input architectural drivers the following design strategies can be used:

- design an **architecture compatible with a mixed mode deployment** model, where we determine how each of the different services and resources should be deployed to meet the specific requirements of a given use case, in a pooled or siloed scenario.
- **Containerization and Orchestration** — The use of containerization will allow each component to run in isolated environments, making it easier to deploy (and scale) them independently. Orchestration tools such as Kubernetes also align very well with the SaaS multi-tenant requirements. Therefore, it will be employed to manage the lifecycle of these containers, enabling dynamic scaling based on demand and providing strong workload isolation through namespace segmentation, resource quotas, and network policies, which contributes to improved security, fault containment, and multi-tenancy support. While not a design strategy in itself, the adoption of a standard like Kubernetes will naturally influence architectural decisions, highlighting best practices, notation and conventions associated with the technology.
- adoptance of a **Service Mesh** pattern which is a dedicated infrastructure layer designed to manage, observe, and control communication between microservices within a Kubernetes cluster. Furthermore, it provides a multitude of mechanisms, which make it a better approach than simply using a sidecar with a reverse proxy. We'll later specify the functionalities and benefits of the service mesh.

5.3.4 Instantiate Architectural Elements, allocating responsabilities and defining interfaces

This section **refines the Application Plane** to enforce tenant isolation through a mixed deployment model, Kubernetes orchestration, and a service mesh. Building on the first iteration, this architecture ensures strict data and resource segregation to meet the isolation quality attribute. The following subsections present the architectural elements, their responsibilities, and their interfaces.

Architectural Elements

To achieve tenant isolation, the Application Plane is restructured using Kubernetes namespaces for each Tenant, with an Ingress Gateway managing incoming traffic. The architecture integrates the following components:

- Pooled Services: Includes an Ingress Gateway, serving as the entry point for tenant-specific requests, as well as the Media Service, which integrates with an external media blob storage solution.
- Siloed Services: Comprises the Auth, Hosted Site and Manager services deployed within separate namespaces for each tenant, Tenant 1 and Tenant 2.
- Supporting Components: Sidecar Reverse Proxy components injected with each microservice for service-to-service communication and tenant-aware routing (included in the Service Mesh Pattern).

Responsibilities

- **Ingress Gateway**: Authenticates incoming requests and routes them to the appropriate tenant namespace based on subdomain identifiers (e.g., t1.contentrus.com, t2.contentrus.com).
- **Auth Service**: Within each namespace, manages user authentication, ensuring secure access within each tenant context.
- **Hosted Site**: Within each namespace, the Hosted Site service delivers tenant-specific websites, handling runtime hosting tasks with isolation enforced by the Sidecar Reverse Proxy.
- **Manager Service**: Similar to the two previous services, but handles CMS management.
- **Media Service**: Manages submission of tenant-specific media assets with external pooled storage.
- **Sidecar Reverse Proxy**: Handles internal tenant-aware routing between services.

Interfaces

The interfaces for each service will not be outlined in this iteration, due to the added components being related to deployment and not changing the business logic of the service components, which are essentially agnostic to their existance (in the case of the sidecar reverse proxies).

5.3.5 Sketch Views and Record Design Decisions

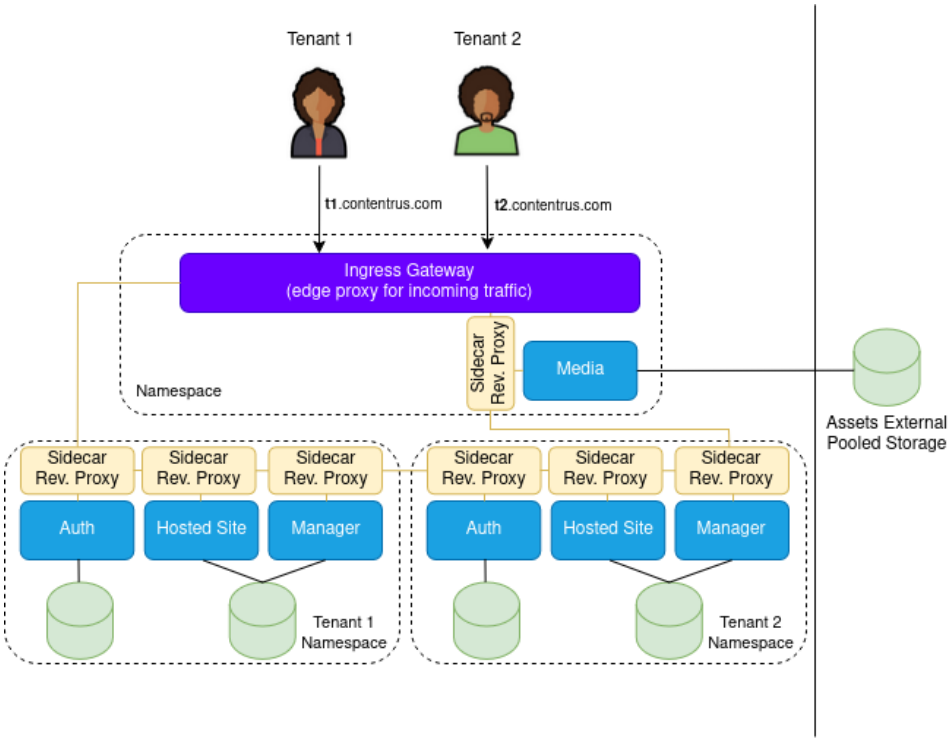

Figure 6: Architecture Diagram — Step 5 Iteration 2

Figure 6 showcases the designed solution at this step (just the application plane). Differently from the previous architecture, in this diagram it is highlighted the entrypoint to the application plane using a subdomain containing the tenantId (e.g. t1.contentrus.com & t2.contentrus.com). Although it may not appear important at first glance, this approach provides clear tenant isolation at the DNS level, simplifying the routing logic. For instance, if we had a custom tenant domain this would introduce the need for a Tenant Mapping service that would may be a possible bottleneck of the system, as every time we would need to route to tenant spefic components the translation/conversion would need to be carried out.

It employs a mixed mode deployment model with the following characteristics:

- Pooled Compute and External Storage for the Media Service. As previously mentioned, this external storage solution will be Azure Blob Storage, which provides policies for access control to the resources belonging to a given tenant. A pooled external storage approach was followed in order to limit the number of credits available at Azure Cloud (budget). As the storage resource is pooled, the sidecar reverse proxy it's extremely important as it provides the tenant context for the component to utilise different credentials to access the storage, specific to that tenant;
- Siloed Compute and Storage for the Auth Service, Hosted Site and Manager microservices.

Pooled resources share a common namespace, while siloed resources have independent namespaces corresponding to the given tenant. Cross-namespace access should be blocked by default. Fully siloed services (including compute and storage) are already bound to a specific tenant, essentially applying their isolation policies when they are deployed and configured (related to the provisioning scripts) — this is called Deployment-Time isolation. Services which posess pooled resources, need to ensure isolation through a runtime isolation model. This includes strategies that rely on the cooperation of the application

code or other constructs to dynamically acquire and apply the isolation policies. For example, by dinamically assuming credentials for a given tenant, the highlighted sidecar allows for limiting the access to azure blobs (Azure Blob Storage Service) correspondant to that tenant.

Furthermore, a service mesh is introduced which is a dedicated infrastructure layer designed to manage, observe, and control communication between microservices within a Kubernetes cluster. Each microservice in the Kubernetes cluster is paired with a sidecar proxy, which intercepts and manages all incoming and outgoing network traffic, routing it to the appropriate tenant as needed. When a microservice sends or receives a request, the sidecar proxy handles the communication, abstracting the network complexities away from the microservice itself. This benefit along with several others are highlighted in the ADR, as the reasons for this design decision.

- **ADR: Defining a Mixed Mode Deployment Model**

  Context The system follows a microservices architecture and needs mechanisms to enforce tenant isolation. It's necessary to define what resources will be siloed and pooled in the multi-tenant environment. Due to cost constraints it's already defined that the media storage will be pooled.

  Decision We have decided to follow a mixed mode deployment model, where some of the resources (both compute and storage) are either siloed or pooled, according with the following list:

  - Pooled Services include the Ingress Gateway, serving as the entry point for tenant-specific requests, and the Media Service (compute), which integrates with an external media blob storage solution (which is pooled due to cost constraints). The ingress gateway needs to be pooled for common access for multiple tenants in a non-tenant environment (entrypoint of application). The Media Service will also be pooled as we believe this will be a service which will not consume much resources, and, thereby, hurt performance. Therefore, measures will have to be taken in order to restrict access to tenant-level blobs on the external provider at runtime.
  - Siloed Services: Comprises the Auth, Hosted Site and Manager services (both compute and storage) deployed within separate namespaces for each tenant. This will be the core, more utilized and resource intensive services. Therefore, they are segregated to their own namespace providing isolation at deployment time at the infrastructure level.

  Consequences

  - Reduced operational/resource cost through shared infrastructure (media service and ingress gateway) and increased operational/resource cost through pooled infrastructure (Auth, Hosted Site and Manager services).
  - Increased complexity in access control and tenant routing: Since the Media Service and external blob storage are shared across tenants, additional runtime access control mechanisms must be implemented and maintained to enforce strict tenant data segregation at the application level. The Media Service (pooled resource) as well as the Ingress Gateway need to communicate with the siloed services, which are located in separate namespaces. This requires additional configuration and tenant-level routing.
  - Scalability flexibility: Siloed services can be scaled independently per tenant based on their specific resource needs.

- **ADR: Using a Service Mesh for service-to-service communication**

  Context The system requires a multi-tenant SaaS architecture that provides strong tenant isolation while maintaining effective service-to-service communication. The system follows a microservices architecture, with a mixed mode deployment model with both pooled and siloed resources. Pooled resources are isolated in independent namespaces, requiring efficient tenant aware routing between microservices in different namespaces.

  Decision We have decided to implement a service mesh pattern to manage service-to-service communication. Each microservice will be paired with a sidecar proxy that intercepts and manages all network traffic, providing tenant-aware routing and context for isolation mechanisms.

  Consequences

- Traffic management: A service mesh can provide advanced traffic management capabilities like load balancing, service discovery, and traffic routing, which help ensure that application traffic is routed efficiently and reliably. Moreover, the service-to-service communication, as well as the other related capabilitieis of the service mesh, are abstracted from the application code.
- Service-level observability — Service meshes typically offer observability signals such as metrics, distributed traces, and logs, which can be sent to our observability platform for analysis, alerting, and automation.
- Security: Service mesh can improve the security of microservices-based applications by providing end-to-end encryption, mutual authentication, and authorization policies.
- Consistency: a service mesh provides a consistent way of managing and configuring services across the infrastructure.
- Prevents Outages by implementing features like request timeouts to terminate long-running requests, rate limits to control traffic flow, and circuit breakers to limit traffic to unhealthy pods.
- The introduction of a service mesh introduces complexity at the deployment level.

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

The achieved design in this iteration focused on achieving tenant isolation within the application plan of the SaaS offering. This was accomplished by defining pooled and siloed resources for the various microservices and using a service mesh for service-to-service communication and tenant-aware routing (which also includes load balance and circuit breaking, among others).

In the next iteration, we'll focus on the performance and scalability of the existing solution in order to further refine its architecture.

5.4 Third Iteration

5.4.1 Establish Iteration Goal and Input Selection

The iteration goal is to refine the system's architecture to meet both functional and quality attribute requirements, with special emphasis on **performance and scalability**. Therefore, this iteration must outline mechanisms to satisfy or further improve the performance and scaling of the solution. It should also dwelve into the Noisy Neighbor problem. The noisy neighbor problem in multi-tenancy refers to a situation where one tenant in a shared computing environment consumes a disproportionate amount of shared resources, thereby degrading the performance and usability experienced by other tenants.

5.4.2 Choose elements of the system to decompose

The application plane is responsible for hosting the actual business logic in tenant-specific environments, ensuring data segregation and consistent performance, according to the deployment model. Therefore, **the components present on the application plane are chosen for refinement** in this iteration, as these are the ones related to the features and capabilities of the solution and that are provided in a multi-tenant environment. "The basic challenge is often all about scaling. Certainly, if a service can scale effectively enough to address the multitude of personas and workloads without overprovisioning or impacting other tenants, then you probably have a reasonable scope for your service. Our focus is on those scenarios where horizontal scale alone may not be effective or efficient enough to deal with the multi-tenant realities of your environment." [1]

5.4.3 Choose Design Concepts that satisfy the iteration inputs

The following design strategies, regarding performance and scalability, are already implemented from previous iterations:

- A **microservices architecture** is implemented with all services being deployed in orchestrated containers, leveraging **Horizontal pod autoscaling** policies based on usage and resource consumption to ensure high availability and performance under variable load conditions. A microservices architecture is of upmost importance due to allowing for scaling independent functions of our system without overprovisioning (don't scale parts of the system that do not need further scaling).

- **Mixed Mode Deployment Model**: Services expected of higher demands were specifically design as fully sioled (across both compute and storage), whereas services expected with less demand (such as Media) are designed to assume pooled computing models.

Although we think the explained measures, already satistfy our performance and scalability requirements, in order to further satify the iteration goal and the input architectural drivers the following additional design strategies are used:

- **Caching Strategy** — Adding distributed caching across multiple instances/pods in order to improve performance. The implementation of Piranha CMS already provides an interface for Distributed Caching.

5.4.4 Instantiate Architectural Elements, allocating responsabilities and defining interfaces

The new caching strategy will not introduce any business level architectural elements that need explanation in this section.

5.4.5 Sketch Views and Record Design Decisions

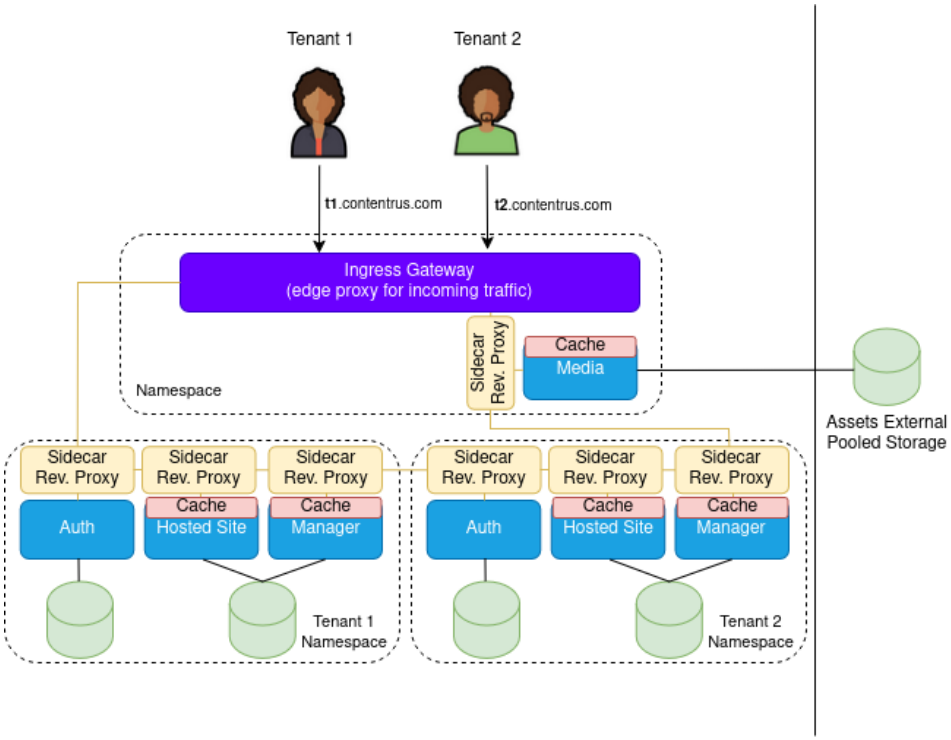

Figure 7: Architecture Diagram — Step 5 Iteration 3

Figure 7 showcases the introduction of caching in Hosted Site, Manager and Media services. Bear in mind this corresponds to distributed caching shared across multiple containers of said services. Furthermore, additional measures regarding isolation are needed in the case of the Media microservice, where cache may contain elements from multiple tenants. This design decision is documented in the following ADR:

- **ADR: Distributed Caching Strategy**

  Context The system follows a microservices architecture with strong tenant isolation and service-to-service communication. Siloed resources are present in a separate namespace for the tenant, while pooled resources share a common namespace.

  Decision We have decided to implemented a distributed caching mechanism accross multiple instances of the Media, Hosted Site and Manager services. The interface for this distributed caching implementation is already defined by the default Piranha CMS.

  Consequences

- Improved Performance and Reduced Latency: By caching frequently accessed data closer to the application services (Media, Hosted Site, and Manager), the system can reduce the number of expensive database and storage calls, thereby improving response times and overall user experience.
- Increased Complexity in Cache Management: The distributed nature of the cache may introduce additional challenges, such as cache consistency, eviction strategies. Special consideration must be given to ensure tenant-specific data is not leaked or incorrectly shared across tenants, especially in the pooled Media service.
- Improved Fault Tolerance: In the event of a backend service degradation or failure, the cache can serve as a temporary data source, improving the system's resilience and availability under stress conditions.

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

The validation of this approach would need load testing for performance and scalability testing for the microservices architecture, with some method to evaluate degradation of performance in neighbors (noisy neighbor problem). However, a baseline of the number of tenants we want to serve and the traffic we are intending to support is also of great importance.

"As a rule of thumb, any environment should be looking at ways to better address performance and scale through more granular services." [1], i.e., as we identify the possible bottlenecks and noisy neighbor candidates we should extract them to scale them as independent services. The services picked on this report (day one) should be expected to morph as the system evolves and we gain richer insights into how and where we're observing noisy neighbor conditions. In order to gain this kinds of insights, observability and resource consumption metrics need to be heavily present.

This corresponds to the final iteration and architecture.

6 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. As this is a multi-tenancy project, we've already been discussing cross-cutting concerns throughout the report.

6.1 Security and Tenant Isolation

Security and authentication are fundamental in any system, but they are even more critical in multi-tenant environments. Users must be securely authenticated and authorized within the boundaries of their own tenant. Therefore, it's necessary to extract the `tenantId` and provide it for the downward services as tenant context. In the designed architecture, extracting the tenant context is the responsability of the Ingress Gateway, with the service mesh proxies being responsible for passing down this context for the communicating services.

Fully siloed resources provide tenant isolation at deployment time, with tenant-exclusive namespaces, which by default will block any cross tenant access. Pooled resources such as the Media Service will provide data isolation to the external storage solution at runtime. As previously mentioned, the service mesh pattern provides the services with the tenant context, which will have to map into credentials of the external media blob storage for that specific tenant. Accessing the cloud storage solution with different credentials is already supported by their API, only being necessary to save these tenant-specific credentials safely.

Furthermore, the existing Piranha CMS solution employs Role-Based Access Control (RBAC) mechanims, where permission claims (such as read, write or delete of certain resources) are assigned to roles, which are then assigned to users. Thus, within each tenant's environment the user's access to resources is determined by the roles assigned to it, following the Least Privilege Principle.

6.2 Logging and Observability

Observability is vital for understanding the system's behavior, supporting debugging, and auditing usage. In a multi-tenant system this gains further importance as it's needed to actively measure the

resource consumption and performance to possibly pinpoint potential bottlenecks and causes for the noisy neighbor problem. Thus, special care must be taken to ensure that logs, metrics and traces are:

- Structured and tagged with `tenantId`
- Isolated to prevent information leakage across tenants
- Compliant with privacy and security standards

Real-time visibility into the system's performance and tenant usage is essential for maintaining the platform's health. Examples of metrics that should be measured include: CPU and memory usage; per-tenant request latency; Error rates per tenant. The observability system, must provide a way to monitor on a per-tenant or global basis. It must also provide alerting mechanisms, when such metrics achieve alarming values.

6.3 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. Well-defined timeout strategies and automatic retry logic is also planned for integrations with external services, such as with the external billing provider through a message queue (already outlined previously). Mechanisms should also be put into place so that everytime a error happens users receive friendly error messages that do not expose internal details. Lastly, the monitoring system should also provide notifications in the case of important/alarming errors.

7 Roadmap

From the outlined architecture and functional requirements, we'll focus for the next few weeks on completing the following features: **1.** Developing the automation provisioning scripts for each tenant, configuring the siloed tenant environments and pooled infrastructure; **2.** Developing the self-provisioning flow, from the interface, until the automated execution of said scripts. This way, we'll complete the main user flow of the application, ending up with multiple logically isolated CMS built websites for multiple tenants.

The roadmap for implementing said features is divided into the following 1 week sprints:

- 06/05 - 13/05 — Separation of Piranha CMS into multiple microservices and containerization of said microservices;
- 13/05 - 20/05 — Orchestration of the containerized environments with kubernetes and developing the automation scripts for provisioning the multi-tenant environment (pooled and siloed resources). This includes enabling service-to-service communication through the service mesh:
- 20/05 - 27/05 — Ensure isolation mechanisms for the pooled media storage and start building the self-provisioning interface;
- 27/05 - 03/06 — Terminate self-provisioning interface and trigger the provisioning scripts for the specified tenant.

The roadmap starts with the multi-tenancy environment setup process, so that we can achieve the main goal of the project earlier in the roadmap deadlines. Building the full self-provision interface with sign-up, tiering, billing and creating the environment may be too much for the 4 week period. If needed, and because this is an academic project, we'll take shortcuts on the billing integration with an external provider (we'll mock it instead).

8 Conclusion

This report detailed the design process and evolution of the open source Piranha CMS platform to support a multi-tenant SaaS platform. The proposed transformations addresses the systems current limitations by applying the Attribute-Driven Design (ADD) methodology across three key iterations: Separation of Concerns, Tenant Isolation, and Performance & Scalability. Through the use of Domain-Driven Design (DDD), core, generic and supporting subdomains were clearly identified and aligned with the system's strategic goals.

Regarding the achieved architecture, the monolithic structure was refactored into a microservices-based architecture divided into a Control Plane and an Application Plane, each with clear responsibilities and interfaces. Kubernetes orchestration, mixed deployment models, and a service mesh pattern were incorporated to enforce deployment-time and runtime isolation, ensure secure and fault-tolerant service communication, and enable horizontal scalability. Cross-cutting concerns such as security, observability, and fault isolation were systematically integrated to uphold the system's reliability and compliance.

The resulting architecture offers strong tenant isolation, enhanced modularity, automated onboarding, and robust scalability strategies while remaining mindful of resource constraints. The solution also establishes a foundation for iterative performance improvements based on observability insights, particularly important to mitigate noisy neighbor effects as the platform scales.

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] Piranha CMS, "How to use multitenancy," accessed: 2025-05-09. [Online]. Available: https://piranhacms.org/docs/master/tutorials/how-to-use-multitenancy