

# **Evolving Piranha CMS for ”Contents’R’Us”:**

Event-Driven Extensions & Secure Integrations

**Daniel Madureira - 107603**

**João Andrade - 107969**

**José Gameiro - 108840**

**Tomás Victal - 109018**

Professor Cláudio Teixeira

Software Architectures

DETI

Universidade de Aveiro

May 2025

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	System Analysis . . . . .	3
1.2	Project vision . . . . .	3
<b>2</b>	<b>Architectural Design Methodology</b>	<b>5</b>
2.1	Advantages . . . . .	5
2.2	Alternatives . . . . .	5
2.3	Application of ADD in Our Context . . . . .	6
2.3.1	Step 1: Identify Architectural Drivers . . . . .	6
2.3.2	Step 2: Establish Iteration Goal . . . . .	6
2.3.3	Step 3: Choose Architectural Elements to Decompose . . . . .	6
2.3.4	Step 4: Select Design Concepts . . . . .	7
2.3.5	Step 5: Instantiate Architectural Elements . . . . .	7
2.3.6	Step 6: Verify and Refine Requirements . . . . .	7
2.3.7	Step 7: Repeat Steps 2-6 for Next Element . . . . .	7
<b>3</b>	<b>Design Iterations</b>	<b>8</b>
3.1	Iteration 0 . . . . .	8
3.1.1	Functional Requirements . . . . .	8
3.1.2	Quality Attributes . . . . .	8
3.1.3	System Constraints . . . . .	9
3.2	Iteration 1 . . . . .	9
3.3	Iteration 2 . . . . .	11
3.4	Iteration 3 . . . . .	12
3.5	Iteration 4 . . . . .	13
<b>4</b>	<b>Core Domains</b>	<b>16</b>
4.1	Core Domains . . . . .	16
4.1.1	Event Publishing Domain . . . . .	16
4.1.2	Subscription Management Domain . . . . .	17
4.1.3	External Integration and Inbound Event Handling Domain . . . . .	18
4.1.4	Security and Access Control Domain . . . . .	18
4.2	Supporting Subdomains . . . . .	19
4.2.1	Monitoring & Observability Subdomain . . . . .	19
4.3	Generic Subdomains . . . . .	20
4.3.1	User Management Subdomain . . . . .	20
4.3.2	CMS Platform . . . . .	21
<b>5</b>	<b>Cross-Cutting Concerns</b>	<b>22</b>
5.1	Security . . . . .	22
5.2	Logging and Auditing . . . . .	22
5.3	Error Handling and Resilience . . . . .	23
5.4	Observability and Monitoring . . . . .	23
5.5	Performance and Scalability . . . . .	23
5.6	Compliance and Data Protection . . . . .	24

<b>6</b>	<b>Proposed Architecture and Roadmap</b>	<b>25</b>
6.1	The Components and their responsibilities . . . . .	26
6.1.1	Controllers . . . . .	27
6.1.2	Webhooks Dispatcher . . . . .	27
6.1.3	Subscriptions Manager . . . . .	27
6.1.4	Publish Receiver . . . . .	27
6.1.5	Message Broker . . . . .	27
6.1.6	Event Manager Service . . . . .	28
6.1.7	Piranha CMS Core . . . . .	28
6.1.8	Key Manager . . . . .	28
6.2	Communication/Interfaces Between Components . . . . .	29
6.3	Road Map . . . . .	29

# 1 Introduction

## 1.1 System Analysis

**Piranha CMS** is a lightweight, cross-platform content management system built on **.NET Core**. It provides essential CMS features such as modular content types, intuitive page editing, media management, and extensibility through plugins and APIs. Its architecture is clean, and the core design principles favor modularity and developer-friendliness, making it a solid foundation for modern applications.

However, as we evaluate the CMS in the context of Scenario 3: Event-Driven Extensions and Secure Integrations, several critical gaps emerge:

- **Lack of Event Infrastructure:** The CMS does not natively support domain event emission (e.g., on content creation or deletion). There is no built-in event bus or observable mechanism for internal services or third parties to subscribe to.
- **No Inbound Event Handling:** Piranha CMS is designed as a pull-based system. It cannot receive or react to externally pushed events.
- **Security Limitations for Integrations:** Although Piranha provides some API-level extensibility, it lacks support for secure, configurable integration endpoints. There are no native mechanisms for key management, event signing, or authentication of external publishers.
- **Synchronous Design Bias:** The current design presumes user-initiated operations and lacks asynchronous processing capabilities (e.g., message queues, background event handling).

These limitations are in conflict with the requirements of a modern CMS that must operate in complex ecosystems involving CRMs, marketing automation tools, analytics platforms, and third-party content pipelines.

## 1.2 Project vision

The vision of this initiative is to transform Piranha CMS into a modern, asynchronous content platform that is integration-ready and resilient. The evolved system will act not just as a content manager, but also provide a real-time event-driven ecosystem.

This transformation will be driven by the principles of loose coupling, secure communication, and configurability, allowing external systems to push and pull data dynamically while maintaining trust, integrity, and performance.

With this in mind, three main strategic goals have been defined:

1. **Custom Publish/Subscribe Mechanism** - Introduce a flexible domain eventing system that allows administrators to configure which models (e.g., pages, posts, media) emit events and under what circumstances. The system must also support outbound event formatting, topic-based filtering, and dynamic subscription registration, signing, and validation.
2. **Inbound Integration via External Publishers** - Enable Piranha CMS to receive and process events from authorized third-party systems. This requires a configuration layer to register publishers, define accepted message types, and map incoming events to internal workflows or commands (e.g., content updates, status changes).

3. **Secure, Configurable Messaging Infrastructure** - Establish robust security practices across all event flows. This includes:

- Key and token management for authentication;
- Message signing and verification to ensure integrity;
- Endpoint configuration interfaces for defining allowed targets and actions;
- Logging and audit trails for traceability.

## 2 Architectural Design Methodology

In order to evolve the Piranha CMS into an event-driven platform with secure integration, we choose the Attribute-Driven Design (ADD) as our primary architectural design methodology.

### 2.1 Advantages

The Attribute-Driven Design provides several advantages that aligns specifically well with our chosen scenario.

ADD focuses on **quality attributes** in architectural decision-making, which corresponds directly with our emphasis on performance, scalability, security, and configurability. By placing these architectural drivers at the center of the design process, ADD enables the construction of a system that is not only functional but also robust, scalable, and future-proof.

Furthermore, ADD's **top-down** and **iterative** approach encourages gradual decomposition of functionality based on concrete scenarios, starting with high-level structures and progressively refining them through multiple iterations. This aligns well with evolving an existing system like Piranha CMS, where we must carefully integrate new capabilities while preserving existing functionality.

ADD provides a catalog of **architectural tactics** that directly address specific quality attributes. These proven design approaches offer valuable guidance for implementing event-driven patterns, security mechanisms, and scalability strategies.

Moreover, ADD complements Domain-Driven Design principles, allowing us to incorporate domain modeling while maintaining a focus on architectural quality attributes. This combination will help ensure that our event system aligns with meaningful business domains.

On top of all these benefits, this methodology also emphasizes documenting both the decisions made and the rationale behind them, creating a valuable reference for future development and maintenance.

### 2.2 Alternatives

When choosing the right architecture design methodology we considered some other alternatives that we later discarded. One of the discarded options is TOGAF (The Open Group Architecture Framework).

TOGAF is a well-established framework, especially in enterprise environments, where long-term planning, organizational modeling, and governance are priorities. However, for a focused, technical implementation like ours, centered on integrating specific components into an existing CMS, it turned out to be too broad and high-level. Using TOGAF would've added a lot of overhead in terms of documentation and formal process, without really helping us solve the technical challenges at hand.

This framework is more useful when working on large-scale systems across multiple departments or organizations, and since we were working on a tightly scoped extension to an open-source CMS, TOGAF felt like overkill. It simply didn't offer the hands-on tools we needed to handle things like validating events, managing retries, or isolating faults in a microservice-style design.

We also looked at ACDM (Architecture-Centric Design Method). It has a more structured, step-by-step approach that focuses on formal processes and repeated cycles of design and validation. ACDM is great for projects in highly regulated industries or systems that need rigorous documentation and oversight. But our project needed to move fast and stay flexible. We had to be able to experiment—try out different messaging flows, plug in new modules, and quickly adapt to feedback. ACDM, with its strict phases and heavier process, would have slowed us down. It also doesn't focus as much on designing around key quality attributes like resilience or extensibility, which were top priorities for us.

In the end, we chose ADD because it gave us the best of both worlds: enough structure to keep our architecture sound, but enough flexibility to evolve quickly. It let us design around real scenarios and quality goals without getting bogged down in bureaucracy, and that made it the right fit for building an event-driven extension that works seamlessly with the existing CMS.

## **2.3 Application of ADD in Our Context**

We will apply the Attribute-Driven Design methodology through the following process:

### **2.3.1 Step 1: Identify Architectural Drivers**

The first step involves identifying the key architectural drivers that will shape our design decisions. These include:

- Primary functionality requirements derived from the strategic goals
- Critical quality attributes identified in our system analysis
- Project constraints, including compatibility with the existing Piranha CMS architecture
- Technical environment considerations, such as deployment platforms and integration technologies

### **2.3.2 Step 2: Establish Iteration Goal**

For each iteration of the design process, we will establish specific decomposition goals. Our initial iterations will focus on the high-level system structure, particularly the event infrastructure and integration gateways. Subsequent iterations will refine component designs and interface specifications.

### **2.3.3 Step 3: Choose Architectural Elements to Decompose**

Based on the iteration goal, we will select specific elements for further decomposition. This process will begin with the core system modules and progressively refine their internal structures and relationships.

#### **2.3.4 Step 4: Select Design Concepts**

For each element being decomposed, we will select appropriate design concepts based on our architectural drivers. These concepts will include patterns such as:

- Event-driven architecture patterns for the publish/subscribe system
- Separation of concerns using a layered approach

#### **2.3.5 Step 5: Instantiate Architectural Elements**

We will instantiate specific architectural elements by applying the selected design concepts. This includes defining components, connectors, interfaces, and their properties. For each element, we will specify responsibilities, relationships, and quality attribute characteristics.

#### **2.3.6 Step 6: Verify and Refine Requirements**

After each iteration, we will verify that the decomposed elements satisfy the architectural drivers and refine our understanding of requirements as needed. This feedback loop ensures that our architecture remains aligned with project goals throughout the design process.

#### **2.3.7 Step 7: Repeat Steps 2-6 for Next Element**

We will repeat this process for each architectural element, progressively refining the overall system design through multiple iterations.



## 3 Design Iterations

Following the ADD methodology, our architecture was developed through five main iterations, which are described below.

### 3.1 Iteration 0

To kickstart the ADD process, we first needed to figure what our primary functionalities were as well as our constraints, quality attributes and architectural concerns. On the following tables we'll expose our main functional requirements, quality attributes and constraints, and explore how they evolved alongside the iterative process.

#### 3.1.1 Functional Requirements

To better organize the functional requirements, we identified two primary system actors: the **Administrator** and the **External Publisher**, each associated with specific responsibilities and interactions within the system.

ID	Requirement	Actor(s)
<b>FR-1</b>	Define possible actions for external publishers	Admin
<b>FR-2</b>	Publish new content	External Publisher
<b>FR-3</b>	See all the existing events	Admin
<b>FR-4</b>	Decide if a given content will be published	Admin
<b>FR-5</b>	Subscribe and unsubscribe to existing events	External Publisher
<b>FR-6</b>	Define authorized external publishers	Admin
<b>FR-7</b>	See new published events	Admin & External Publisher
<b>FR-8</b>	Needs to integrate easily with external services	Admin & External Publisher

#### 3.1.2 Quality Attributes

Our quality attributes focus on 9 main categories that are presented below:

ID	Attribute	Category
<b>QA-1</b>	Event processing (publishing and inbound) must occur with a maximum latency of 1 second under normal load	Performance
<b>QA-2</b>	The system must support up to 10,000 concurrent event subscriptions and 1,000 inbound events per minute without service degradation	Scalability
<b>QA-3</b>	All event communications (inbound and outbound) must be authenticated and signed to ensure integrity and authenticity	Security
<b>QA-4</b>	The system should achieve 99% up time for the Event Manager and Inbound Handler modules	Availability
<b>QA-5</b>	Code for new modules should follow clean architecture principles and allow easy extension	Maintainability

ID	Attribute	Category
<b>QA-6</b>	If an event fails to deliver, the system should try again up to three times with increasing delays before sending it to a dead letter queue.	Reliability
<b>QA-7</b>	The admin configuration interface must allow an event to be fully configured (published or subscribed) within 5 clicks	Usability
<b>QA-8</b>	All event transactions (successful, failed, retried) must be logged with timestamp, event type, and endpoint information	Logging & Monitoring
<b>QA-9</b>	If sensitive data is part of the payload, messages must comply with GDPR (i.e., encryption at rest and in transit)	Compliance

### 3.1.3 System Constraints

The table below summarizes the key technical and organizational constraints that guided our system design.

ID	Constraint
<b>CON-1</b>	Our work must be based on the existing Piranha CMS (.NET Core, C#) framework
<b>CON-2</b>	Must use open-source or .NET-compatible libraries (no paid 3rd-party tools without prior approval)

With these artifacts in mind, we were finally able to fully start the iterative process within the ADD methodology.

## 3.2 Iteration 1

The initial design focuses on introducing a foundational event-driven architecture layered on top of the existing Piranha CMS core. This version introduces several new components to support publish/subscribe functionality.

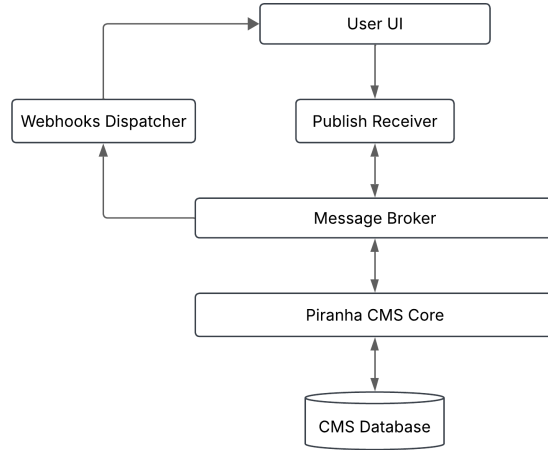


Figure 1: Iteration 1 Architecture

The core components in this iteration include the following:

- **User UI:** Acts as the primary interface through which users trigger content actions (like publishing, updating or deleting content);
- **Publisher Receiver:** Captures publishing events from the UI and passes them to the message processing system;
- **Message Broker:** A decoupled messaging layer that asynchronously routes events between components, ensuring scalable and reliable delivery;
- **Piranha CMS Core:** The existing CMS logic which processes the event data and updates the database;
- **CMS Database:** The persistent storage used by Piranha CMS;
- **WebHooks Dispatcher:** Handles outbound events by forwarding messages to external subscribers (third-party systems).

This iteration focuses on basic decoupling and allows the system to be observed externally, referencing WebHooks as described by Microsoft[1].

## Limitations and Observations

With this initial architecture in mind, our team found some aspects that were missing:

- **Security:** While the architecture introduces integration points for external services, it does not yet address message integrity, endpoint authentication, or permission controls;
- **Asynchronous layer:** After taking another look at our quality attributes, we noticed that this architecture can't handle asynchronous event delivery, as it needs that the event consumers to be online to receive the said event;
- **Subscription Management:** A component is missing to allow for management subscriptions. Without it, system administrators or external clients have no way to configure which events they want to subscribe to;

- **No User-Defined Configurations:** There isn't a way for administrators to configure endpoints, keys, or choose which models should publish or subscribe to events.

### 3.3 Iteration 2

In our next iteration, we focused on the **asynchronism** of the system and on **subscription management**. This step introduced the logic to handle event routing when subscribers are offline, enabling the system to operate asynchronously. After that, we also reviewed the event flow from the previous iteration and improved it by applying the publish/subscribe (pub/sub) pattern.

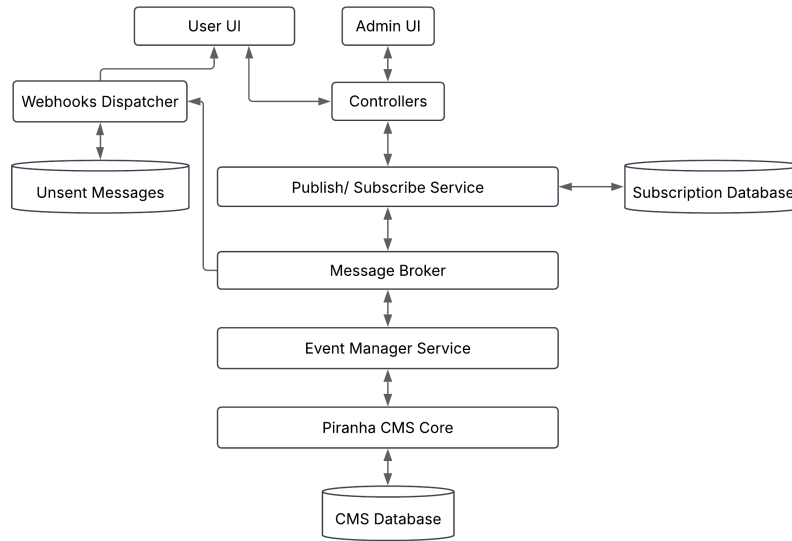


Figure 2: Iteration 2 Architecture

With these goals in mind, this iteration introduced some core components to our architecture:

- **Message Broker:** This component acts as a central channel for communication between producers (like the CMS core or external event generators) and consumers (such as webhook dispatchers or subscription managers). It abstracts the complexities of direct communication and enables asynchronous messaging.
- **Publish/Subscribe Service:** This service registers consumers interested in specific event types and routes messages accordingly. This introduces a scalable pattern where multiple services can react to CMS events without tight coupling to the CMS logic.
- **Subscription Database:** A new persistent storage layer is introduced to keep track of subscriber preferences. This enables dynamic registration and ensures resilience during service restarts or failures.
- **Controllers and UI Refinement:** The controller layer is now formally introduced to interface between the Admin/User UIs and the publish/subscribe layer, ensuring a clear separation between presentation and logic layers.

- **Webhooks Dispatcher Integration:** The webhooks dispatcher now consumes messages from the broker. If a webhook cannot be sent, it is stored in the Unsent Messages database for retries, introducing reliability and fault-tolerance.

With these new components we were able to introduce some architectural patterns and reinforce some others. On one hand, the broker pattern ensures the loose coupling and the asynchronous processing of the events while the publish/subscribe pattern allows for extensibility, enabling easy on-boarding of new consumer services such as analytics, monitoring, or real-time notifications.

## Limitations and Observations

Now with these changes applied to the architecture our group still noticed that some components were missing:

- **Security:** A security module was still not addressed to evaluate the integrity of all the messages exchanged, endpoint authentication or permission controls;
- **User-Defined Configurations:** Another limitation that was also not addressed in this iteration was the lack of allowing administrators to configure endpoints, keys, or choose which models should be published or subscribe to events.

## 3.4 Iteration 3

For the third iteration, we focused on improving the publish/subscribe system by decoupling responsibilities into distinct services and establishing internal communication protocols. The goal was to enhance maintainability, enable more granular control over event flow, and begin addressing foundational security concerns.

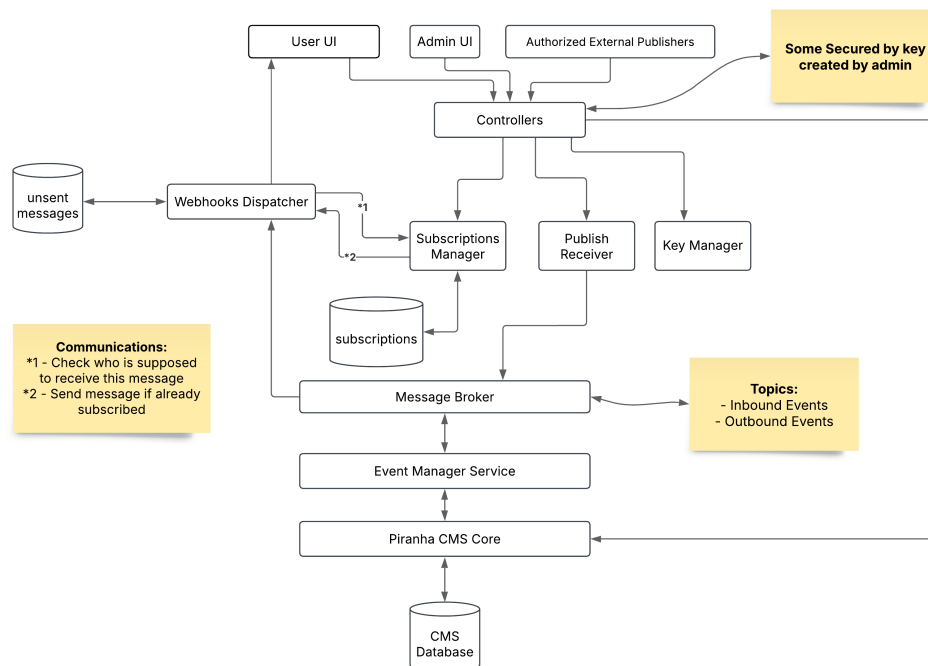


Figure 3: Iteration 3 Architecture

As we refined the architecture, we added some more core components:

- **Service Separation:** The publish and subscribe functionalities are now handled by separate services. This separation enables independent scaling, clearer responsibilities, and easier evolution of each component.
- **Subscription Manager & Webhooks Dispatcher Integration:** A communication channel was established between the Subscription Manager and the Webhooks Dispatcher. This allows the dispatcher to:
  - Know which subscribers should receive messages for a given event.
  - Receive notifications when a previously subscribed user reconnects, triggering the dispatch of any unsent messages.
- **Unsent Message Handling:** With the new coordination between services, unsent messages can now be reliably delivered when conditions allow (e.g., subscriber becomes available again), improving delivery guarantees.
- **Authentication Service:** A new service was introduced to manage security credentials. It supports:
  - Generating authentication keys for publishers.
  - Validating these keys before allowing message publication.

This iteration improved system modularity through clearer service boundaries and enhanced reliability by integrating subscription management with message delivery. Early authentication mechanisms were introduced to support secure event publishing, and the system became more resilient to temporary delivery failures, ensuring better service for subscribers.

## Limitations

While this iteration introduced key architectural improvements, the authentication service is still in an early stage and lacks fine-grained access control. Additionally, message de-duplication and rate-limiting are not yet implemented, which may pose issues in high-traffic environments.

Another limitation we identified was the lack of detail regarding the Piranha CMS Core. In this iteration, the component is represented in an abstract and high-level manner, which made it difficult to determine which specific internal modules or services our implementation would need to interact with.

## 3.5 Iteration 4

In our final iteration, we took the feedback provided from our presentation and focused on the most referred topic: the unclear integration with the Piranha CMS core and how external systems can see content from our CMS.

The goal in this iteration was to address how our system would interact with the existing Piranha CMS Core given its poorly described architecture. In the end, we achieved this architecture:

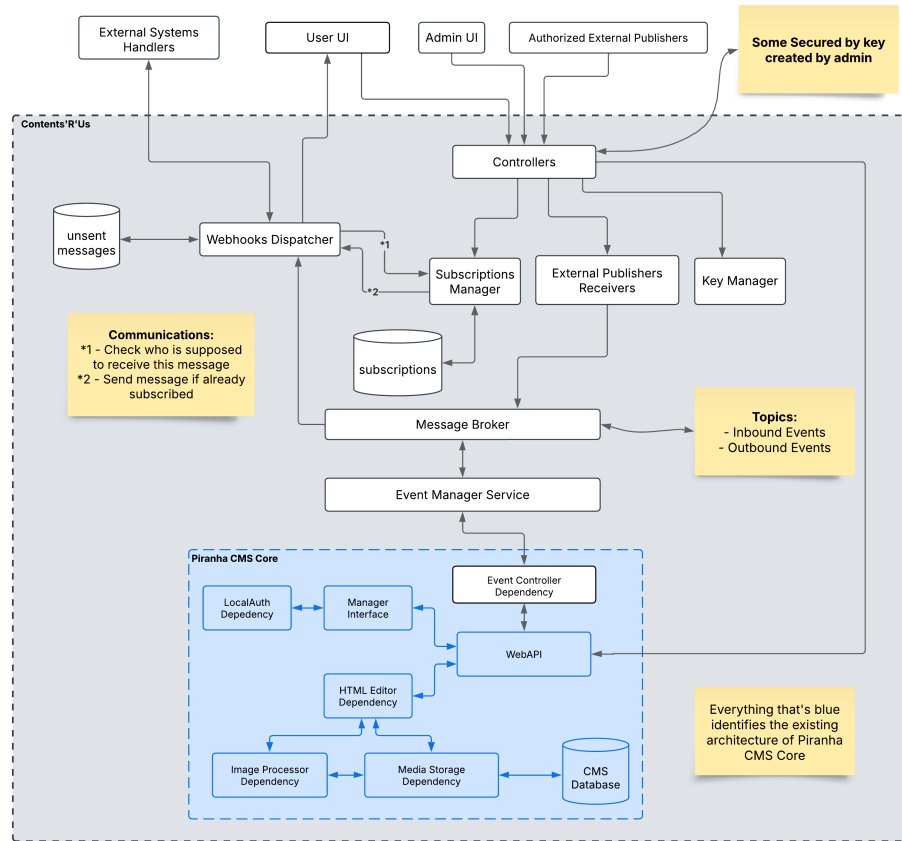


Figure 4: Iteration 4 Architecture

This last iteration added more components to our architecture:

- **Piranha CMS Core components:**

- WebApi;
- HTML Editor Dependency;
- Image Processor Dependency;
- Media Storage Dependency;
- Manager Interface;
- LocalAuth Dependency;
- CMS Database.

- **Event Controller Dependency:** Captures and forwards internal CMS events to the Event Manager Service, acting as the bridge between traditional CMS operations and the event-driven infrastructure.

This iteration allowed us to rethink how we integrate the newer architectural components into the existing core application. With this new architecture we explicitly mapped the dependencies to existing Piranha CMS Core (blue components), (LocalAuth, WebAPI, HTML Editor, Media Storage, etc). We also added a new dependency to extend the core CMS and effectively enabling allows the communication between our new components and the ones present the Piranha CMS core.

This final iteration significantly enhances the architecture by explicitly defining the boundaries and dependencies of the Piranha CMS Core, resolving previous ambiguities. By introducing the Event Controller Dependency, the system now supports seamless event propagation from internal CMS actions to external services, enabling robust publish/subscribe capabilities.



## 4 Core Domains

To implement a modern, event-driven architecture in Piranha CMS, we applied **Domain-Driven Design (DDD)** to identify, model, and modularize the key areas of the system. Our goal is to isolate core domains that encapsulate distinct responsibilities, maintain high cohesion, and remain loosely coupled to facilitate extensibility, integration, and testability.

### 4.1 Core Domains

We identified the following **four core domains**, each reflecting a critical capability required to achieve the project vision:

- **Event Publishing Domain**
- **Subscription Management Domain**
- **External Integration and Inbound Event Handling Domain**
- **Security and Access Control Domain**

These domains represent distinct bounded contexts. They communicate via well-defined interfaces and event-driven mechanisms, ensuring that changes in one domain have minimal impact on others.

#### 4.1.1 Event Publishing Domain

This domain handles the production, structuring, and broadcasting of outbound domain events related to CMS actions. It ensures decoupled, asynchronous communication with external systems.

#### Key Responsibilities

- Define **event contracts** for core CMS actions (e.g., `ContentCreated`, `ContentUpdated`, `ContentDeleted`);
- Manage a central **Message Broker** to decouple producers (CMS Core/Event Manager) from consumers (external systems);
- Coordinate with the **Webhook Dispatcher** to deliver events only to valid subscribers;
- Tag each event with metadata (e.g., content type, timestamp, origin) for filtering/routing;
- Support topic-based filtering and load balancing for scalability.

## Key Entities/Services

- **Event**: Domain object representing a business-relevant action;
- **EventTopic**: Logical grouping of similar events;
- **MessageBrokerService**: Handles asynchronous event queuing and delivery;
- **WebHookDispatcher**: Retrieves pending messages and invokes external webhooks;

## Technical Considerations:

- Use a reliable queueing mechanism like **Kafka**.
- Guarantee **at-least-once** or **exactly-once** delivery semantics

### 4.1.2 Subscription Management Domain

This domain is responsible for handling how external consumers subscribe to CMS events. It encapsulates the logic and policies for managing subscriptions and their lifecycle.

## Key Responsibilities

- Manage and store subscription data, topics, endpoints, filters, and permissions;
- Provide an admin interface to register/update/remove subscriptions.
- Validate new subscriptions (e.g., endpoint health check, authentication);
- Coordinate with the WebHook Dispatcher to ensure correct message routing.

## Key Entities/Services

- **Subscription**: Domain entity representing a single subscription instace (e.g., topic, endpoint, secret);
- **SubscriptionManagerService**: Core service that verifies and maintains the subscription catalog;
- **API Gateway**: Exposes public/admin-facing endpoints for subscription management;
- **subscriptions data store**: Persistent storage for registered subscription.

## Technical Considerations

- Rate-limit and throttle endpoints to prevent overload;
- Support retry policies and failure handling for undelivered messages;
- Encrypt sensitive data like webhook secrets or callback URLs.

### 4.1.3 External Integration and Inbound Event Handling Domain

Handles secure authenticated intake of domain events initiated by external systems (like blogs, social media, etc). It translates these events into internal CMS actions.

#### Key Responsibilities

- Accept inbound event through a secure API interface;
- Validate incoming messages using API keys and digital signatures;
- Forward validated events to the Message Broker for processing by the **Event Manager Service**
- Support a wide range of actions triggered externally, like, content publication, status updates, etc.

#### Key Entities/Services

- **InboundEvent**: Represents externally sourced events;
- **Publisher**: Represents an authorized external entity allowed to send incoming events;
- **PuublishReveicer**: Component that receives, authenticates, and validates incoming events;
- **EventProcessor**: Applies CMS-side logic based on event type (example: update content in DB).

#### Technical Considerations

- Enforce validation, signature verification, and IP whitelisting;
- Log and audit all incoming events for traceability and debugging;
- Allow external publishers to manage configurations through the admin interface.

### 4.1.4 Security and Access Control Domain

A cross-domain, foundational layer that ensures only authenticated and authorized actors interact with the system, protecting both event publishing and subscription mechanisms.

#### Key Responsibilities

- Generate and manage API keys for all publishers and subscribers;
- Authenticate incoming and outgoing requests via keys or tokens;
- Define role-based access control (RBAC) for internal users managing subscriptions;
- Encrypt/decrypt secure fields and enforce HTTPS-only communication.

## Key Entities/Services

- **Key**: Represents credentials issued to a publisher or subscriber;
- **KeyManager**: Service responsible for creating, validating, rotating, and revoking keys;
- **AuthService**: Provides token validation, signature checks, and scopes;
- **Permission**: Defines what each key can do, like, publish only, subscribe only.

## Technical Considerations

- Implement JWT-base verification;
- Provide an admin dashboard for managing secrets;
- Include audit trails and anomaly detection for suspicious usage.

## 4.2 Supporting Subdomains

For **Supporting Subdomains** our group only found and defined one, **Monitoring & Observability**

### 4.2.1 Monitoring & Observability Subdomain

This domain is responsible for ensuring operational transparency across the system. It provides real-time and historical insights into the delivery pipeline of events and helps diagnose system issues, ensures Service Level Agreements (SLAs), and support auditing for compliance.

## Key Responsibilities

- Log **delivery attempts**, including timestamps, endpoint targets, and payload metadata;
- Track **delivery status**: successful deliveries, retries, and failures;
- Store and expose **metrics** like latency, throughput, error rates, retry counts, and subscriber-specific health;
- Maintain **audit trails** of event flows for compliance and debugging (for example, GDPR, or HIPAA);
- Alert anomalies like repeated failures or unexpected latency spikes.

## Key Entities/Services

- **DeliveryLog**: Records each outbound delivery attempt and its outcome;
- **RetryMonitor**: Tracks and schedules retries for failed deliveries;
- **AuditTrail**: Captures immutable records of key actions (e.g., key creation, publisher registration);

- **MonitoringService**: Collects and aggregates system metrics;
- **AlertingSystem**: Generates alerts based on metric thresholds or error patterns.

## Technical Considerations

- Integrate with tools like Prometheus, Grafana, Jaeger, or ELK stack;
- Ensure logs are structured and filterable by event type, endpoint, and status;
- Implement retention policies and anonymization if sensitive data is logged;

## 4.3 Generic Subdomains

In terms of **Generic Subdomains** only identified 2 main ones:

- **User Management**
- **CMS Platform**

### 4.3.1 User Management Subdomain

Leverages the existing identity and access control infrastructure provided by Piranha CMS. It manages users who interact with admin-facing features like key management, subscriptions, and integration configuration.

## Key Responsibilities

- Authenticate and authorize admin users;
- Manage user roles and permissions, like, system admin, integration manager, etc;
- Enable secure login and session management;
- Support multi-user operations, audit user actions, and enforce RBAC policies.

## Key Entities/Services

- **User**: Represents a system user with login credentials and metadata;
- **Role**: Encapsulates a permission group, that can manage keys, or view logs, etc;
- **AuthenticationService**: Handles sign-in and session validation;
- **AuthorizationService**: Enforces what actions a user is permitted to take.

## Technical Considerations

- Extend Piranha's built-in identity provider where necessary;
- Enforce password and session security best practices;
- Log user actions related to integration and security settings.

### 4.3.2 CMS Platform

This subdomain consists of the foundational Piranha CMS capabilities. It manages content models, templates, media, and editorial workflows, acting as the *content engine* that event-driven extensions are layered on top of.

#### Key Responsibilities

- Handle core content creation, editing, versioning, and publishing workflows;
- Serve API responses and web views using Piranha’s headless and hybrid CMS features;
- Act as data source for outbound events, like when a new page is published;
- Enable extension through modules, middleware, and API endpoints.

#### Key Entities/Services

- **Page, Post, Block, Region:** Piranha’s core content entities;
- **ContentService:** Handles business logic around content lifecycle;
- **CMS API:** Serves data to front-end clients or headless consumers;
- **Piranha Middleware:** Allows hooks and pipelines for integrating event triggers.

#### Technical Considerations

- Ensure decoupled triggers so event publishing doesn’t block content updates;
- Use versioned APIs for external systems consuming content;
- Apply performance tuning and caching for scalable API response times.

## 5 Cross-Cutting Concerns

In the context of evolving Piranha CMS to support event-driven communication and secure external integrations, managing cross-cutting concerns is critical to ensure the system is not only functional but also secure, maintainable, and observable. These concerns span multiple architectural layers and influence both runtime behavior and system resilience.

Below are the primary cross-cutting concerns relevant to Scenario 3, along with our strategies for addressing them:

### 5.1 Security

Security is the most critical concern in this scenario. The system will expose endpoints for receiving and sending sensitive content-related events, potentially across organizational boundaries.

#### Strategies

- **Authentication and Authorization:** Implement token-based (e.g., JWT) authentication for external publishers and internal consumers. Role-based access control (RBAC) will restrict access to critical operations like subscription management or content updates;
- **Message Integrity and signing:** All messages exchanged between systems will be signed using HMAC or asymmetric encryption (e.g., RSA) to ensure tamper resistance. Verification is enforced at both ends before processing;
- **Key Management:** Admin users can configure and rotate keys through a secure UI. Key storage will be encrypted and auditable;
- **HTTPS and TLS Enforcement:** All communication channels must be encrypted via HTTPS. Self-signed certificates will be disallowed for production environments;
- **Replay Attack Protection:** Timestamps will be included in message payloads to avoid replay attacks. Messages with expired timestamps will be rejected.

### 5.2 Logging and Auditing

Observability is key in an event-driven system, particularly when dealing with asynchronous workflows and third-party integrations. Logs are essential for debugging, auditing, and security analysis.

#### Strategies

- **Structured Logging:** Use structured logging frameworks (e.g. Serilog) to emit logs in a consistent JSON format for easy parsing by log aggregators (e.g., ELK stack, Seq);
- **Correlation IDs:** Every event or API call will carry a correlation ID to trace its life-cycle across services, improving diagnostics and incident response;

- **Audit Trails:** Sensitive actions (e.g., publishing events, accepting inbound messages, key rotation) will be logged in an immutable audit log;
- **Log Levels and filtering:** Logging granularity will be configurable. Sensitive data will be masked from logs, and error logs will include contextual information.

## 5.3 Error Handling and Resilience

Event-driven systems must handle failures gracefully to avoid cascading issues and data loss, especially with asynchronous message delivery.

### Strategies

- **Retry Policies:** Failed outbound or inbound event processing will trigger exponential back-off retries, with configurable limits.
- **Unsent Messages DB:** After retry limits are reached, failed messages will be moved to a database.
- **Fallback Handlers:** Define fallback logic for common failure scenarios (e.g., timeouts from external services).

## 5.4 Observability and Monitoring

The system must offer real-time insights into event flow health, performance bottlenecks, and integration statuses.

### Strategies

- **Health Checks and Dashboards:** Expose health endpoints for integration with monitoring tools (e.g., Prometheus, Grafana). These include event queue sizes, and key rotation status.
- **Real-Time Metrics Collection:** Track metrics such as event throughput, processing latency, and success/failure rates.
- **Alerts and Notifications:** Threshold-based alerts (e.g., high growth in the unsent messages DB, repeated signature verification failures) will notify administrators for quick remediation.

## 5.5 Performance and Scalability

Asynchronous systems that rely on messaging can experience bursts of traffic or load spikes. Poorly designed systems may suffer from message bottlenecks, thread exhaustion, or delayed content updates.



## Strategies

- **Asynchronous Message Queues:** Implement queue-based architecture (e.g., RabbitMQ, Kafka) to decouple producers and consumers, ensuring smooth load balancing.
- **Back-pressure Mechanisms:** Introduce rate limiting and flow control to prevent overload on internal or external systems.

## 5.6 Compliance and Data Protection

When transmitting or processing content-related events, regulatory compliance with data protection laws (e.g., GDPR, HIPAA) becomes crucial.

### Strategies

- **Data Minimization:** Ensure that only necessary data is included in messages. Personally Identifiable Information (PII) should be redacted unless explicitly required.
- **Consent and Access Logging:** Record user consent and track how and when data is shared with third parties.
- **Retention Policies:** Define clear retention rules for logs, messages, and audit data to comply with regulations and reduce storage costs.

## 6 Proposed Architecture and Roadmap

The system architecture is based on Piranha CMS, which serves as the foundation for managing core content and operations. The architecture then incorporates multiple components to implement an asynchronous publish-subscribe (pub-sub) system. To make this feasible, a webhook pattern is used by the clients, along with a message broker that stores and forwards events between Piranha and the system responsible for managing subscriptions.

To arrive at the final architecture, we followed an iterative process based on the Attribute-Driven Design (ADD) method. We began by designing the components necessary for sending events to external clients, focusing on the webhook delivery mechanism. In the next iteration, we extended the architecture to support the internal publication of events from the CMS, ensuring reliable and scalable event generation and distribution. Finally, we addressed the security requirements, particularly the management of API keys and access control, with emphasis on the role of the administrator who oversees key generation and subscription permissions.

Ultimately, we have developed an architecture consisting of eight components that work together to enable the event-driven CMS. In this section, we will explain each component in detail, along with their respective interfaces.

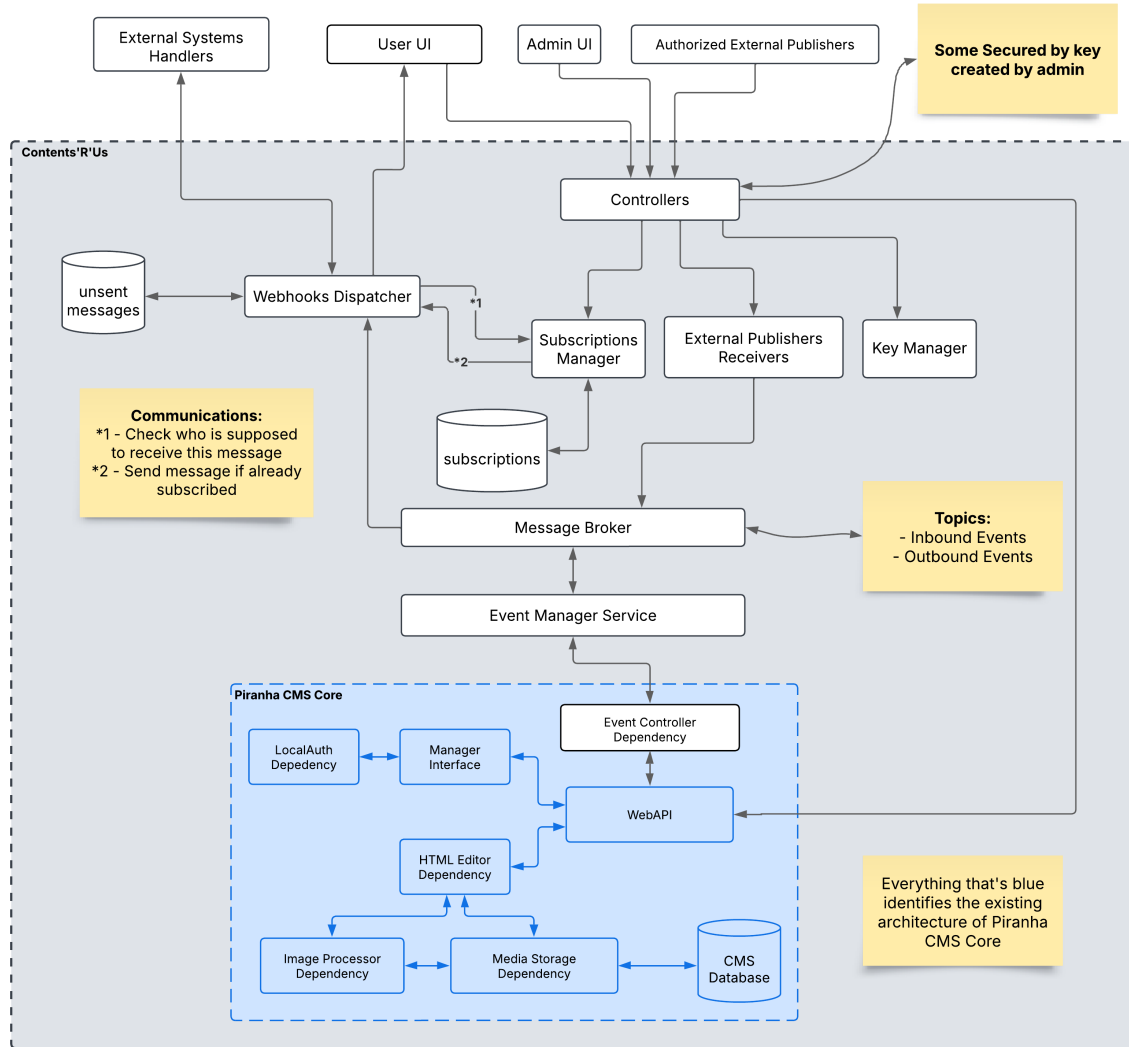


Figure 5: Proposed architecture

## 6.1 The Components and their responsibilities

The architecture comprises several essential components necessary for the system to operate effectively, as illustrated in the image above (Figure 5). These components are outlined as follows:

- Controllers
- Webhooks Dispatcher
- Subscriptions Manager
- Publish Receiver
- Message Broker
- Event Manager Service
- Piranha CMS Core

- Key Manager

### **6.1.1 Controllers**

The controllers are the entry points where requests arrive in the system. They perform security verification based on the type of request and forward the information to the appropriate service for processing.

### **6.1.2 Webhooks Dispatcher**

The Webhooks Dispatcher is the component that ensures the communication of events to the clients subscribed or that need to receive information from the CMS. It ensures that events generated within Piranha CMS are communicated to external systems or subscribers. It listens for events created in the Event Manager Service that are stored in the message broker. The dispatcher works by receiving event notifications, processing them, and sending them to the clients based on the subscriptions stored.

### **6.1.3 Subscriptions Manager**

This component is responsible for managing relations between events and clients. It saves and queries over a list of active subscriptions, mapping events to multiple clients that have the interest of receiving notifications for the events they subscribed to. It ensures that only relevant information is given to the clients. It communicates closely with the Webhook Dispatcher and also handles the logic of creating new subscriptions and sending events that trigger the delivery of unsent events when an already registered client registers again. It also sends events to the Webhook Dispatcher to delete unread messages from a client that does not register.

### **6.1.4 Publish Receiver**

The Publish Receiver is the component that receives event messages from external sources or other components in the system. It is ready to accept incoming events that need to be processed and published by the system. The Publish Receiver publishes events to the message broker, and its primary responsibility is to ensure that events are delivered to the appropriate component for further processing. It plays a key role in integrating external event sources such as updates and posts into the Piranha CMS ecosystem and ensures that the system remains reactive to external triggers.

### **6.1.5 Message Broker**

The Message Broker is the central component that makes the infrastructure more scalable by allowing multiple instances of a single component. It facilitates communication between various services in an event-driven manner. It serves as a queue of messages and events for multiple services to consume. The Message Broker decouples components, allowing them to communicate asynchronously. This supports message persistence and enables scalability by distributing messages across multiple consumers.

### 6.1.6 Event Manager Service

The Event Manager Service is the core component responsible for managing the flow of events throughout the system. It monitors changes within the Piranha CMS Core (such as content creation, updates, or deletions) and triggers corresponding events based on these changes. Once an event is generated, it is placed in the message broker, ensuring that all other services are notified of the event. The Event Manager Service acts as the controller of event generation, ensuring that events are triggered at the right time and in the correct format, depending on the information received by the CMS.

### 6.1.7 Piranha CMS Core

At the heart of the event-driven architecture lies the Piranha CMS Core. This is the central system that powers the content management functionalities, such as creating, updating, and deleting content. When a change is made in the CMS, a trigger can be initiated through the Event Manager Service. The events are then propagated throughout the system, reaching components like the Webhook Dispatcher. The Piranha CMS Core is the origin of the events and is central to the operation of the event-driven system.

The CMS Core is composed of several key modules:

- **Manager Interface:** The administrative interface used for managing content and configuration. It serves as the main interaction point for users and connects to subsystems like the WebAPI, HTML Editor and manage keys.
- **WebAPI:** Provides programmatic access to CMS functionalities via HTTP. It connects with the Event Controller to notify the event system of content changes and receive content from Event Controller.
- **Event Controller Dependency:** Captures and forwards internal CMS events to the Event Manager Service, acting as the bridge between traditional CMS operations and the event-driven infrastructure.
- **LocalAuth Dependency:** Manages authentication and authorization within the CMS, ensuring secure access to admin users.
- **HTML Editor Dependency:** Enables rich text content editing within the Manager Interface and supports integration with image and media resources.
- **Image Processor Dependency:** Responsible for processing images and integrates with the media storage system.
- **Media Storage Dependency:** Handles the storage and retrieval of uploaded media files and interacts directly with the CMS database.
- **CMS Database:** Stores all persistent data including content, media assets, and user credentials. It serves as the backbone of the CMS's data layer.

### 6.1.8 Key Manager

The Key Manager enables administrators to securely manage API keys used to authenticate requests to the Publish Receiver. It ensures that only authorized systems or users can publish events into the CMS infrastructure, enhancing security and access control across the event-driven ecosystem.

## 6.2 Communication/Interfaces Between Components

The communication between these components is organized as a flow that starts with content changes within the Piranha CMS Core and ends with the dissemination of events to interested parties. Here's how the components interact:

1. The **Piranha CMS Core** detects a change in the system (e.g., new content created) and triggers an event that is sent to the Event Manager Service.
2. The **Event Manager Service** listens for these changes and publishes them to the **Message Broker** where they will be consumed.
3. The **Message Broker** distributes the events to the entire system. It is part of two main flow one where the information goes to the PiranhaCMS and other where the information goes to the Webhooks Dispatcher.
4. The **Subscriptions Manager** filters the events, ensuring that only the subscribers receive the event this information is consumed by the Webhook Dispatcher. The Controllers allows clients and admins to interact with this module changing subscriptions.
5. The **Webhooks Dispatcher** consumes events from the message broker and sends the event to external systems based on the subscriptions present in the Subscription Manager.
6. The **Publish Receiver** can listen to incoming request to publish information into the system, this component sends events to the message broker.
7. The **Key Manager** can manage the keys generated by the admin and secure login with key verification in requests.

This architecture ensures that Piranha CMS is responsive and scalable, publish and subscribe content manager system. Each component plays a vital role in maintaining the overall structure and functionality of the system.

## 6.3 Road Map

The development of the event-driven extension for Piranha CMS will follow a phased approach to ensure proper integration, testing, and functionality. The following are the three principal features to be built:

1. **Event Publishing Infrastructure**  
Development of the internal event flow, including the Event Manager Service and integration with a Message Broker. This step ensures that events from the CMS Core are properly captured, and that events originating from authorized publishing clients can reach the CMS.
2. **Webhook Delivery System**  
Construction of the Webhooks Dispatcher, responsible for delivering event payloads to external subscribers. It will check for subscriptions created in the Subscription Management Module.

### 3. **Subscription Management Module**

Implementation of a robust Subscriptions Manager capable of registering, updating, and deleting client subscriptions. This module will also handle logic related to event filtering and mapping subscribers to specific events. This is foundational to enabling selective notification in the publish-subscribe model.

Each feature will be developed iteratively and validated against real CMS interactions to ensure full compatibility and performance under expected system loads.

## References

- [1] Microsoft. *ASP.NET WebHooks Overview*. <https://learn.microsoft.com/en-us/aspnet/webhooks/>. Accessed: 2025-05-11. n.d.