universidade de aveiro deti departamento de eletrónica, telecomunicações e informática

Software Architectures

EVOLVING PIRANHA CMS FOR "CONTENTS'R'US"

University of Aveiro
Master's Degree in Software Engineering

Roberto Castro - 107133
Tiago Gomes - 108307
Sara Almeida - 108796
Duarte Santos - 124376

Contents

This project integrates the final assignment for the Software Architectures course, which has as its main goal to apply architectural design principles learned in class to evolve an existing software system, with **performance** and **scalability** in mind as key top priorities. For this, we had to choose a scenario from a set of proposed evolution paths for Piranha CMS, an open-source **Content Management System**, and then detail in a comprehensive manner the architectural approach taken to address the selected challenge.

We chose **Scenario 2: Editorial Workflows & Content Approvals**, where the objective is to idealize and design a system that extends **Piranha CMS's** capabilities to support advanced editorial workflows, enabling multi-stage content approvals and governance for large teams.

After a careful design process and architectural planning, the proposed solution was fully implemented, resulting in a functional, observable, and extensible editorial workflow system that aligns with the scenario's strategic goals. The complete implementation is available in the public repository: https://github.com/RobertoCastro391/AS_Project_2Scenario_Piranha.

1. System Analysis & Project Vision

1.1. Current State Analysis of Piranha CMS

Modular Foundation with Monolithic Constraints

Piranha CMS is an open-source content management platform built on .NET that provides a modular and extensible architecture. Its foundation leverages dependency injection and a plugin-friendly design, offering flexibility for integrating new features. It is also database-agnostic, thanks to its use of Entity Framework Core, and supports cross-platform deployment. These characteristics make it an attractive base for evolving into a system that can support advanced editorial workflows.

However, despite its modular approach, much of the core functionality still runs in-process, making it closer to a monolithic architecture in practice. While its modularity allows for the addition of features like editorial workflows, it lacks out-of-the-box support for distributed components or message bus integration—elements that are typically critical for orchestrating complex, asynchronous editorial processes at scale. Furthermore, while performance is generally acceptable, there are no explicit optimizations aimed at high-concurrency environments or editorial workloads involving many users collaborating in real time.

Existing Role and Content Models

Piranha CMS includes a basic system of permission policies (such as Admin and Editor) and content states (implicitly Draft and Published), offering some foundation for editorial control. It provides core functionality for content creation, editing, and publication, with built-in authentication and authorization features. This already supports small editorial teams working with straightforward publishing needs.

Yet, when assessed against the demands of complex editorial workflows, significant limitations emerge. The current system does not support custom content states such as "In Review," "Approved," or "Legal Check," nor does it provide configurable, multi-stage approval flows. There is no native concept of hierarchical role-based approval—any user with editing rights can publish content, with no built-in separation of duties or staged responsibilities. Likewise, handoffs between roles (e.g., from a writer to a reviewer, then to legal, and finally to a publisher) are not orchestrated in a structured way.

Limited Workflow and Collaboration Capabilities

Another gap is in team collaboration features. Piranha CMS lacks mechanisms such as inline comments on content, task assignment, or notifications, which are essential for large editorial teams to coordinate efficiently. There's also no built-in audit trail or detailed versioning system that tracks the progression of content through its editorial lifecycle. This absence makes it difficult to maintain transparency or accountability in content operations—a critical requirement in regulated or high-complexity publishing environments.

Extensibility and Integration Challenges

While Piranha CMS allows extensions through plugins, it does not come with a built-in workflow engine or support for message queues and event-driven architecture. This limits its ability to integrate with external systems (e.g., notification services, approval tracking tools, or analytics platforms) or to scale editorial logic across microservices. These custom modules may be implemented to fill these gaps, but doing so in full would require significant architectural effort and expertise.

1.2. Vision & Strategic Goals

As said before, Piranha CMS already has a **modular layout**, but it lacks native support for multi-stage editorial workflows, state-based permissions, and traceability features. The next evolution will aim to make the system a **full-fledged editorial solution**, able to serve organizations with complex content creation, review, and approval processes.

The main goal is to position Piranha CMS as a **leading editorial management platform**, offering advanced approval workflows, robust editorial governance, and an optimized user experience for large, distributed teams.

This vision seeks to meet demands for **rigorous editorial processes**, full control over the content lifecycle and compliance with demanding internal policies.

Strategic Objectives

- **Multi-Stage and Configurable Workflows:** Develop editorial cycles composed of configurable stages (e.g.: Draft, Editorial Review, Legal Approval, Publication), allowing each organization to adapt workflows to their own rules.
- **Granular Permissions Based on States and Transitions:** Associate permissions with the content state and transition so that only authorized users can take critical steps (i.e., approval from Review to Final Approval)
- **Process Automation and Notifications:** Automate the exchange of content between roles (authors, reviewers, approvers), with automated notifications, to ensure smoothness and speed in the editorial process.
- **Intuitive User Experience:** Provide simple, information-rich interfaces from which users have an easy-to-understand view of content status, can readily perform workflow activities, and obtain revision histories naturally.
- **Performance and Scalability:** Test at high maximum number of editorials under heavy load and have the system in place when architecturally ready to scale horizontally when the team size and content volume grow.
- **Traceability and Complete Auditing:** Store all relevant actions (edits, approvals, rejections, publications) to ensure complete transparency and allow for auditing when needed.
- **Enhanced Security:** Put in place very strict contextualized access controls to protect content and allow only duly authorized users to proceed at each stage of the editorial
- cycle.

Priority Quality Attributes

- **Performance:** Fast and stable responses even in high competition scenarios.
- **Scalability:** Support continued growth without the need for re-engineering architecture.
- **Reliability:** Providing consistency in content states and transitions.
- **Auditability:** Immutable and searchable history of all meaningful actions taken.

- **Security:** Stringent separation of duties followed at any point of the workflow.
- **Usability:** User convenience on accessible interface accessibility which is simplified for editorial teams possessing different levels of technological expertise.

These were to close a gap and bring the system in line with editorial management best practices. The development is intended to address not just functional needs of intricate workflows but also that the solution is solid, dependable, secure, and positioned to grow without loss of performance or usability.

2. Architectural Design Methodology

2.1. Attribute-Driven-Design (ADD)

We chose the **Attribute-Driven-Design (ADD)** for Architectural Design Methodology. The project's scenario places strong emphasis on achieving specific quality attributes (performance, scalability and reliability) and ADD focuses precisely on defining quality attributes that will be addressed early and directly in the design process and will guide all architectural decisions, which aligns perfectly with the strategic goals. Besides the **quality attributes focus** being the center of our scenario's requirements, there are more reasons that have driven our decision:

- **Iterative and Systematic Approach:** ADD allows for incremental design decisions, allowing us to tackle the complex editorial workflow system in manageable iterations which aligns well with evolving an existing system.

- **Modularity and Refinement:** The methodology supports breaking down the system into components, which is beneficial since the existing modular architecture of Piranha (as seen in the folder structure) provides a good starting point for our evolution process.

- **Scenario-Driven:** ADD uses real use-case scenarios to test design decisions against requirements, enabling us to validate our design choices against editorial workflow needs, like role-based interactions.

In comparison with other approaches:

- *TOGAF ADM* is an enterprise architecture methodology that involves business, data, application, and technology architecture and is often too heavyweight for single-system software design.
- *Architecture-Centric Design Method (ACDM)* also has an iterative nature but is more general-purpose lacking a specific focus on quality attributes, which is critical in our case.

2.1.1. Application of ADD Methodology

As mentioned above, ADD follows an iterative process and in short goes through the following steps:

- **Identifying Architectural Drivers -** inputs like design goals, functional requirements, the key quality attributes, constraints, etc.;
- **Decomposing the System** into modules and components;
- **Selecting architectural patterns, tactics** (like caching, role-based access control, etc.) and **allocating functionalities**;
- **Designing Interfaces** between components;
- **Analysing** continuously how well the architecture satisfies the drives and **refine** as needed;

Given these steps, in the context of this project, ADD was applied as follows:

1. **Identify Architectural Drivers**
   We started by analyzing the business goals, key quality attributes and technical requirements of this editorial workflow scenario. This analysis provided a starting point for our enhancement strategy and it resulted in identifying the following architectural drivers, which can be deepen throughout the implementation with a more comprehensive practical knowledge of the core components and interfaces of the project:

   - **Business Drivers:**
     - **i.** Support multi-stage editorial workflows;
     - **ii.** Enable role-based permissions and content approvals;
     - **iii.** Ensure scalability, high performance and reliability of the system.

   - **Functional Requirements:**

ID	Actor	Functional Requirement
FR1	Administrator	Manage users, roles, and associated permissions for workflow stages.
FR2	Administrator	Set up notification rules and preferences.
FR3	Administrator	Access full audit logs and version histories of content items.
FR4	Administrator	Modify permissions without system downtime.
FR5	Administrator	Monitor the status of all active workflow instances via the Workflow Monitoring Dashboard.

FR6	Editor (Content Creator)	Create and submit content items for editorial review.
FR7	Editor (Content Creator)	Edit content that is in allowed states (e.g., Draft).
FR8	Editor (Content Creator)	View the current workflow status of submitted content.
FR9	Editor (Content Creator)	Receive notifications when content is reviewed, approved, or sent back for revisions.
FR10	Editor (Content Creator)	View state history and audit logs of their own content items.
FR11	Reviewer (Editorial Role)	View assigned content items awaiting review.
FR12	Reviewer (Editorial Role)	Approve or reject content, with optional comments.
FR13	Reviewer (Editorial Role)	Receive notifications for new review tasks and workflow updates.
FR14	Reviewer (Editorial Role)	View workflow history of reviewed content items.
FR15	Legal Reviewer	Access content requiring legal approval.
FR16	Legal Reviewer	Approve or flag content for compliance issues.
FR17	Legal Reviewer	Provide detailed comments/feedback on compliance concerns.
FR18	Legal Reviewer	Receive notifications for legal review tasks and workflow changes.
FR19	Legal Reviewer	Track legal review decisions in the audit history.
FR20	Approver/Publisher	View content that is ready for final approval.
FR21	Approver/Publisher	Approve content for publication or request final revisions.
FR22	Approver/Publisher	Publish content upon final approval.
FR23	Approver/Publisher	Receive notifications when content reaches the final approval stage.
FR24	Approver/Publisher	Access publication history and audit logs.

- **Quality Attributes:**
  The quality attributes we consider crucial were already mentioned above, but in this subsection, as the ADD methodology supports, we prioritize them and detail a scenario for each quality attribute.

Priority	Quality Attribute	Non-Functional Requirements	Quality Attribute Scenario
1	Performance	The system must process workflow transitions within 2 seconds under normal load; UI actions should respond instantly (<1 sec).	**Stimulus Source:** Editor **Stimulus:** Submits bulk content for review **Artifact:** Workflow Engine **Environment:** Normal usage **Response:** System queues and processes the requests **Response Measure:** 95% of submissions processed within 2 seconds
2	Scalability	The system must support horizontal scaling to handle growth in users and content, ensuring stable performance even with 500+ concurrent operations.	**Stimulus Source:** Load testing tool **Stimulus:** 500 concurrent content operations **Artifact:** Workflow Engine & DB **Environment:** Peak load **Response:** System completes all operations **Response Measure:** 95% complete within 3s; no degradation observed
3	Security	Enforce strict role-based access control (RBAC); all sensitive actions (e.g., approval, publishing) must be logged and accessible only to authorized users.	**Stimulus Source:** Unauthorized user **Stimulus:** Attempts to approve restricted content **Artifact:** User & Permission Modules **Environment:** Anytime **Response:** Action is denied and logged **Response Measure:** 100% of unauthorized actions blocked and logged within 1s
4	Reliability	Ensure workflow consistency—state transitions must be atomic and fail-safe; the system must maintain 99.9% uptime with automatic recovery.	**Stimulus Source:** Editor **Stimulus:** Workflow transition during DB outage **Artifact:** Workflow Engine **Environment:** Network failure **Response:** Transaction rollbacks; system recovers **Response Measure:** Zero data corruption; full recovery logged
5	Usability	Provide an intuitive, user-friendly interface;	**Stimulus Source:** Admin **Stimulus:** Opens workflow dashboard

		workflow tasks and statuses must be easily visible and actionable with minimal training required.	**Artifact:** Admin UI **Environment:** Normal use **Response:** UI loads content states and other information **Response Measure:** Dashboard loads fully in ≤1.5 seconds
6	Auditability	Maintain immutable logs of all editorial actions (approvals, rejections, edits).	**Stimulus Source:** Admin **Stimulus:** Requests log for content item **Artifact:** Audit Log Service **Environment:** Normal operation **Response:** Log returned with full action history **Response Measure:** Complete logs displayed within 5s; entries immutable

- **Constraints:**
  - **i.** Must integrate seamlessly with the existing Piranha CMS architecture;
  - **ii.** Should leverage .NET-based technologies and fit within the current tech stack.

2. **Decompose the System**

We identified and broke down the system into **modules** that map directly to the business needs and quality attributes. In the two iterations we did over the architecture design process, this step was the one where there was a significant change that shaped the final decisions of our last architecture.

Iteration 1: Combined User & Permission Logic

Initially, we managed users and their roles/permissions in a single module. This aligned with the base structure of Piranha CMS but became increasingly complex as our needs evolved (e.g., stage-based permissions, task-specific authorizations).

Iteration 2: Decoupled Permission and User Modules

We later distinguished them for better separation of concerns, modifiability, and security traceability.

- The User Module now focuses purely on user data, credentials, preferences, and teams.
- The Permission Module enforces contextual RBAC, maintains role-permission mappings, and handles access checks for workflows and content stages.

Module	Responsibility
Workflow Engine Module	Manages workflows, including stages, transitions, and state consistency.
Content Service Module	Handles content items and their binding to workflows; manages editorial states.
User Module	Manages users, profiles, credentials, teams, and notification preferences.
Permission Module	Manages roles, permissions, and access control policies; enforces RBAC for workflow actions.
Notification Module	Manages delivery of alerts and notifications across channels; handles queueing and retries.

From here we will choose which module to decompose and focus for each iteration.

3. **Select Architectural Patterns and Tactics**

For each prioritized quality attribute, we selected design tactics and patterns to address specific needs:

Quality Attribute	Tactics/Patterns Applied	Why
Performance	Caching	Ensures fast content status checks, smooth workflow transitions, and reduces database load.
Scalability	Modular Architecture, Horizontal Scaling Readiness	Supports growth by distributing load across multiple servers; decouples components for scalability.
Security	Role-Based Access Control (RBAC), Audit Trails	Enforces fine-grained access control and tracks sensitive editorial actions.
Reliability	Transactional Integrity, Failover Support	Ensures workflow state consistency even in case of failures; robust error handling.
Usability	Responsive UI	Ensures user-friendly, efficient workflow navigation for editors and reviewers.

Auditability	Immutable Logs, Versioning	Maintains tamper-proof history of content actions to meet compliance and traceability requirements.

These patterns are mapped directly to our **Quality Attribute Scenarios** to ensure full alignment between requirements and design.

4. Design Interfaces

We defined **explicit APIs** and interfaces for clean communication between modules:

Module	Key Interfaces
Workflow Engine	*CreateWorkflow(workflowDefinition), StartWorkflow(contentItemId), AdvanceWorkflow(workflowInstanceId, action)*
User	*GetUser(userId), UpdateUserProfile(userId, profileData), GetUserTeams(userId)*
Permission	*GetUserRoles(userId), CheckPermission(userId, action, contentState), AssignRole(userId, role)*
Notification	*SendNotification(userId, message), ConfigureNotificationPreferences(userId), GetNotificationHistory(userId)*
Content Service	*GetContent(contentItemId), BindWorkflow(contentItemId, workflowInstanceId)*

UI extension points are also designed for the **Workflow Dashboard** to ensure a smooth user experience without tight coupling to backend logic.

5. Analyse and Refine

We reviewed the architecture we will present below to verify:

- All **quality attribute scenarios are covered**.
- The design maintains **alignment with the Piranha CMS architecture**.
- Patterns and tactics are feasible within the project's **constraints and future growth expectations**.

However, and in alignment with the ADD's principles, this architecture is **iterative and evolvable.** As requirements change or new challenges emerge during implementation, we will **apply new ADD cycles** to refine and extend the design without compromising quality, and always with the quality attributes in mind.

3. Identification and Modeling of Domains

To mature Piranha CMS towards a fully robust editing solution, we have resorted to **Domain-Driven Design (DDD)** principles. This means the business logic can be structured into cohesive domains with bounded responsibilities and clear separation from the most complex parts of the system. Below, we describe the identified domains, detailing their main elements and their interrelationships.

3.1. Content Domain

This is the **core** of the system and governs all editorial content, i.e., pages and articles. The domain is structured around the **Aggregate Root** *ContentItem*, which gives coherence and contains the related entities such as *ContentVersion* and *ContentState*. Each item holds key properties (title, body, metadata such as tags and categories), along with a version hierarchy that keeps track of all changes during the editorial process.

Furthermore, each content item has an editorial status (e.g., In Draft, In Review, Approved, Published), governing the allowable actions and targeted users for each stage. For instance, a content item cannot be published if all the necessary review stages have not been done.

Key elements:

- **Entity (Aggregate Root):** *ContentItem* - represents the editorial item itself.
- **Entity:** *ContentVersion* - stores historical versions for auditing and change tracking.
- **Value Object:** *ContentState* - defines the current editorial state of the content.

The model ensures consistency by ensuring that any modification to the content generates a new version and respects the state transitions established.

3.2 Workflow Domain

This domain serves as the organization's assigning editorial flow in the life cycle. Hence, it includes a workflow to stipulate how content moves from creation to publication. It is modeled on the **Aggregate Root** *Workflow* that includes associated entities like *WorkflowStage*, *WorkflowTransition*, and *WorkflowAssignment*. The various stages of the workflow determine rules and specific requirements. For example, the editorial process may be composed of steps such as Editorial Review, Legal Review, and Final Approval.

Each stage is concerned with defining who shall have rights to act on that stage so that its responsibilities may be clearly identified and to ensure proper structure to the process. Transition rules will agonize the workflow as to when and how content may enter the next stage in order to make sure each step is performed so that none is skip and bypass.

Key elements:

- **Entity (Aggregate Root):** *Workflow* – defines the overall editorial process and maintains control over stages and transitions.
- **Entity:** *WorkflowStage* - each individual stage, with its characteristics.

- **Entity:** *WorkflowTransition* - rules that control the transition from one stage to another.
- **Entity**: *WorkflowAssignment* - linking users/roles and specific stages.

This domain is heavily integrated with the Content Domain, for every **ContentItem** is related to a **Workflow** that governs the passage of that particular ContentItem through the editorial lifecycle. Through the enforcement of responsibilities and transition logic of stages, the workflow thoroughly enforces editorial governance.

3.3 Permission Domain

This domain governs access control and security throughout the editorial process, defining who can perform which actions at each stage. It is structured around the **Aggregate Root** *Role*, which encapsulates related entities like *Permission* and PermissionAssignment. While scanty, role classifications such as Editor or Reviewer exist, this model would make sure that the role chosen has been explicitly validated against the permitted actions for a given content state and workflow stage.

Merely assigning a user to be an Editor or a Reviewer is not enough; the system should be able to determine if and when the role can carry out a specific action (e.g., Approve Content or Edit Draft) in some workflow context. Such fine-grained control is most demanding whenever strict governance has to be well promulgated, especially in an editorial environment characterized by multiple stages of approval and responsibility.

Key elements:

- **Entity (Aggregate Root):** *Role* – specifies editorial roles within the system (e.g., Author, Editor, Legal Reviewer).
- **Entity:** *Permission* – defines the actions that are permitted (e.g., Approve Content, Edit Draft).
- **Entity:** *PermissionAssignment* – maps roles to permissions, potentially limited by scope (e.g., only within a specific section or workflow stage).

This area is strongly related to the Workflow Domain as permissions themselves control who or what roles are able or not able to carry out an activity on transitions across workflow phases, to enable safe and compliant editorial progression.

3.4 Notification Domain

This domain handles all system-level communication related to editorial workflows. It ensures that users and teams are kept informed of relevant events (e.g., content creation, submission for review, approval, rejection) through system-generated notifications.

The domain is centered on the **Aggregate Root** *Notification* and its related entities, providing a consistent mechanism to trigger, store, and deliver notifications across the system.

Key elements:

- **Entity (Aggregate Root):** *Notification* - represents an event-driven message to be delivered to a user or team.
- **Entity:** *NotificationChannel* - defines the delivery mechanism (e.g., email, in-app alert, push notification).
- **Value Object:** *NotificationContent* - encapsulates the message body, metadata, and context.

The Notification Domain integrates with the Workflow Domain to listen for state transitions and with the User Domain to determine recipients and preferences. This ensures timely and targeted communication throughout the editorial process.

3.5 User Domain

This domain manages the users who interact with the system, covering not only authentication and credentials but also their editorial profiles and organizational structure. It is built around the **Aggregate Root** *User*, which maintains associated components such as *Team* and *Preferences*. Even though Piranha CMS already comes with basic user management, this model enlarges the scope to accommodate editorial workflows, mainly on the basis of group assignments and individual customization.

Apart from running and managing individual users, the domain supports creating editorial teams so that groups of users performing similar editorial functions can be grouped together. For example, a Legal Review Team may consist of two or more users who share the same responsibilities within the approval chain. Setting up such a group will allow tasks to be assigned more easily, as well as managing workflows, by applying permissions and actions to the team level when appropriate.

Key elements:

- **Entity (Aggregate Root)**: *User* – stores user details, access credentials, and editorial profile information.
- **Entity**: *Team* – groups users with shared editorial responsibilities to streamline task assignments.
- **Value Object:** *Preferences* – defines user-specific settings, such as notification preferences or interface customization.

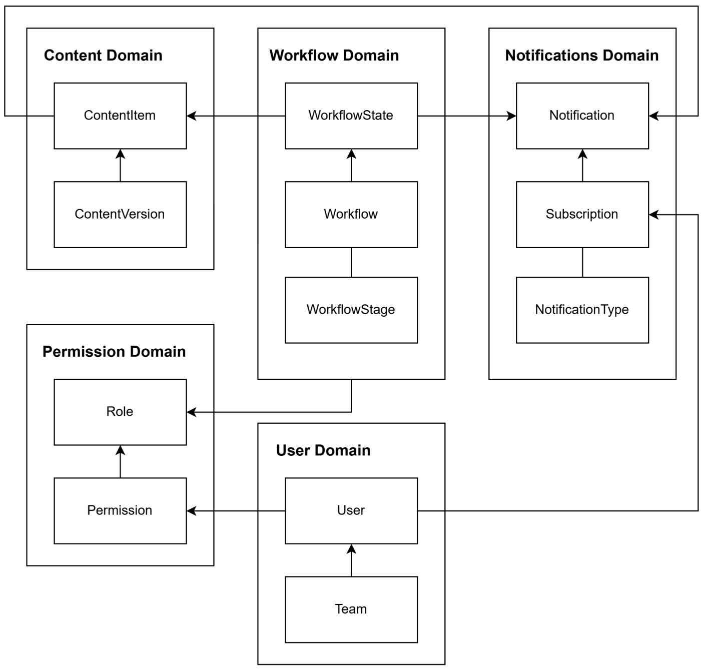
This domain is directly connected to both the Workflow and Permission domains. Users and teams are assigned to workflow stages through **WorkflowAssignment**, and their actions are regulated by the permissions framework to ensure that each step of the editorial process is handled by authorized individuals or groups.

3.6 Relationships Between Domains

Top-level domains in this model are not isolated; rather, they interact closely in support of a cohesive and controlled editorial workflow. Each domain is responsible for a particular aspect, but they share common consistencies on security and flexibility throughout the content lifecycle.

- Each **ContentItem** (Content Domain) gets linked to a **Workflow** (Workflow Domain), which governs the processing of editorial steps into various stages. The editorial state of said content is thus made up by the current entry into the workflow.
- The **WorkflowStage** then determines fine-grained controls on the **Roles** (Permission Domain) that are allowed to provide services at each step.
- **Users** and **Teams** (User Domain) then become assigned through **WorkflowAssignment** to certain workflow stages, and those actions are checked against Permission to ensure editorial policy is followed.
- The **Permission** entities control and enforce governance and security across all transitions and actions, thereby ensuring the separation of duties and strict access control.
- Versioning and auditing features within **ContentItem** and **ContentVersion** ensure traceability and accountability: all changes and approvals may be tracked in translucent fashion.
- Notifications are triggered by workflow events and delivered to users, closing the feedback loop between content progress and editorial actors

Through the joint orchestration of these domains, their respective Aggregate Roots, and related entities, a strong and reliable infrastructure can be provided for editorial governance to occur, wherein content flows from creation to publication severely disguised and ready for total compliance.



3.7 Domain Classification

In the context of Domain-Driven Design (DDD), domains are typically classified into **Core Domains**, **Supporting Domains**, and **Generic Domains**. This categorization helps prioritize architectural efforts and clarify where business-critical complexity resides.

In our case, the following domains have been modeled:

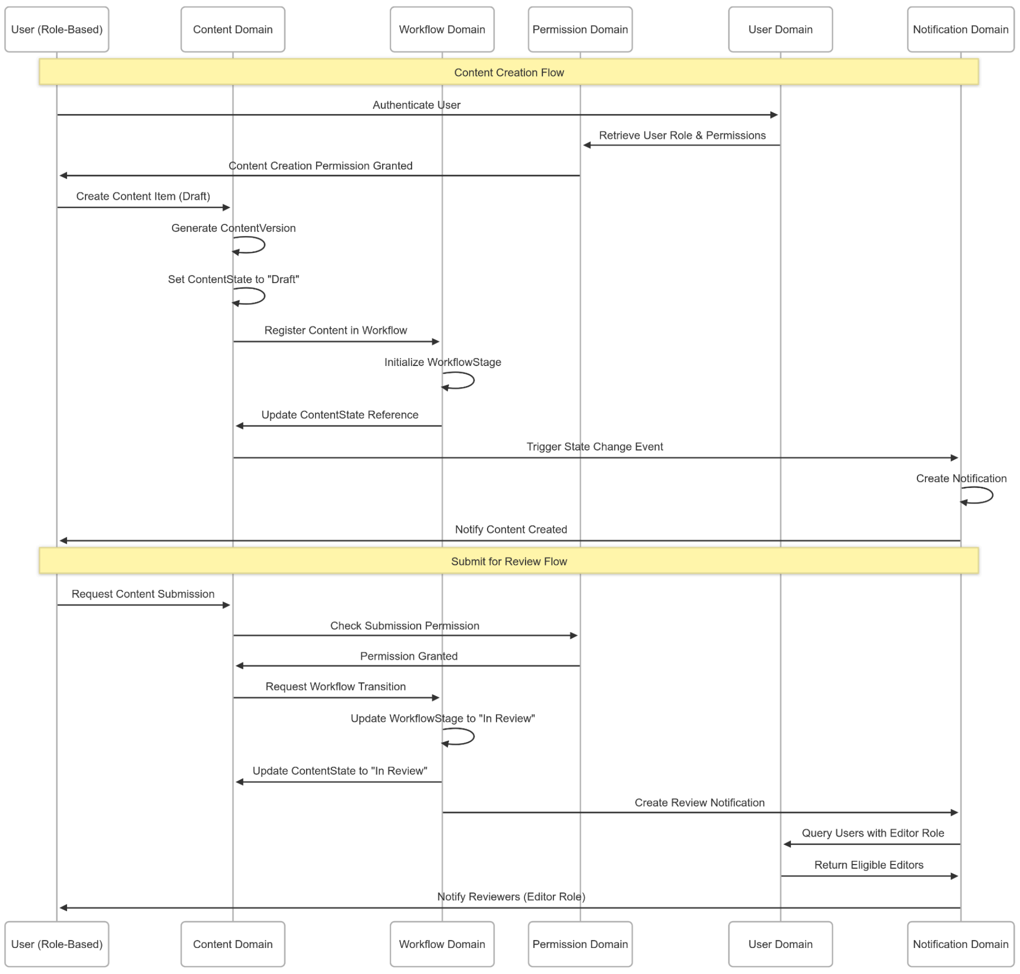
Domain	Classification	Rationale
Content Domain	Core Domain	Central to managing editorial content, versioning, and state — the backbone of the editorial process.
Workflow Domain	Core Domain	Defines and controls the editorial lifecycle, ensuring correct sequencing and governance of content approvals.
Permission Domain	Core Domain	Enforces fine-grained access control, guaranteeing that only authorized roles can act at specific workflow stages.
Notification Domain	Supporting Domain	Manages communication and alerts across the editorial process, ensuring users are informed of relevant actions and updates.
User Domain	Supporting Domain	While managing users and teams is foundational, its primary role is to support editorial workflows, not to embody business complexity on its own.

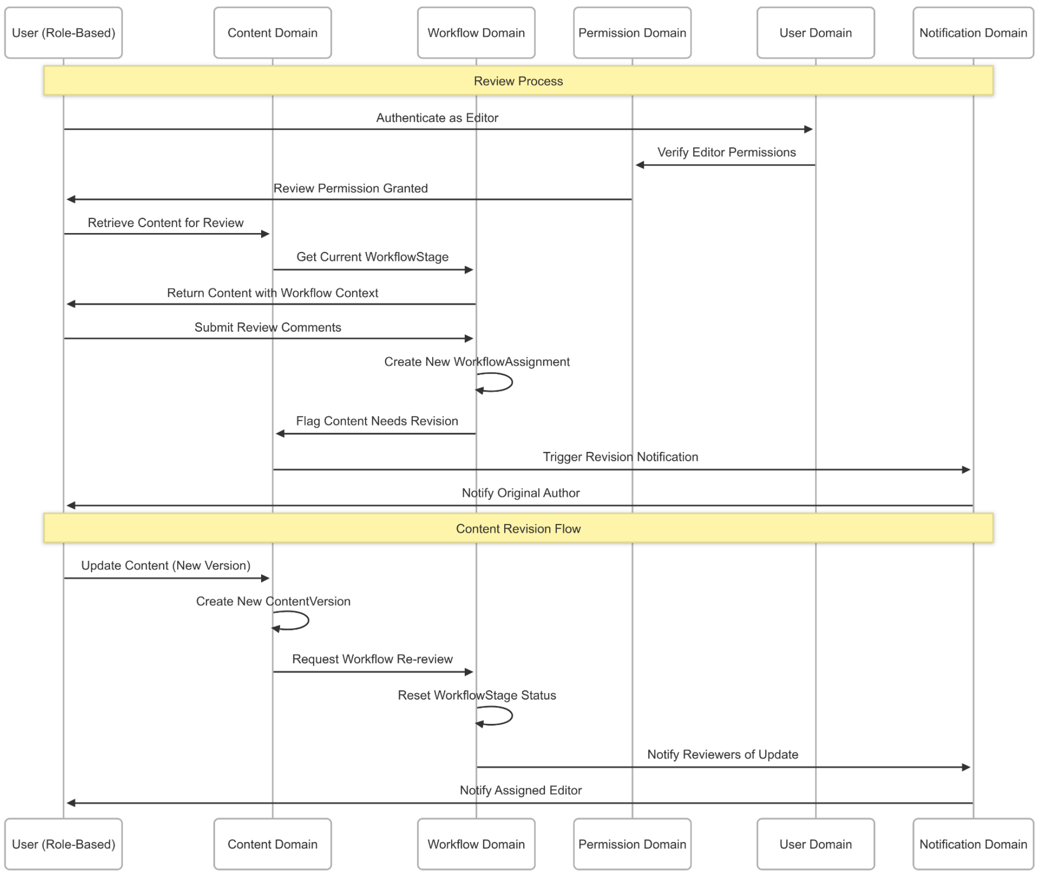
We have deliberately chosen to focus exclusively on these key domains, as they fully cover the critical business requirements for editorial workflows and approvals.

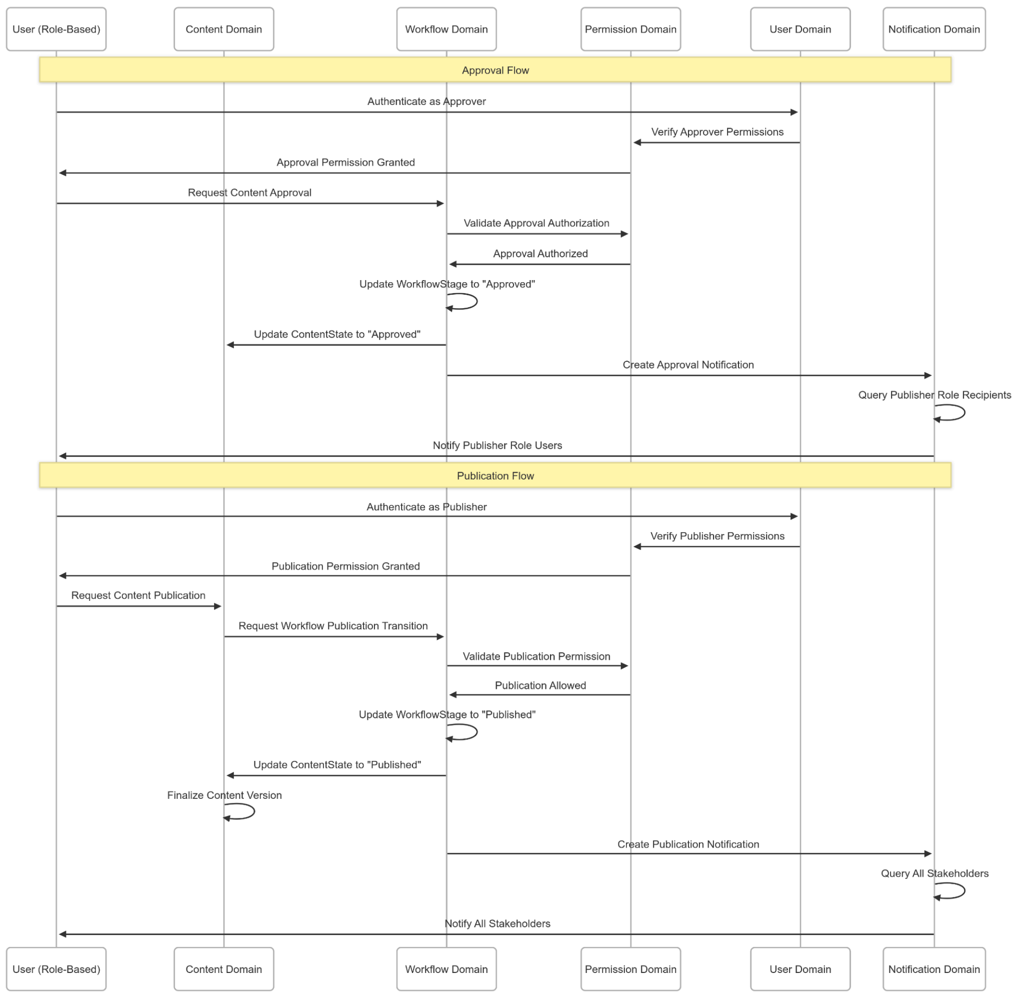
By prioritizing these domains, we ensure that the architectural effort is directed toward the areas with the highest business impact, while avoiding overengineering in peripheral concerns.

This targeted modeling approach provides a solid foundation for scalability, maintainability, and alignment with Domain-Driven Design (DDD) best practices, allowing the system to evolve gracefully as future business needs emerge.

3.8 Domain Sequence Diagram


User (Role-Based)
Content Domain
Workflow Domain
Permission Domain
User Domain
Notification Domain
Content Creation Flow
Authenticate User
Retrieve User Role & Permissions
Content Creation Permission Granted
Create Content Item (Draft)
Generate ContentVersion
Set ContentState to "Draft"
Register Content in Workflow
Initialize WorkflowStage
Update ContentState Reference
Trigger State Change Event
Create Notification
Notify Content Created
Submit for Review Flow
Request Content Submission
Check Submission Permission
Permission Granted
Request Workflow Transition
Update WorkflowStage to "In Review"
Update ContentState to "In Review"
Create Review Notification
Query Users with Editor Role
Return Eligible Editors
Notify Reviewers (Editor Role)


User (Role-Based)
Content Domain
Workflow Domain
Permission Domain
User Domain
Notification Domain
Review Process
Authenticate as Editor
Verify Editor Permissions
Review Permission Granted
Retrieve Content for Review
Get Current WorkflowStage
Return Content with Workflow Context
Submit Review Comments
Create New WorkflowAssignment
Flag Content Needs Revision
Trigger Revision Notification
Notify Original Author
Content Revision Flow
Update Content (New Version)
Create New ContentVersion
Request Workflow Re-review
Reset WorkflowStage Status
Notify Reviewers of Update
Notify Assigned Editor


User (Role-Based)
Content Domain
Workflow Domain
Permission Domain
User Domain
Notification Domain
Approval Flow
Authenticate as Approver
Verify Approver Permissions
Approval Permission Granted
Request Content Approval
Validate Approval Authorization
Approval Authorized
Update WorkflowStage to "Approved"
Update ContentState to "Approved"
Create Approval Notification
Query Publisher Role Recipients
Notify Publisher Role Users
Publication Flow
Authenticate as Publisher
Verify Publisher Permissions
Publication Permission Granted
Request Content Publication
Request Workflow Publication Transition
Validate Publication Permission
Publication Allowed
Update WorkflowStage to "Published"
Update ContentState to "Published"
Finalize Content Version
Create Publication Notification
Query All Stakeholders
Notify All Stakeholders

4. Cross-Cutting Concerns

Beyond the core domain functionalities, the architectural evolution required to support editorial workflows must also address a set of *cross-cutting concerns*—aspects that impact multiple layers and modules of the system. These include **Security**, **Auditing**, **Logging**, and **Performance**. Properly handling these concerns ensures that the system remains secure, traceable, reliable, scalable, and user-friendly even as complexity increases.

4.1 Security (Authentication, Authorization, and Access Governance)

Piranha CMS already integrates *ASP.NET Identity* for authentication and uses a claims based security authorization mechanism. Controllers such as ContentApiController, PageApiController, and PostApiController include attributes like [Authorize(Policy = Permission.Admin)], demonstrating a foundation for security. These roles are built within

Piranha CMS codebase with their Permissions System which allow them to create policies/claims and then roles with those policies/claims.

Enhancements Needed:

- **Fine-Grained Permissions:** The ability to authorize specific actions such as "approve," "reject," or "request changes" per user or role creating new policies/claims and apply them in the new features.
- **Stage-Based Content Security:** Enforcement of access rules based on the content's current workflow stage (e.g., only reviewers can edit content in "Review").
- **Governance and Accountability:** Track who performed sensitive actions, such as publishing or approving content, for compliance.

These measures will elevate security from static role checks to dynamic, stage-aware access governance, critical in collaborative editorial environments.

4.2 Auditing and Logging

Logging in Piranha CMS currently uses the standard *ILogger* infrastructure from *ASP.NET Core*, suitable for basic diagnostics and error reporting. However, it lacks an audit trail specific to editorial and workflow operations, making it difficult to trace who did what, when, and why during content processing.

Enhancements Needed:

- **Editorial Audit Trails:** Track every transition in the workflow, including who approved or rejected content, timestamps, and optional comments or justifications.
- **System-Wide Activity Logging:** Log user activities such as login, logout, permission changes, and role assignments to support governance and traceability.
- **Compliance-Grade Logging:** Ensure logs are immutable and organized for audits and regulatory reporting.

Logging Integration:

- Log critical events such as workflow initiation, state changes, failures, and completions.
- Include contextual metadata for all logs to support real-time monitoring and post-mortem diagnostics.

4.3 Performance and Scalability

Piranha CMS is designed with performance in mind, utilizing *Entity Framework Core* efficiently and applying in-memory caching in components like PageService and PostService. However, as editorial workflows introduce more state transitions, user roles, and conditional logic, performance bottlenecks may emerge—especially in large teams or high-content-volume environments.

Enhancements Needed:

- **Improved Caching:** Cache frequently accessed data such as content lists, workflow definitions, and items pending review to reduce database load.
- **Asynchronous Operations:** Offload non-critical or long-running processes (e.g., sending notifications) to background tasks.
- **Database Query Optimization:** Ensure content listing and filtering (e.g., by workflow stage, assigned user) are fast and responsive under load.

These improvements ensure the system can scale horizontally and deliver consistent performance under editorial load, even with rich workflow logic.

5. Proposed Architecture & Roadmap

5.1 High-Level Architecture Overview

The architecture we propose is based on Piranha CMS's own layered architecture, but we have added new modules to support editorial workflows, permission based roles, and robust auditing and logging. Although Piranha CMS's core is modular, still with monolithic execution, this new design introduces cross-cutting services and domain-specific engines that reinforce the separation of concerns and improve maintainability.

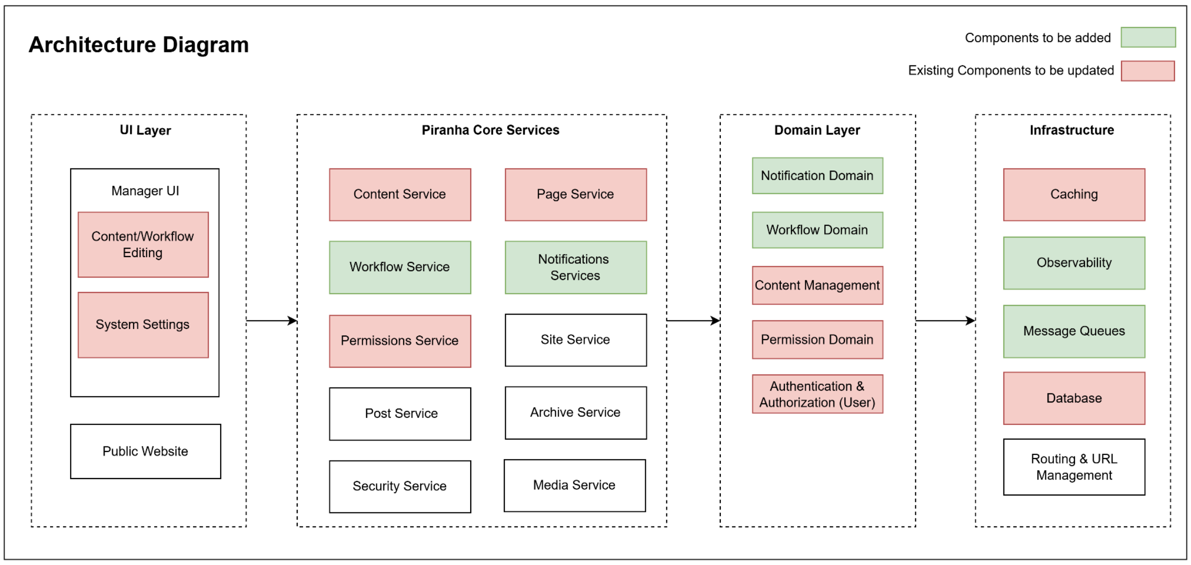
- **UI Layer:** The UI layer has editorial workflows exposed through the Manager UI, Content Editing UI, and WebAPI controllers. It should allow users to interact with the content editorially, approving content in real time.

- **Application Layer:** Facilitates business use cases by combining the core services (Content, Page, Post, Media) with the Workflow Engine Service, Audit Service, and Notification Service. This layer will be the middleware that connects UI to domain Logic so that these are segregated responsibilities.

- **Domain Layer:** The business logic and rules operate through the Content Model, Workflow Domain, Permission Domain, and Notification Domain. Each domain is responsible for its bounded context, Workflow Domain being in charge of proper content movement through the editorial process, and Permission Domain putting a strict enforcement behind access control.

- **Infrastructure Layer:** Provides concrete implementation of storage, identity management, caching, and communication channels. It notably includes modules for Audit Log Storage, Notification Channels and essential middleware for recording and handling exceptions.

This design will guarantee that the system remains extensible and robust, and that it can evolve into a distributed or microservices-compatible architecture in the future if needed.



5.2 Architectural Components

The Domain Layer contains the core business models and logic, including the Content Management, Workflow Domain, and Permission Domain. It ensures that business rules are centralized and reusable across services, improving maintainability and traceability.

- **Content Management Service:**

As said before, it is the same as the existing functionality of Piranha CMS, now with workflow-based state tracking capability. Content items are linked to workflows and their states are automatically updated based on editorial actions.

- **Workflow Engine Service:**

This is the core of our new architecture. Its main job is to manage workflow definitions, orchestrating the transitions between different steps, validating rules, and making sure everything stays consistent. It works hand-in-hand with the Content Service to ensure that the editorial process sticks to the configured steps without any issue.

- **Advanced Permissions System:**

This system enhances Piranha's authorization framework by adding state-aware permissions that specify who can take action at each step of a workflow. It accommodates both role-based and more detailed permission assignments, ensuring that governance is tight and effective.

- **Audit Service:**
  This service provides complete traceability by recording all editorial actions: edits, approvals, rejections, and publications. Stored in a dedicated audit log repository, this information allows administrators to track the entire history of any content item.

- **Notification Service:**

  A real-time messaging system that notifies users (via in-interface alerts or external channels like email) when content is assigned for review, approved, or rejected. This service follows the Observer/Publish-Subscribe standard, ensuring decoupled communication.

5.3 Key Features Selected

We selected the following key features. We chose these features not only to address critical gaps identified in the current system analysis, but also to strengthen Piranha CMS's editorial governance, traceability, and usability.

1. **Workflow for Publishing Content**

Implement a complete editorial workflow for content publishing, allowing you to create a multi-step approval cycle (e.g., Draft → Editorial Review → Legal Review → Publication). Each content must follow these steps, ensuring compliance and editorial quality. This feature is the core of the intended evolution, transforming PiranhaCMS into a tool capable of supporting complex, multi-step editorial workflows, essential for organizations with rigorous publishing processes.

2. **Permissions Control**

A high-level permissions control mechanism that links specific actions (approve, reject, edit, publish) to specific roles and the current status of the content. This permission control is also state-sensitive in that permissions will vary, depending on the status of the content. The mechanism helps ensure the establishment of good security and editorial governance by limiting important actions to only the properly authorized user, at each stage of the workflow.

3. **Real-Time Workflow Dashboard**

A real-time dashboard that displays the current status of all content in the workflow. Users can see what content is in progress, who is currently responsible, and what steps are left until publication. This dashboard will help improve visibility and operational efficiency by enabling easy and intuitive management of editorial content at scale.

4. **State History & Accountability**

Complete and detailed logging of all content state transitions. Shows who performed each action (approval, rejection, editing, publishing), when it was performed, and, where applicable, the associated comments or justifications. It offers full traceability and guarantees transparency at all stages of the editorial cycle. This record is essential for audits, legal compliance, and to ensure that the history of editorial decisions is always accessible.

5.4 Implementation Roadmap

The implementation is divided into **4 weekly phases**. The roadmap ensures a progressive and functional build-up of the editorial workflow system, focusing on **incremental delivery** of both backend services and user-facing features.

Week	Goals
Week 1	**Foundations for Workflow & Permissions** - Implement the **Content Workflow Model** with basic states (Draft, Review, Published). - Build the **initial Workflow Engine logic** (state machine for moving content through the defined steps). - Improve the **Permissions Model**, defining roles and mapping them to workflow states (data layer only). - Set up basic API endpoints to query workflow states for content items.
Week 2	**Workflow Execution & Permissions Enforcement** - Extend the **Workflow Engine** to support multi-step workflows (e.g., Editorial Review → Legal Review → Publication). - Implement **Permissions Enforcement**: integrate role-based & state-aware checks in the content workflow (e.g., only Legal can act on the Legal Review stage). - Build backend support for **State History logging** (record actions + timestamps + user).
Week 3	**Dashboard & Accountability** - Develop the **Real-Time Workflow Dashboard UI**: list content items, show current status, next required actions. - Test **workflow transitions with permission validation** end-to-end (from Draft to Published).
Week 4	**Finalization & UX Enhancements** - Polish the **State History & Accountability view**: ensure all logs are clear and audit-ready. - Run **security & performance tests** for Permissions Control and Workflow transitions. - Prepare system demonstration and check repository information completeness

6. Overview of the Implemented Solution

In this section it is shown how the architectural proposal was materialized in the source code, database and management interface of Piranha CMS. We maintained as a guideline the same quality attributes identified in Part 1, in section 2.1.1 – **performance, scalability, auditability, security, usability and reliability**– and systematically verified whether each technical decision contributed to these objectives.

6.1 Solution Overview

The solution was divided into five logical projects, aligned with the layered architecture defined in Part 1. We maintained unidirectional dependencies (Domain → Application → Infrastructure → Presentation) to facilitate testing and future updates of Piranha CMS.

Layer	Project	Project References	Key Responsibility
Domain	Piranha.Editorial .Abstractions		Domain models (entities, enums, interfaces) without external dependencies.
Application	Piranha.Editorial	Piranha.Editorial.Abstractions Piranha.Data.EF.SQLite Piranha.Data.EF.SQLServer	Business rules / services (EditorialWorkflowService, validators) and migrations across multiple providers.
Infrastructure / Data	Piranha.Data.EF .SQLite	Piranha Piranha.Data.EF Piranha.Editorial Piranha.Editorial.Abstractions	Extending the original DbContext with workflow DbSets + migrations.
Presentation	RazorWeb	Piranha, Piranha.AspNetCore Piranha.Data.EF Piranha.AspNetCore.Identity Piranha.Manager Piranha.Editorial Piranha.Editorial.Abstractions Piranha.ImageSharpPiranha.Local.FileStorage	Host ASP.NET Core, controllers, Vue dashboard, seeders, OTEL.
Observability	Grafana Jaeger		Prometheus/Grafana dashboards for

	OTel Collector		workflow metrics and traces.

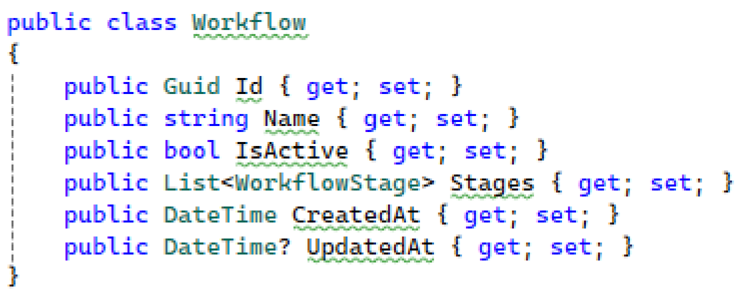
6.2 Domain Modeling

To support editorial workflows with multi-stage approval and traceability, the core CMS domain was extended to include a rich set of editorial concepts. These new domain entities enable structured content lifecycle management while remaining fully decoupled from presentation and persistence concerns.

The goal was to ensure that each editorial item could be created, reviewed, approved, published (or rejected) according to a predefined sequence of steps, always with associated traceability and permission control.

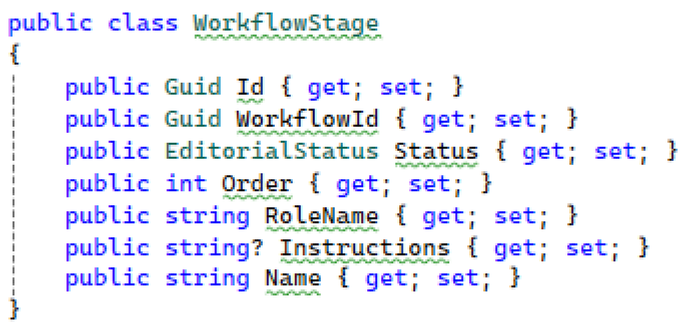
To achieve this vision, the domain model was developed in a modular way and compatible with the existing CMS structure. All editorial entities were created in the ***Piranha.Editorial.Abstractions*** project, remaining independent of the presentation and data layers. They were designed to support multiple workflows, ordered steps, specific transitions, current states per content, and also a complete record of all relevant actions throughout the editorial cycle.

The **Workflow** entity represents a configurable editorial flow and groups the sequence of steps through which the content should pass.



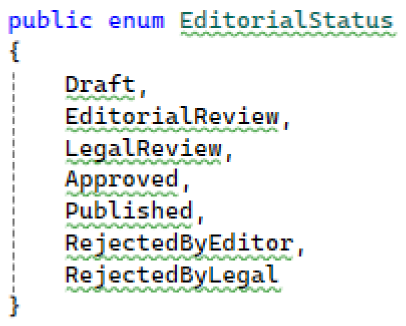
```
public class Workflow
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public bool IsActive { get; set; }
    public List<WorkflowStage> Stages { get; set; }
    public DateTime CreatedAt { get; set; }
    public DateTime? UpdatedAt { get; set; }
}
```

This ordered list of stages is represented by the **WorkflowStage** entity, which defines not only the stage order, but also the role responsible for that stage and the editorial status associated with it. This allows each organization to adapt its workflows to its internal approval policies.



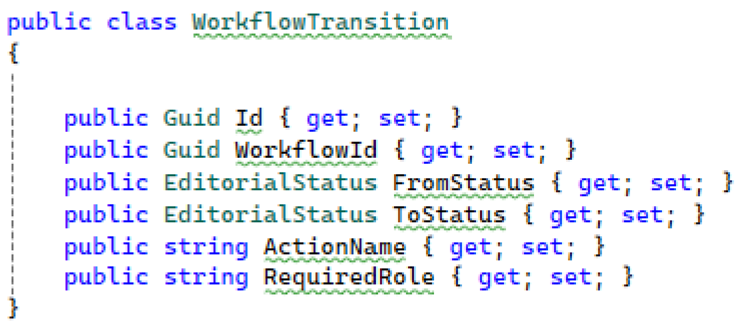
```
public class WorkflowStage
{
    public Guid Id { get; set; }
    public Guid WorkflowId { get; set; }
    public EditorialStatus Status { get; set; }
    public int Order { get; set; }
    public string RoleName { get; set; }
    public string? Instructions { get; set; }
    public string Name { get; set; }
}
```

On the other hand, the **EditorialStatus** enum standardizes the status to give semantic consistency to the different review stages, transitions, and historical records, with the possible values being *Draft, EditorialReview, LegalReview, Approved, Published, and Rejected*. These values mark generic phases of content review in an institutional or regulated setting.



```
public enum EditorialStatus
{
    Draft,
    EditorialReview,
    LegalReview,
    Approved,
    Published,
    RejectedByEditor,
    RejectedByLegal
}
```

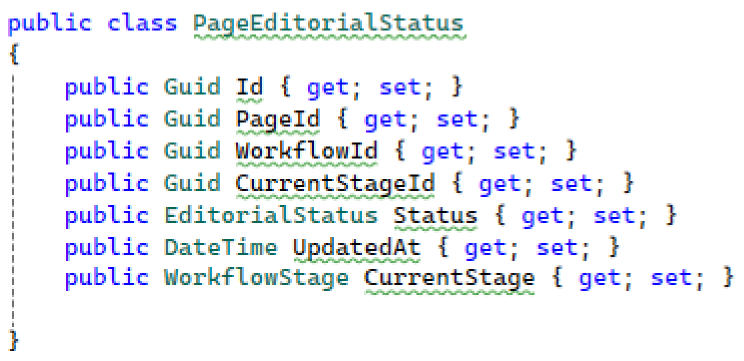
Besides the conventional sequence of steps, the **WorkflowTransition** entity was introduced. This entity defines explicit transitions between two states, allowing these transitions to be tied with a ***certain role*** to reinforce ***permission control***, thereby permitting only duly authorized users to proceed with a step or move it back.



```
public class WorkflowTransition
{

    public Guid Id { get; set; }
    public Guid WorkflowId { get; set; }
    public EditorialStatus FromStatus { get; set; }
    public EditorialStatus ToStatus { get; set; }
    public string ActionName { get; set; }
    public string RequiredRole { get; set; }
}
```

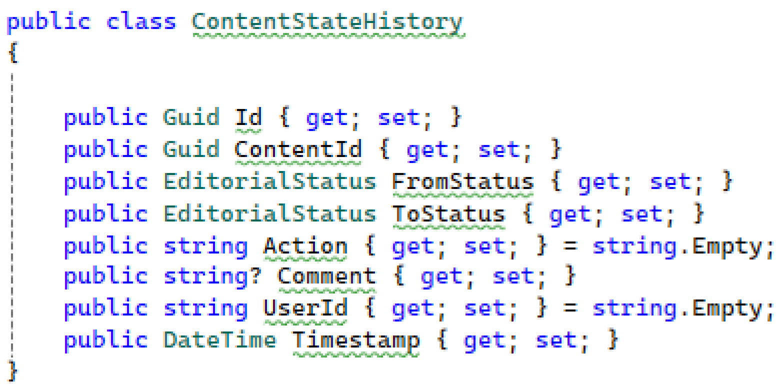
In the CMS, every content piece now maintains an editorial state tracked through the **PageEditorialStatus** entity. This entity has a 1:1 relationship with the page indicated by PageId. Thus, it holds the active workflow identifier, the current state (*EditorialStatus*), and the date when it was last updated. Separation from Piranha's Page entity means no intrusive changes in the future that could prohibit compatibility with the CM.



```
public class PageEditorialStatus
{
    public Guid Id { get; set; }
    public Guid PageId { get; set; }
    public Guid WorkflowId { get; set; }
    public Guid CurrentStageId { get; set; }
    public EditorialStatus Status { get; set; }
    public DateTime UpdatedAt { get; set; }
    public WorkflowStage CurrentStage { get; set; }

}
```

To ensure auditability and traceability, the **ContentStateHistory** entity was introduced. This table stores all relevant editorial transitions, including the transition source and destination, the responsible user, and the date of the operation. This history is critical for review processes, compliance, and generating metrics (such as average time to publish).



```
public class ContentStateHistory
{

    public Guid Id { get; set; }
    public Guid ContentId { get; set; }
    public EditorialStatus FromStatus { get; set; }
    public EditorialStatus ToStatus { get; set; }
    public string Action { get; set; } = string.Empty;
    public string? Comment { get; set; }
    public string UserId { get; set; } = string.Empty;
    public DateTime Timestamp { get; set; }
}
```

Altogether, these entities form a rich, expressive, and extensible domain upon which the editorial logic of the system will be established. In the next section, the application layer interacting with this model—which includes services, repositories, and state transition mechanisms—will be explained.

6.3 Application Logic

After having a well-defined and isolated editorial model, it became necessary to create the services that would enforce the business rules governing the editorial cycle of content. These services, along with the supporting repositories, were gathered under the **Piranha.Editorial** project, clearly separating domain from applications. This, in turn, enables models to be reused in different contexts depending on the need (API, UI, tests), thereby ensuring all of the editorial logic is focused and cohesive.

Acting as the heart of this application is the service called **EditorialWorkflowService** that performs validation and application of state transitions and assignment of workflows to newly created content and consultation of available transitions by user roles. This service depends on a repository (**IWorkflowRepository**) and interacts directly with the shared **DbContext** through dependency injection.

6.3.1 Managing the initial state

Whenever editorial content is created—for example, a new page—the system needs to ensure that its editorial state is correctly initialized. This responsibility falls to the **EnsurePageStatusAsync** method, which queries the **PageEditorialStatuses** table and, if there is no entry for that page yet, creates a new record with the initial step defined in the active workflow.

The method also immediately records this creation in the history, and is fully instrumented with OpenTelemetry, assigning useful tags and statuses for tracking.

6.3.2 Validating and Applying Transitions

Transitions are editorial movements done through the **ApplyTransitionAsync** method defined in the **EditorialWorkflowService** service. This single method contains all the logic that ensures the transition is permitted and safe and can be audited while moving content from one stage to another in the editorial workflow.

In the beginning, the page status is obtained from **PageEditorialStatus**, which contains the **WorkflowId**, status, and stage identifier. It uses this information to establish if there is a transition set up between the current state and the target state by fetching entries in the **WorkflowTransition** table. If any are there, it also specifies which role should be allowed to make that transition; the role check will have happened earlier in the UI layer with **GetAvailableTransitionsAsync**.

Should the transition be valid, the target stage related to the new state is acquired, so that the system can work out the stage's order in the flow, expected role, and instructions for the new stage.

Before applying the change, the method records several observability metrics via OpenTelemetry:

- A counter that records the transition (**from** → **to**), useful for frequency reporting.
- A histogram that measures the time elapsed since the last transition (based on **UpdatedAt**), grouped by source and target.
- If the transition is a return to draft, a specific counter is incremented per paper type (e.g., editor rejection).

Additionally, the system implements a temporary caching logic: when a page enters “Draft”, the timestamp of this entry is stored in memory. If the same page is later published, the total time elapsed between the start of the draft and publication is calculated and recorded in a **timeToPublish** metric, also via OTEL.

Then, the state is effectively changed: the **PageEditorialStatus** table is updated with the new state, the **CurrentStageId** corresponding to the target stage, and the date/time of the transition.

If the new state becomes Published, the Piranha page service is called and the page is marked as published in the CMS data model (Published = DateTime.UtcNow). Similarly, when the content falls under **Draft**, the publication flag is set to null (Published = null), causing the actual unpublishing of the page; hence, content editorial state is always aligned with its public visibility.

Eventually, the change is entrenched in the **ContentStateHistory** table. This change comprises the source state, the target state, the page identifier, the actor who executed the transition, the name of the action (e.g., "Final Approval"), and the date of the event. This history serves as a basis for future auditing and can be viewed in the admin interface.

The entire method is instrumented with OpenTelemetry and structured logging. Exceptions caught during the operation are recorded in the **trace** with the **Error** status, and the full error is saved as a tag in the trace context, allowing later analysis via Jaeger/ Grafana.

This approach ensures that:

- Transitions are **validated against real data and configurable rules**.
- **Page publishing and unpublishing** always follow the editorial status.
- The system is **auditable, observable, and responsive**.
- Logic is **encapsulated and centralized**, keeping the domain clean and testable.

6.4 Data Initialization and Seeders

In order for the editorial system to function correctly from the very beginning, it was necessary to create an initial data seed mechanism that would guarantee the existence of editorial roles, a functional workflow and respective steps. This phase is essential for the system to be installable autonomously, that is, for any new instance of the CMS to have immediately available the structure necessary to manage content based on editorial workflows.

The seed process was divided into two main points:

- Creating the roles and respective claims.
- Creating the base editorial workflow with defined steps, order and states.

6.4.1 Creating the editorial roles

The **SeedEditorialRoles** function is responsible for ensuring that the roles essential to the editorial process exist and are duly registered in the identity system. The roles defined were:

- **Author** – responsible for the initial creation of the content.
- **Editor** – in charge of editorial review and corrections.
- **Attorney** – performs the legal review of the content.
- **Director** – responsible for final approval and publication.

For each of these roles, an entry was created, if not already in place, in the Roles system, associating it with the claims (permissions) appropriate to the functionalities that you should be able to access in the CMS. This process is idempotent — that is, it can be executed multiple times without creating duplications — and was implemented using the **RoleManager<T>** service of the ASP.NET Core framework.

6.4.2 Creating the base workflow

The creation of the initial editorial flow was done using **EditorialSeeder**, a service that directly queries the **DbContext** and registers the first workflow only if none already exists. This workflow is called "**Workflow Académico**", the default one, and contains five steps, ordered as follows:

- **Draft** – assigned to the Author.
- **EditorialReview** – assigned to the Editor.
- **LegalReview** – assigned to the Attorney.
- **Final Approval** (Approved) – given to the Director.
- **Published** – also under the responsibility of the Director.

Each step defines the corresponding editorial status, the expected role, and optionally contextual instructions that can be displayed in the UI.

The seeder also ensures that all expected transitions between these steps are created in the **WorkflowTransitions** table. These transitions are essential for the system to correctly validate the flows and know which actions are allowed at each point.

6.4.3 Automatic seed execution

The seed process was incorporated directly into the application's boot phase, through the **Program.cs** file. As soon as the **WebApplication** is created, a service **scope** is opened, and within it the following are executed:

- **SeedEditorialRoles** – for roles.
- **EditorialSeeder** – for the workflow and its steps.

This mechanism ensures that, when the CMS is run for the first time (for example via **dotnet run** or in Docker), the system is already fully prepared to create pages, associate them with an editorial workflow, and perform transitions.

6.4.4 Advantages of the approach

Several advantages come with this initialization strategy:

- **No manual configuration**- flexibility to perform any manual intervention after installation.
- **Avoid inconsistencies**- the same workflow and the roles should be used in dev/test/prod.
- **Compatible with CI/CD**- can be run in deployment pipelines to prepare new databases.
- **Secure and repeatable**- pre-existence checks for every insertion are carried out.

With this initial database, all features of the editorial system are active right away, which allows the full Draft → Publish flow to take place with the first content produced.

6.5 User Interface Integration

The integration of the editorial logic into the Piranha CMS administration interface has been implemented to ensure usability, visibility, and coherence with the domain-defined roles. The purpose was to let users of different roles (Author/Editor/Lawyer/Director) check the status of content and carry out permitted actions in an obvious, contextualized, and intuitive manner.

6.5.1 Dynamic editorial actions

On the content editing page, dynamic action buttons were added that represent the possible transitions from the current state. These buttons are generated at runtime, based on response of the **GetAvailableTransitionsAsync** method, which filters the valid transitions according to:

- The current state of the content.
- The workflow to which it belongs.
- The roles assigned to the authenticated user.

Depending on one's role, some buttons have to be shown, and some have to be hidden. For example, submission for legal review appears for an Editor, while the Lawyer will see only approval for the publication or rejection.

To get support for that logic straight into the Piranha Manager interface, the Vue.js **piranha.pageedit** component had to be extended to handle editorial data and actions. Some of the improvements done are:

- Storing the current editorial state;

- Loading transitions valid for the user dynamically;
- Submitting pages to editorial review;
- Applying transitions through AJAX calls, updating page status and providing visual feedback;
- Rendering buttons with icons and styles depending on the action type (e.g., publish, reject, return to draft).

All the above interactions are based on the new editorial feature, ensuring that transitions obey the defined workflows and user permission. So with this, the editorial flow gets fully integrated into the editing experience in an intuitive, secure, and extension-free way.

6.5.2 Identifying the editorial status

An editorial status badge has been added to the top of the editing interface, with different color and text depending on the current status.

The badge serves to immediately inform the user about the point at which the content is, for example:

- Draft
- Editorial Review
- Legal Review
- Approved
- Published

6.5.3 Workflow Dashboard

A new interface has been developed to centrally monitor the editorial status of all pages. This interface, accessible through the “Content” section of the CMS, displays pages organized by site (useful in multi-tenant environments) and grouped in a format similar to an interactive sitemap.

Each group corresponds to a distinct site (e.g.: “Institutional Portal”, “Engineering College”), and displays:

- The title of the page.
- The date of the last update.
- A badge with the name of the current editorial stage, colored to visually highlight the progress of the content.

Below each page, if there are records in the history, a compact table is displayed with all previous transitions, including:

- Action performed (e.g.: “Send for Editorial Review”).

- Source and target status.
- Identifier of the user who executed the transition.
- Formatted timestamp (date and time).
- If the content has not yet undergone any transition, the indication “No editorial history” is displayed.

This dashboard allows you to:

- Have an overview of all editorial content on a single screen.
- Check who did what and when.
- Quickly diagnose stagnant or rejected pages.
- Group by context (e.g. site or section), which helps distributed editorial teams.

6.6 Observability and Metrics

One of the pillars of the project was to ensure that the editorial system not only worked correctly, but was also *observable*, *auditable* and *measurable*. To this end, several tools from the **OpenTelemetry (OTEL)** stack were integrated, allowing real-time metrics to be collected, executions to be tracked and bottlenecks or usage patterns to be identified in the editorial cycle.

The instrumentation was applied both at the **aggregated metrics** level (via Prometheus/Grafana) and distributed tracing level (via Jaeger), enabling both *quantitative* and *behavioral* analysis of editorial workflows.

6.6.1 OpenTelemetry SDK

The instrumentation was configured directly at the initialization of the application of the RazorWeb project, where a **Meter** was registered with the name of the service ("RazorWeb"), which was later injected into the appropriate projects, and the OTEL integrations were activated for:

- ASP.NET Core (HTTP requests).
- HttpClient (external calls).
- Manual events defined by code (*ActivitySource* + *Meter*).

The metrics were exported via OTLP to the Collector, from where they were redirected to Prometheus (metrics) and Jaeger (tracking).

6.6.2 Editorial flow metrics

Several customized metrics were defined and registered, specific to the editorial lifecycle:

Counters

- **workflow_transition_total**: number of transitions applied, grouped by *from* → *to* pair.
- **workflow_rejected_total**: number of rejections, with *role, from, to* labels.

These statistics reveal a very important quality, they let us know how extensively each walk is trodden before a walk-through is rejected and which key roles are active in the many roles in an editorial process.

Histograms

- **workflow_transition_duration_seconds**: the time elapsed in a transition, calculated from the difference between the UpdatedAt date of the previous step and the moment of the transition.
- **workflow_time_to_publish_seconds**: The total time taken between going into a draft and publishing it at last. The very first timestamp is cached at the first entry in "Draft" and retrieved on publication.

Those histograms help analyze **editorial efficiency**, trying to identify content that takes very long to get approval or to be published.

6.6.3 Distributed tracing with Jaeger

In addition to the metrics, detailed instrumentation of the main methods of the **EditorialWorkflowService** was performed, such as:

- **EnsurePageStatusAsync**
- **ApplyTransitionAsync**
- **GetAvailableTransitionsAsync**

Each of these methods creates an OTEL Activity, with relevant tags such as:

- page.id, workflow.id, stage.id
- target.status, duration.seconds
- exception (in case of an error)

These spans are automatically grouped into complete traces, which can be visualized in Jaeger, allowing analysis of the system's behavior in each editorial request — from the creation of a page to its publication, with all intermediate points included.

6.6.4 Grafana Dashboards

The data collected by Prometheus could then be visualized in launched **dashboards in the Grafana interface**, including:

- Transition stage-by-stage graphs (e.g., Draft → Editorial Review).
- Histogram of average publication times for content.
- Tables of records of refusals per role.
- Global metrics (transitions per day, per site, per author).

The dashboards are useful for **operational monitoring** and **strategic analysis** to enable editorial teams to evaluate how well workflows are working and, if necessary, adjust editorial practices.

6.6.5 Benefits for Scenario 2

Observability at this level of integration is not just technical; it is thoroughly aligned with the Scenario 2 objectives – Editorial Workflows & Content Approvals:

- Transparency, whereby all steps of a content are recorded.
- Accountability: every action is assigned to a user and a role.
- Editorial performance: measure and optimize time between submission and publication.
- Rapid diagnosis of errors and anomalous behaviors, which can be detected visually.

This is a substantial improvement over the previous CMS operation, working toward turning Piranha into a fully auditable and intelligent editorial platform.

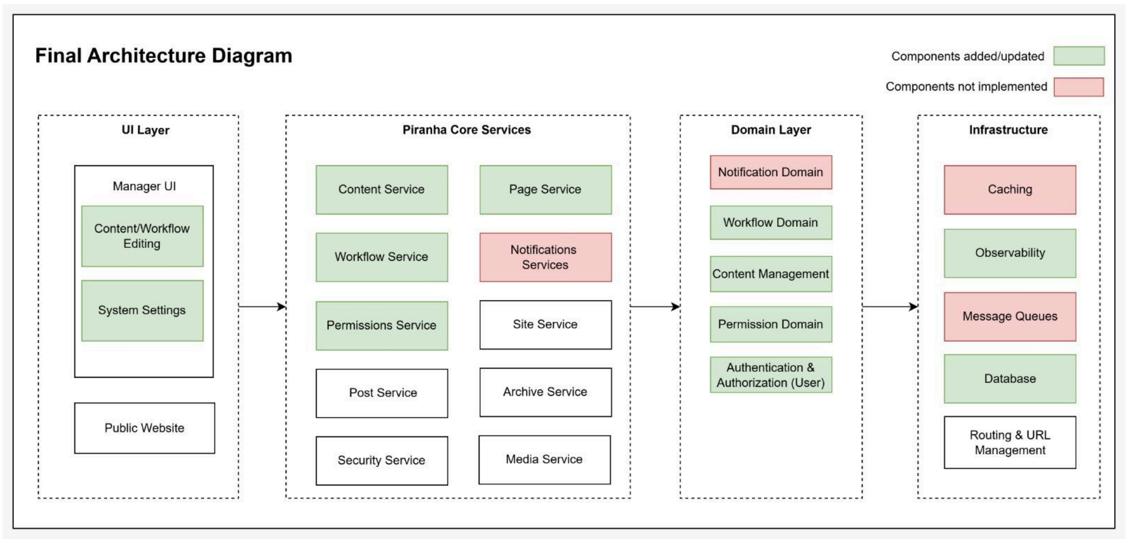
6.7 Final Considerations

The implementation of the editorial logic on Piranha CMS proved to be effective, modular and extensible. By introducing editorial workflows with ordered steps and secure transitions with role validation, it was possible to establish a coherent, auditable and adaptable editorial cycle to multiple organizational realities.

The layered architecture, the use of interfaces and separation of responsibilities allowed maintaining compatibility with the CMS core, without compromising the capacity for future evolution. Integration with the management interface and tools such as Prometheus, Jaeger and Grafana also ensured full visibility into the behavior and performance of the system.

The result is a functional, observable editorial solution that is strongly aligned with the objectives of the proposed scenario, suitable for adoption in real contexts with multi-stage validation and publication requirements.

To conclude, the following architecture diagram summarizes the evolution of the system throughout the implementation. Components added or extended in this scenario are shown in green, while not implemented components are marked in red.



7. Conclusion

This project demonstrated how it is possible to evolve an existing platform — in this case, Piranha CMS — to meet more complex and demanding scenarios, without compromising the fundamental principles of architecture, performance and maintenance.

By introducing editorial workflows, the system began to support robust approval flows, with granular control by role, historical record of transitions and native integration with the content lifecycle. The proposed solution was developed in a modular way, with functional tests and continuous integration, respecting good software architectures and engineering practices.

In addition, the concern with observability and metrics from the initial phase allowed us to ensure that the system not only works, but can be monitored, audited and optimized over time.

As a whole, the project constitutes a concrete demonstration of how to apply concepts of software architecture, domain-oriented design and modern instrumentation to solve a real problem with efficiency, extensibility and technical rigor.