This proposal defines the vision and scope of our next-generation CMS for the Eudonet group. The goal is to unify existing solutions, simplify the editor experience, and provide a flexible, scalable foundation that supports both simple sites and complex multi-site deployments. It outlines the core principles we want to enforce from day one—performance, security, governance, and extensibility—while also identifying phased enhancements to keep the solution sustainable. The document is intended as a baseline for product leadership decisions and as a starting point for the technical specifications our teams will draft next.

Next-Generation CMS: Solution Draft Proposal
From fragmented tools to a unified, next-generation CMS.

Writer	**Loïc Février**	**R&D Manager**
Validator	**Franck Amiot** **Marc Thevenin** **Renaud Sibel**	**CTO** **CPO** **CEO**
Diffusion	Internal	

Date	Version	Description of changes	Author
04/09/2025	V0.1	1st draft	**Loïc Février**
20/09/2025	V0.2	1st version	**Loïc Février**

1 EXECUTIVE SUMMARY

This Solution Draft Proposal defines the vision and scope for our next-generation, group-wide CMS. The objective is to unify capabilities across products, give editors a coherent and efficient experience, and lay a scalable foundation that supports both simple marketing sites and complex multi-site deployments. Starting fresh lets us design for performance, governance, and extensibility from day one, rather than carrying legacy compromises.

Our approach centers on a **headless architecture**, a **block + plugin model** for flexible composition, and **design tokens with themes/variations** to ensure brand consistency at scale. We adopt clear **routing/SSR** rules, **canonical URLs** with localized slugs, and built-in **redirect management** to protect SEO. Search launches with a lean v1.0 (pages + articles) and a governance path (synonyms, analyzers, "no-results" insights) to maintain relevance as content grows.

Quality and compliance are first-class: **performance budgets** (server TTFB, JS/HTML caps, image/font guidance) and **EU-based CDN**; **accessibility (RGAA) commitment** with target levels to be decided at group level (front vs back); **SEO guardrails** (hreflang, sitemaps, heading semantics, absolute breadcrumbs); and **analytics with CMP** (Matomo) plus a lightweight **events specification** so CMS and plugins emit consistent, privacy-aware data. Time handling is standardized (**stored in GMT**), with timezone detection in the front-office and user-selectable timezones in the back-office.

Security follows a pragmatic v1.0 posture and tightens over time: **trusted plugin manifests** declaring routes, SSR/cache/SEO behavior; sensible defaults (CSP/WAF, least-privilege tokens), and progressive isolation as the ecosystem expands. **Media** is stored in **S3-compatible** storage with **on-the-fly image resizing** via URL parameters and controlled cache eviction. Operability and sustainability are built in: beyond AI usage, we instrument **FinOps hooks** for storage, bandwidth/CDN, image transforms, and search queries, enabling transparent reporting, guardrails, and future plan tiers.

Migration is treated as a product experience: **pre-flight diff reports** (page counts, broken links, redirect plan), **self-service tools**, and automatic **301s** to protect traffic. Internally, we coordinate closely with **One-Platform** on design system alignment and API coherence; documentation is deliverable, not an afterthought.

Our MVP (v1.0) at a glance

- **Pages** with SSR, draft/versioning, i18n slugs, canonical URLs, and redirect management.
- **Articles** with flat categories + tags and **auto-generated landing pages** (e.g., Newsroom).
- **Themes & variations** driven by **design tokens**, with **custom CSS disabled by default** and unlockable as an advanced option.
- **Search v1.0** indexing pages and articles.
- **Plugin manifest v1.0** exposing cacheability/SSR flags and one flagship **route-owning app**.
- **Performance budgets + EU CDN** (clear caps and enforcement).
- **AI assist** limited to editorial help, translations, alt-text, and basic SEO.
- **Analytics (Matomo) + CMP**, with a minimal **events spec** for consistent telemetry.

Immediately after v1.0 (candidate v1.1 focus)

- Search governance basics (synonyms, locale analyzers, "no-results" insights).
- Editorial workflow enhancements (scheduling, light audit).
- Media metadata/tagging; selective DAM features.
- **Integration & automation via webhooks + n8n (v1.5)** to avoid connector sprawl and unlock no-/low-code flows, including AI-assisted content ops.

How to read the roadmaps
Version labels (**v1.0, v1.1, v2, v3**) are **per-capability maturity stages**, not a single global release train. Beyond v1.0, **Product Owners** will sequence delivery by business value and readiness; "v2" for Search may land before "v2" for Workflows.

Top risks & mitigations (tracked)

- **Dependency on Unlayer** → adapter boundary + periodic checkpoint.
- **KPIs not yet finalized** → instrument telemetry in v1.0 and set 3–4 north-star KPIs in parallel.
- **Migration complexity** → pre-flight diffs, redirect automation, and staged cutovers.

Decision needed from leadership: endorse the **MVP scope and guardrails**, the **per-capability roadmap interpretation**, and the **governance priorities** (performance, accessibility targets, analytics/PII policy) so Product and Engineering can proceed to detailed specifications with confidence.

2 TABLE OF CONTENTS

1 Executive Summary

2 Table of Contents

3 CMS Workshop Summary – June 2025

3.1 Introduction

3.2 Current Situation

3.3 Analysis of Netanswer (Workshop Conclusions)

3.4 Implementation Approach

3.5 Overall Vision

4 A New AI Direction

4.1 Context

4.2 Needs

4.3 A Headless CMS

4.4 AI-First

4.5 Step 1: A Single Website With Static Content

4.6 Step 2: Articles, Content Types & Taxonomies

4.7 The Power of Dynamic Blocks Within an Open Ecosystem

4.8 Step 3: Multiple Websites/Tenants

4.9 Step 4: Integrate a Private Space

5 A full CMS

5.1 Internationalization (i18n)

5.2 RBAC (Role-Based Access Control)

5.3 Editorial Workflows & Governance

5.4 Search

5.5 Media & Digital Asset Management (DAM)

5.6 Redirect Management

5.7 Themes and Variations

5.8 Analytics and A/B Testing

5.9 A Focus on Performance and Security

5.10 Integration & Automation

6 AI: Vision, Governance, Guardrails

6.1 AI at the service of our clients

6.2 Our AI Posture

6.3 Future Outlook: AI-driven Site Generation

7 Risks

7.1 Dependency on Unlayer (Editor)

7.2 Lack of clear KPIs to guide roadmap decisions

7.3 Migration of existing clients

7.4 Other future risks
8 Implementation Notes & Open Topics
8.1 Pricing Model
8.2 APIs
8.3 Technical Foundations & Work Organisation

3 CMS WORKSHOP SUMMARY – JUNE 2025

3.1 Introduction

This document summarizes the conclusions of the workshop held in June regarding the future of the CMS within Netanswer. The objective at that time was to analyze the current situation, identify limitations, and explore possible evolutions. This serves as a reference point before presenting a new proposal.

3.2 Current Situation

3.2.1 Netanswer CMS

- **Static content**
  - Edited with Unlayer (and older pages with TinyMCE).
  - Each page has its own URL and can be included in menus.
  - A Page Builder exists for predefined static content blocks.
- **Dynamic content**
  - News, announcements (marriage, birth, death, appointments, etc.).
  - Each item has its own page; there is also a “news” page that can be added to menus and configured.
- **Menus**
  - Can include static content, external links, and dynamic pages (news, job board, events, shop, groups, etc.).
  - Access rights, SEO, and multilingual menu relationships are managed here.
  - **Note**: static and dynamic pages also have their own access rights outside menus, leading to duplicated controls.
- **Homepage and group websites**
  - Main homepage has its own editor with configurable blocks.
  - Groups’ mini-sites use Page Builder with other block types.
  - Multi-tenant support allows multiple front-ends with separate menus and content.
  - Groups can either:
    - Have a dedicated page with submenu inside the main site, or
    - Run an independent mini-site with their own menu.
  - Event mini-sites exist with predefined templates (menus cannot be changed).
- **Private space and login**
  - CMS integrates login and member private space (view/edit personal data).
- **Limitations**
  - Fragmented tools (Unlayer, TinyMCE, Page Builder, homepage editor).
  - Inconsistent design across modules (buttons, pagination, etc.).

3.2.2 WMS CMS

- Main differentiator: ability to combine **static blocks** and **dynamic blocks** on the same page.
- This feature is not possible in Netanswer today.

3.3 Analysis of Netanswer (Workshop Conclusions)

3.3.1 Content Editing (Unlayer as Common Base)

- Remove Page Builder and homepage editor; achieve functionality with dynamic blocks in Unlayer.
- Migrate all TinyMCE content to Unlayer (AI-assisted, with a client migration plan).
- Create new configurable dynamic blocks in Unlayer to guide clients toward good results.
- Dynamic pages become Unlayer blocks, which can be combined with static blocks.
- Access rights managed at content level, menus used primarily for organisation.
- For non-activated modules, display a placeholder message rather than breaking the page.

3.3.2 Dynamic Pages

- Dynamic pages generated strictly from code, no direct DB intervention.
- Multiple block variants for certain content (e.g., events list).
- **Decisions to be made:**
    - Breadcrumbs: absolute vs. menu-based.
    - Handling related URLs (e.g., /directory vs. /directory/search).
    - News page action buttons: should they be removed?

3.3.3 Design System

- Need for a unified design system to standardize UI (buttons, pagination, forms, headings).
- Replace infinite scroll with consistent server-side pagination.
- Standardize rendering across modules (news, events, groups).
- Adoption of VueJS as front-end framework.
- **Decisions to be made:**
    - Degree of customization allowed in VueJS (standardized vs. custom).
    - Integration of multiple icon libraries (e.g., FontAwesome and others).

3.3.4 Miscellaneous Requirements

- Draft mode available for all content, including homepages.
- Rework private space layout (current width too limited).
- Redesign login panel for better UX (see Mantis #83115).
- Support for .webp image uploads.

3.3.5 Technical Issues

- Update jQuery versions in front-office and reduce dependency.
- Clean up unused JavaScript libraries.
- Improve RGPD compliance and accessibility.
- Fix inconsistent heading structure (H1/H2/H3).
- Strengthen SEO (canonical tags, metadata).

3.3.6 Customization

- Allow static content above/below dynamic blocks.
- Keep dynamic template customization limited within CMS.
- For larger clients:
    - Provide CSS/template overrides.

    - Extend templates via code.
- Enable custom blocks, third-party integrations (chatbots, fonts, tools).
- Explore Elementor-like extensibility for future enhancements.

3.4 Implementation Approach

- Develop **v3 in parallel** with current version, initially for new sites.
- Proceed **module by module**, analyzing constants along the way.
- Manage co-existence of Bootstrap 4 and 5 during transition.
- Identify clients with local CSS that may break during migration.
- Handle legacy homepages before standardized ones (e.g., AAENP, AEGE, AFSE, Bginette, PFEI, Prytanée).
- Define billing model for updates to client-specific dynamic pages.

Required resources:

- Short-term front-end expert for setup.
- Potential support from an agency.
- Long-term front-end profile to assist Pierre (currently only full-stack).
- Decide on final stack (VueJS, Tailwind, Flowbite).

3.5 Overall Vision

The objective is to progressively transition Netanswer towards a **block-based architecture** centered on Unlayer, integrated with a unified design system.

- **Technical direction**: move from Twig/MVC templates to an API-first approach with VueJS or another JS framework, while maintaining hybrid compatibility where necessary.
- **Main challenges**:
    - Ensure progressive migration without breaking existing sites.
    - Manage customization requests and legacy pages.
    - Deliver a consistent and modernized user experience.

4 A NEW AI DIRECTION

4.1 Context

With the global strategy for 25/26 the possibility to do a full rewrite of the Netanswer's CMS has been considered. That strategy would allow us to have a full commitment to One-Platform.

The CMS component has been the subject of a proof of concept (POC, August 2025) exploring the feasibility of building a next-generation CMS from scratch with AI assistance. The objective is a **best-in-class CMS** enabling standardized publishing workflows through reusable components, while remaining compatible with One-Platform from the outset and deployable as a stand-alone solution in the interim.

The POC was conclusive: we can be confident in our ability to deliver a new CMS that will serve both the One-Platform and strategic initiatives such as the **AssoPro remediation plan** (in combination with EudonetCRM and ExtranetX) and the **WMS migration**. Some additional work will be required to support more complex private spaces (e.g., Netanswer's member areas) and highly dynamic pages (job boards, directories), but the roadmap is clear toward a **plugin-based, extensible CMS** architecture.

Our experience building that POC showed us the importance of a clear direction (with detailed instructions) and then build things progressively with the help of the AI. Thus we need to build a detailed specification of what we want our future CMS to be able to do, from the high level constraints/possibilities up to our wishes in term of personalisation and including detailed information about how things are supposed to work. The technical choices are out of scope of that document but the architecture, seen from the possibilities that they would offer are absolutely on scope.

4.2 Needs

We don't need "a" CMS, we need "multiple" CMS :

- We need a light CMS ("Website-Builder") that can be used within the One-Platform to publish static content. Example: before presenting a form for X or Y, I want a full page of presentation of the context.
- We need a CMS to power the new version of GiveXpert so possibly multiple mini-websites with integration of dynamic widgets (the amount collected for a given collection campaign) and then integration of the GiveXpert donation form within that CMS.
- We need a CMS that also works as a stand-alone solution (i.e. outside the One-Platform) in combination with Eudonet CRM and ExtranetX and possibility integrate ExtranetX in a full user-experience.
- We need a CMS that can be integrated into the current Netanswer solution with all the dynamic pages, including events, jobbboard, private space…
- We need a CMS than can handle more complexed cases such as multiples websites, mini-websites for event, mini-websites for group, different websites for different tenant (multi-association)

Of course, we need these CMS to be a **single CMS** able to handle these different contexts with a deep integration into our full eco-system.

4.3 A Headless CMS

A traditional CMS combines the back-end (where content is created and stored) with the front-end (the website that displays it). A headless CMS separates the two: the back-end only manages and exposes content through APIs, while one or several front-ends decide how to present it. This means we can enter content once and use it everywhere — for example, the same news article can appear on our main website, in a mobile app, and in a client portal, without duplicating effort. It also gives us the freedom to build different front-ends using the most suitable technology for each case — a full-featured website in one framework, and a lightweight, simpler portal

in another. The main advantage is flexibility and consistency across platforms, while the trade-off is the need for more development work on the front-end side.

Let's revisit what we envisioned for the CMS in our workshop, this time through the lens of the new architectural direction. We'll proceed by growing the scope progressively for easier understanding of the full picture, but the actual order of development might be different.

Note: because we're building from scratch, we don't need to address compatibility or migration of existing sites for now

4.4 AI-First

Adopting an AI-First approach in the design of our new CMS (2025–2026) is essential to ensure both long-term relevance and immediate value. From a business perspective, AI capabilities can accelerate content production, simplify personalization, and continuously optimize SEO, accessibility, and user engagement — features increasingly expected by clients. From a technical perspective, embedding AI into the core architecture enables automation of repetitive tasks, consistent quality control, and a modular design that evolves with new AI models without requiring structural rewrites. Positioning AI as a foundation rather than an add-on transforms the CMS into an active co-pilot for both users and developers, ensuring efficiency, scalability, and differentiation in the market.

Let's imagine what our flagship AI features could be

Short-term (2025) – "AI-assisted content editing"

The CMS integrates AI natively into the editor to suggest improvements to grammar, style, tone, and readability, while also generating SEO metadata and translations on the fly. This provides immediate, visible value to all users and sets the baseline expectation for an AI-First platform.

Medium-term (2026) – "AI-powered design generation"

Beyond text, the CMS includes AI-driven layout and image tools. Users can request new landing pages or mini-sites built from existing APIs and content, or transform visuals directly within the CMS ("make brighter," "remove background," "add a banner"). This reduces dependency on external tools and empowers clients with more design freedom inside the platform.

Long-term (beyond 2026) – "Adaptive personalization engine"

The CMS evolves into a dynamic system where content, design, and even layout adapt automatically to the visitor's profile, behaviour, or context. Combined with predictive recommendations and compliance guardrails, the platform becomes not just a publishing tool but a personalized digital experience engine.

We'll discuss more about the possibility of AI and the associated risks later on in that document.

4.5 Step 1: A Single Website With Static Content

4.5.1 The "Page": Our Base Component

In our CMS, everything starts with the concept of a **Page**. A Page is the core unit of content: it can be published, indexed, and accessed at its own **canonical URL**, whether it appears in a menu. Menus are strictly navigation tools — they never define access rights. Permissions are managed directly on the Page through **RBAC (Role-Based Access Control)**.

Breadcrumbs will be generated as **absolute paths based on routes**, not menus. This ensures predictable behaviour for SEO and caching, while keeping menus strictly navigational. By decoupling breadcrumbs from menu structures, we avoid inconsistencies and guarantee that content always exposes a canonical path.

Pages are versatile: some sites will only use static Pages, while others will combine **static content** (text, images, formatted layouts) with **dynamic blocks** (e.g., news lists, events, job boards). Each Page contains at least a title, metadata for SEO, and rich content edited in **Unlayer**, the editor already used in our applications.

To reduce copy/paste and ensure consistency, we will provide custom Unlayer blocks for reusable fragments such as disclaimers, CTAs, or address blocks. These components will draw from site settings where relevant, giving editors a simple way to maintain consistency without relying on custom CSS.

To support safe editing, we will provide **draft mode and versioning**: editors can work without impacting live content, and restore older versions if needed. We will also provide a **library of configurable blocks** that span a full row and allow limited adjustments (e.g., CTA text, colors, image). This gives clients structure and helps them produce good-looking websites quickly, even without design expertise.

Some Pages may serve a **special role** (e.g., Contact Page). These will be identifiable in the CMS so templates can link to them automatically, even if the client later changes their URL. And to accelerate onboarding, each new instance will ship with **base Pages and templates** ready to use.

Finally, we need to support **i18n**, including translated URLs and right-to-left content (Arabic). The website shell may stay left-to-right at first, but content itself must render correctly in RTL languages.

Technical note

- Pages map to **Nuxt routes**.
- They are rendered with **SSR (server-side rendering)** for SEO and performance, then hydrated in the browser for interactivity.
- RBAC is applied via **route guards**, ensuring consistent access across direct links and menus.
- SEO metadata and canonical URLs are handled at the Page level.

4.5.2 Dynamic Pages (i.e., Blocks)

Not all content fits into static Pages. To support richer experiences, we introduce **Blocks**, which can be dropped into Pages. Depending on how much structure and visibility they require, Blocks fall into three categories:

- **Simple widgets** are the smallest units, such as a news teaser carousel or a KPI counter. They live inside Pages, don't need their own URLs, and simply render content.
- **Full applications** are larger modules like a Job Center or Alumni Directory. These need their own **first-class routes** (e.g., /jobs, /directory). In the page builder, they appear as a block, but technically they mount their own routes inside the Page layout.
- **Hybrid blocks** sit in between. An agenda, for example, can be placed inside a Page with some introductory text. Navigation inside the block (e.g., switching months) is handled client-side and doesn't create new URLs, but event details still get their own canonical routes (e.g., /event/my-event-123).

To provide flexibility, we will offer **multiple block variants** (e.g., list vs grid) — still selected manually, not dynamically generated by AI.

Technical note

- **Widgets:** SSR for SEO + hydration for small interactions.
- **Full applications:**
    - Plugins declare their **routes** via a manifest (e.g., /jobs, /jobs/post).
    - CMS mounts these as Nuxt nested routes inside layouts.
    - Editors drop a **Route Mount block** into the Page builder, which connects to these routes.

- **SSR + hydration** for all indexable subpages; detail pages (e.g., job detail) get their own canonical URLs.

- **Hybrid blocks:**
  - Base route is SSR'd (/agenda).
  - Secondary navigation can use query params (?month=2025-09) or child routes (/agenda/2025/09)
  - Detail pages always have dedicated canonical URLs (/event/my-event-123) (SSR)

.

4.5.3 Design System

The **back-office** will adopt the existing **One-Platform design system**, ensuring coherence across our internal tools. The **front-office** will also follow a design system, though not necessarily the same, and proposals will need to be delivered during the first study phase by the development team.

Both environments will rely on **design tokens** (colors, typography, spacing, elevation, etc.) to guarantee consistency, allow theme variations, and simplify future maintenance. **Multiple icon libraries** (e.g., FontAwesome and others) will be made available to balance flexibility with developer efficiency.

To allow customization by internal experts or external agencies, **theming and CSS overrides** will be supported, but the scope and guardrails must be explicitly defined (e.g., which layers of the system can be overridden and which cannot).

From a usability perspective, **responsiveness is mandatory**: the front-office must be **mobile-first**, while the back-office must remain **fully usable on desktops and tablets**. Accessibility will be measured against **RGAA standards**, though the exact target score (for front vs. back) is still to be decided at group level.

Decision on target levels should be taken at group level before v1.0

Finally, all front-ends must respect **SEO best practices**, including correct use of canonical tags, metadata management, and semantic heading structures (H1/H2/H3).

4.6 Step 2: Articles, Content Types & Taxonomies

4.6.1 What is an Article

Beyond static Pages, we also need a way to publish shorter, simpler pieces of content such as news, announcements, or publications. These will be managed as **Articles**. An article is similar to a page but much more lightweight: it focuses on text and images only, without dynamic blocks or complex layouts inside the body. The aim is speed and simplicity — one wants to "say something" without building a full page.

Articles are not placed directly in menus. Instead, they are displayed through listing blocks (e.g., "Latest News" on the homepage) and accessed through their own dedicated detail pages. Each article has its own URL and its own access rules, with defaults inherited from its main category to minimize editor effort.

For v1.0, we will start with a few **predefined subtypes** (News, Announcement, Publication). These are not separate schemas but rather labels that may influence how content is displayed or filtered. Over time, we may allow a light schema builder, but for now the focus is on speed and predictability.

Technical note: Article detail pages are SSR'd with canonical URLs and per-locale slugs. Changing a slug automatically generates a 301 redirect. Listing endpoints are indexed in Elasticsearch with pagination capped (12–24 items) to preserve performance.

4.6.2 Article Fields

Articles share a common set of fields:

- **Required**: title, slug, main category, publish status/date, author(s), body (rich text), cover image, SEO title/description.
- **Optional**: secondary categories, free-form tags, attachments (documents).

The editor experience should remain consistent with pages: draft → preview → publish, with version history. AI assistance will help generate titles, excerpts, SEO metadata, and even suggest tags.

Technical note: Validation enforces unique slugs per locale. HTML from editors is sanitized before storage. Cover image metadata includes focal-point selection for responsive cropping. Open Graph and Twitter card metadata are auto-generated.

4.6.3 Taxonomies

Articles are organized through **taxonomies**, providing structure and discoverability.

- **Categories (flat, required)**: every article belongs to one main category, with the option of adding secondary categories. Categories are used for routing, default access rules, and navigation.
- **Tags (free-form, optional)**: flexible labels that editors can add for search and filtering. Drift is accepted in v1.0, as the scale will remain manageable.

In v1.0, categories and tags are **shared globally at the tenant level**. This avoids messy merges if multiple sites exist in the same tenant. Only administrators can create or edit taxonomies, keeping things simple at the start.

Technical note: Categories and tags are tenant-scoped. Rename and soft-delete operations generate redirect mappings where needed. Tags are normalized (trimmed, length-limited) but otherwise free-form.

4.6.4 Listing & Discovery

Articles are discovered via **blocks** that list or promote them. An “Article List” block can filter by type, category, or tag, and sort by date or manual pinning. Several templates will be provided (card grid, compact list, featured article + list).

In addition, **category and tag landing pages will be generated automatically**. These pages serve as permanent entry points for groups of articles (e.g. a “Newsroom” or “Announcements” section) and remove the need for editors to create “fake” listing pages. Each landing page will follow a consistent URL policy (e.g. /category/slug or /tag/slug) to support SEO. Changes to categories or tags will automatically trigger permanent redirects, ensuring stability of inbound links.

4.6.5 Roadmap for Content Modeling (v1.1 → v2)

Articles and taxonomies will then evolve in future iterations.

First:

- Tag management improvements (merge, rename).
- Scheduled publishing and basic pinning for featured content.

Then:

- **Light schema builder** to extend Article subtypes with custom fields (guardrails needed on who can define them).
  (See what directus is proposing https://directus.io/docs/guides/data-model/fields)
- **Hierarchical categories** (parent/child structure).
- **Controlled vocabularies** (predefined facets like “Region”).
- Tag governance features (suggestions, deduplication, analytics).

4.6.6 Open Questions for Feedback

- **Schema builder ownership**: who should be allowed to add custom fields in v2 — R&D only, integrators, or client admins?
- **Category defaults**: should categories support theming/SEO defaults (e.g., default cover image)?
- **Tag drift tolerance**: at what scale should we activate governance tools like suggestions and deduplication?
- **URL strategy**: in v1.0, article URLs will remain flat (/news/<slug>). Should we later support category segments (/news/<category>/<slug>) when categories become hierarchical?

4.7 The Power of Dynamic Blocks Within an Open Ecosystem

The real power of Blocks is their ability to integrate with other services — either internal (other modules of the One-Platform) or external (third-party systems). For example:

- On a donation site, a block might display the live amount collected for a campaign and embed the donation form.
- A CRM form block could capture data directly into the CRM module.
- An event block could list upcoming events, with details linking to an external registration process.

Each Block contains **metadata** to describe itself (name, image, category), configuration parameters (e.g., available campaigns), and endpoints capable of rendering HTML (with standardized CSS classes). JavaScript may be included for complex cases. For SEO and performance, data is fetched by the backend whenever possible (except client-side navigation in hybrid blocks).

To support this, we introduce the notion of a **content plugin**. A plugin describes its capabilities in a **manifest**, including:

- Routes it owns (with flags like indexable, ssr, cacheability).
- Embeddable views it offers (widgets) and their SSR status.
- SEO metadata providers (title, description, breadcrumbs).
- RBAC scopes it requires.

The CMS uses this manifest to mount routes, enforce guards, and expose blocks to editors. Editors can then either embed a plugin as a widget, or mount its routes into layouts as a Route Mount block.

This model makes the CMS **extensible by design**: new modules can be added without rewriting the core, while ensuring SEO, permissions, and navigation remain consistent.

This system would make it easy for a client to integrate other tools within it, such as the display of information from internal systems or external sources for example.

Technical note

- **Plugin manifest schema**: routes, SSR requirements, cacheability, SEO metadata, RBAC.
- **Route Mount blocks**: block type that connects plugin routes into layouts.
  - In the builder, the editor adds a “Job Center (routes)” block to a page/section.

    - Under the hood, this block **mounts /jobs/*** into that location via Nuxt nested routes.
    - We can still add text above/below (static content stays in the parent page).
- **Implementation sketch (Nuxt + Vue):**
    - Layouts define global chrome (header, footer, slots).
    - Pages render widgets (SSR snippets) or route mounts (nested router-views).
    - Global guards enforce RBAC at the route level.
- Menus are purely navigational: they link to routes and hide items if the user lacks access, but never control access themselves.

A bit more details about the different cases we will have:

- /agenda → SSR entry
- /agenda/2025/09 → SSR (discoverable month)
- /jobs/:id → SSR (detail)
- /agenda?sort=date&page=2 → SSR base + client transition (secondary state)

4.8 Step 3: Multiple Websites/Tenants

4.8.1 Definition of a Website

Up to now we only talked about the content (static or dynamic) but not the website itself: header, menu, footer. We have to consider the notion of a website, which is:

- A dedicated URL (independent domain/sub-domain)
- A theme (from a catalog) and theme configuration (colors, fonts, logos, header/footer variants, login on/off)
- Site settings (entity name, postal address, email, phone, legal info) that can be used in the theme
- Menu(s) (site-level navigation; blocks inside pages can still filter content).

Note that concerning the header, footer, menu and other big components they would not be free, the user can't change everything, we remain within the allowed configuration of that theme.

For example, let's imagine a Eudonet's client that is using the One-Platform solution in the ESR sector:

- I don't want to replace the website of the school.
- But I need a website to provide a registration form for new students with contextual information about the procedure.
- I also need a website for my alumni association to animate the community and make if vibrant.
- That association also has a magazine with a specific old-newspaper type of design.
- I also need a website for the school foundation to organize the collect of donations
- This year the students want to collect donation of a local community association and need a website to present their project.

We can see how offering multiple websites would be a real differentiator for that client!

4.8.2 Sub-sites and Channels

Sub-website:
A sub-website has a website as a parent and inherit its theme and theme configuration that can't be changed. There can be different types of sub-website, "group" is one of them.
Two variations are possible:

- On an independent domain/sub-domain with its own menu and possible overload of settings (e.g. mini-site).
  Example: mini-website for groups that currently exist in Netanswer
- On a sub-path (e.g. /group/finance-sector), it's then only a menu below the main menu
  Example: group page within the main website that we have in Netanswer

Channel:

A website or sub-website is by itself a channel or publishing target. So is the mobile app, an RSS feed or a newsletter.
The set of channels is defined globally and has its hierarchy, for example:
Registration Website (website+channel)
Alumni Website (website+channel)
> Alumni Mobile App (channel)
> RSS Feed to the School website (channel)
> Finance sector (sub-website of type "group")
> Promo 1980 (sub-website of type "group")

The choice of channels is made within the article: sites can still subscribe to taxonomy filters (e.g., "News: Alumni"), but an article can explicitly target one or many channels.

As we don't necessarily want pages to be visible in each website by direct URL access (some information is tenant-specific and should not be available elsewhere), within each page one can choose to make it available globally or only for specific websites (by default only for the current website)

4.8.3 Multi-sites and Tenancy

Now that we have multiple websites, we are going to have tenancy and access issues. If we continue the same example:

- How can I separate the school website from the alumni one in terms of content?
- How can I ensure that I can give access to a student to administer the donation to the local community without him breaking anything/accessing else?
- How can I give a user access only to the perimeters of his own professional or promotion groups?

Each content (page or article) belongs to a website and can then be shared or not with other websites with a default mode configured at the website level.
This multi-tenancy means that when working on the CMS the choice of the current working website must be made explicit and clearly visible.
If a content is shared with other websites, then on these other websites it must be accepted, as simply as "allow from other websites" when configuring the dynamic block. No specific approval on a case-by-case basis.

Content from a sub-website is automatically shared with the website above.

4.9 Step 4: Integrate a Private Space

4.9.1 Introduction / Vision

The **private space** is a dedicated application that manages all authenticated member features (profile, directory, events, job board, donations, etc.). It is not part of the CMS itself but exposes its routes via a manifest, just like any other application. This keeps responsibilities clear: the private space manages user identity and permissions, while the CMS focuses on content and presentation.

The CMS must nonetheless be aware of whether a user is logged in and be able to adapt blocks or pages accordingly. The goal is a smooth integration where the CMS provides login components and personalization features, while the heavy lifting (authentication, sessions, permissions) is entirely delegated to the private space.

4.9.2 Authentication & Sessions

The CMS does not manage identities. Instead, it always delegates authentication to the private space, using standard SSO protocols (OAuth2/OIDC).

- A native login component in the CMS (widget or page) allows users to connect.

- Behind the scenes, the CMS delegates the login flow to the private space.
- Once authenticated, the CMS simply consumes the identity token to adapt the experience.

This delegation model simplifies the CMS and centralizes user management in one place. It also introduces a dependency on the private space's availability and performance, which must be considered.

4.9.3 User Data in CMS

For personalization purposes (e.g., "Hello Alice," displaying a profile picture, showing a user email), the CMS requires minimal user profile data: name, email, phone, and picture.

There are two options for handling this data:

- **Event-driven sync**: the CMS stores a copy of minimal user data. When a change occurs in the private space (e.g., the user updates their profile picture), an event is sent to refresh the CMS copy.
    - *Pros*: faster access, user info available even if private space is under load.
    - *Cons*: requires synchronization, risk of stale data if events are delayed or lost.
- **On-demand fetch**: the CMS does not store user data. Each time a personalized block is rendered, it makes a small API call to the private space to fetch the required information.
    - *Pros*: data always fresh, simpler system (no sync).
    - *Cons*: adds latency for every personalized block, less cacheable.

This choice impacts performance and caching strategies, and will need to be evaluated during implementation.

4.9.4 Block Behaviour when Logged In

Blocks in the CMS may render differently when a user is authenticated. Examples include:

- Showing "Hello Alice" in a header banner.
- Displaying personalized content, such as events relevant to the logged-in member.
- Restricting donation history to the authenticated user.

Technically, the CMS backend will send the authenticated user context to the external service powering the block. This has caching implications:

- Pages viewed anonymously can be cached at the CDN/edge.
- Pages with personalized blocks cannot be cached in the same way, since they depend on the logged-in user.

This reinforces the need for a clear strategy distinguishing **public pages** (fully cacheable) and **authenticated pages** (private, API-driven).

4.9.5 RBAC & Permissions

In this model, the CMS does not manage detailed member permissions (e.g., access to directory, jobs). Those checks remain entirely in the private space.

However, the CMS does need to handle a **limited set of states** to control page visibility and block rendering:

- **Authenticated vs. anonymous**: whether the visitor is logged in.
- **Membership status**: whether the logged-in member has a paid/active membership.
- **Other simple states**: a defined list of statuses or flags (e.g., "donor," "student," "alumni") that may be used to restrict access to specific pages or blocks.

For example, the CMS could display a “Premium Articles” page only to logged-in users with an active membership, or show a personalized banner to alumni while hiding it from students.

This allows the CMS to adapt navigation and content visibility without embedding the full complexity of member permissions. The private space remains the source of truth for these states, while the CMS consumes them to enforce simple access rules at the page level.

4.9.6 Implementation Options

We see two possible paths for the private space itself:

- **Option A: Convert the current Netanswer private space into a service-oriented app**
  - *Pros*: reuses mature, proven functionality; less work initially.
  - *Cons*: legacy code, deeply integrated with other systems, harder to modularize cleanly.
- **Option B: Rewrite the private space from scratch**
  - *Pros*: clean architecture from day one, aligned with the new CMS and One-Platform direction.
  - *Cons*: initial loss of functionality, higher development risk, longer to reach parity.

At this stage, the decision is deferred. We will first validate the new CMS rewrite and explore how AI-assisted development can accelerate complex rewrites. Once proven, this will inform whether a full rewrite of the private space is feasible.

4.9.7 Conclusion

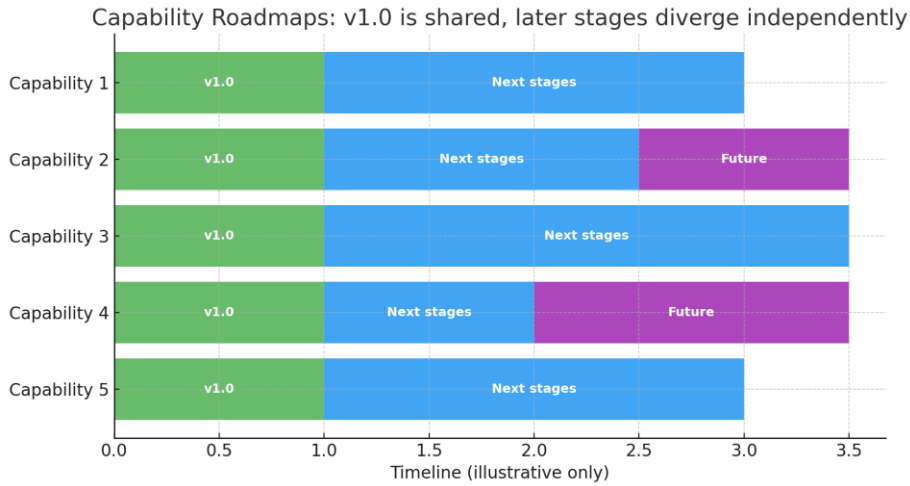
The private space is a critical piece of the One-Platform but sits **next to the CMS**, not inside it. The CMS provides login components, consumes identity from the private space, and adapts blocks/pages depending on authentication state. Minimal user profile data may be stored or fetched on demand, with trade-offs to be evaluated.

Caching strategies will need to account for public vs. authenticated pages. Permissions remain entirely in the private space. Two paths exist for the private space implementation (reuse vs. rewrite), and input is sought to identify potential blockers early.

5 A FULL CMS

In the last section we described an overall view of what we envisioned for this new CMS, but we have a few maybe more technical subjects to address to have a great CMS.

Before diving into each capability area, it is important to clarify how the phased roadmaps should be interpreted. For every domain (e.g., Articles, Search, Media, Workflows), we outline a progression with labels such as v1.0, v1.1, v2, v3. These labels represent the **maturity stages of that capability**, not a unified global release plan. Beyond the initial v1.0 delivery, priorities and sequencing will be determined by Product Owners based on business value, client demand, and technical readiness. As a result, "v2" in one section may not align chronologically with "v2" in another — they evolve independently.



5.1 Internationalization (i18n)

Supporting multiple languages is essential for our clients; the CMS will rely on **AI-powered translations with human corrections when needed**, combined with localized URLs and SEO features, while keeping the editorial workflow as simple as possible in early versions. This approach is **optimized for efficiency rather than perfect linguistic quality in v1.0**, with stronger editorial controls planned for later versions.

Note: The phases below describe the evolution of this capability. They do not represent the global release sequence of the CMS; prioritization beyond v1.0 will be set by Product Owners.

v1.0 (baseline)

- **Single content with variants**: Each page or article exists once, with translations as variants.
- **AI-powered translations**: Content is automatically translated into the languages configured for the site. Human edits are possible but not required.
- **Website-level settings**:
  - Main language chosen per website.
  - Allowed languages also configured per website.
- **Interface language for administrators** is set per profile (independent of website language).
- **Per-locale slugs**: URLs are localized (/en/news/my-article, /fr/actualites/mon-article).
  - For non-alphabet languages (e.g., Chinese), we can either:
    - Use transliteration to generate slugs (e.g., /zh/xinwen/wo-de-wenzhang).
    - Or allow numeric/short IDs in the slug for simplicity (e.g., /zh/12345).
  - Default approach: transliteration with fallback to ID-based slug when transliteration is not practical.
  - Once published, editors are warned not to change them to avoid breaking links.

    - If a slug change is unavoidable, the system will automatically create a **301 redirect** to preserve SEO and inbound traffic.
- **SEO readiness**: Automatic hreflang tags and localized sitemaps.

v1.1 (short-term enhancements)

- **Slug customization per locale**: Editors can override AI-generated or transliterated slugs if needed.
- **Preview in multiple languages**: Ability to see the page/article in each language before publishing.

v2 (advanced workflows)

- **Translation status tracking**: Each language variant can have its own workflow state (e.g., *AI Draft*, *Reviewed*, *Published*).
- **Configurable approvals**: Certain languages (e.g., "official" versions) may require human validation before publishing.

5.2 RBAC (Role-Based Access Control)

A user's permissions are scoped to a website. Permission to a parent propagate in the children in the hierarchy.

Roles available:

- **Site Owner**: full control of site settings and theme config.
- **Site Manager**: manage menus, pages,; cannot change theme catalog or global settings.
- **Publisher**: approve & publish to this site's channel(s).
- **Editor**: create/edit content assigned to this site; cannot publish. The two roles of Editor/Publisher could be merge into one, decision to be made there.

These roles are assigned website by website (or sub-website).

5.3 Editorial Workflows & Governance

In addition to role-based access control (RBAC), we need to manage how content moves from creation to publication. To balance **simplicity in v1.0** with **room to grow later**, we propose the following staged approach:

Note: The phases below describe the evolution of this capability. They do not represent the global release sequence of the CMS; prioritization beyond v1.0 will be set by Product Owners.

v1.0 (baseline)

- Simple workflow: *Draft* vs *Published*.
- Any authorized user with the right role can publish content directly.
- Basic version history (last saved states) included.

v1.1 (short-term enhancements)

- **Scheduling**: when clicking "Publish," users can optionally set a future date/time for automatic publishing (and later unpublishing).
- **Audit trail**: start recording who published/unpublished content and when.
- **Extended version history**: more granular tracking of content changes.

v2 (advanced governance)

- **Multi-step workflows**: e.g., *Draft* → *In review* → *Approved* → *Scheduled* → *Published*.

- **Configurable approvals**: rules per site/channel (e.g., content must be approved for Site A and Site B separately).
- **Flexible role mapping**: “Author” content requires review by “Editor/Admin” before going live.
- **Content governance windows**: optional ability to enforce content freezes during sensitive periods (major events, migrations).

This phased approach ensures we cover immediate needs without over-engineering, while leaving a clear path for stronger governance as larger clients and multi-site deployments demand it.

5.4 Search

Search is a critical feature both for **end-users (front-office)** and for **administrators (back-office)**. Our approach is to deliver robust search capabilities early with Elasticsearch, while planning a roadmap toward richer indexing and semantic search.

Note: The phases below describe the evolution of this capability. They do not represent the global release sequence of the CMS; prioritization beyond v1.0 will be set by Product Owners.

v1.0 (baseline)

- **Front-office**: Site-level search covering pages and articles.
- **Back-office**: Global search across the websites an administrator has rights on.
- **Search engine**: Elasticsearch as the default backend, as we have a good experience with it
- **Permissions-aware**: Search results respect RBAC rules (e.g., member-only content not shown to anonymous visitors).

v1.1 (short-term enhancements)

- Extended indexing of additional metadata (author, publish date, tags).
- Basic result ranking configuration (e.g., boosting by recency or category).

Alongside the initial search rollout, we will introduce early governance features: basic synonym dictionaries per locale, sensible stopword/analyzer defaults, and a simple “no-results queries” dashboard. This lightweight layer gives editors and product teams the means to improve search relevance from day one, without waiting for the more advanced v2–v3 features.

v2 (expanded search)

- Indexing of dynamic content (e.g., directories, events, jobs) in addition to static pages and articles.
- **Search analytics**: track top queries and zero-result queries to guide content strategy.
- Multi-language analyzers to improve results in non-English locales.

v3 (advanced search)

- Semantic/AI-powered search using embeddings for natural language queries.
- Synonyms and related-term support.
- Faceted navigation and filters (e.g., by date, type, taxonomy).
- Experimentation with alternative engines (e.g., Meilisearch, Algolia) for specific scenarios.

This roadmap ensures we deliver **useful search from day one**, while leaving room for more advanced capabilities as clients demand richer search experiences.

5.5 Media & Digital Asset Management (DAM)

A **Digital Asset Management (DAM) system** is a structured library for storing, organizing, and reusing digital assets such as images, videos, and documents. For our CMS, we will adopt a phased approach, starting with simple media handling and progressively expanding toward a full DAM as client needs grow.

Note: The phases below describe the evolution of this capability. They do not represent the global release sequence of the CMS; prioritization beyond v1.0 will be set by Product Owners.

v1.0 (baseline)

- **Unlayer image library** used as the default picker for editors (sufficient for logos and frequently reused images).
- Basic image uploads for articles and pages, stored in an **S3-compatible backend** with on-the-fly resizing.
- Media storage **isolated per site** to keep things simple and avoid unintended sharing across websites.

v1.1 (media lite)

- Per-asset basics for images only: title, alt text, credit/rights, and a simple free-tag field.
- Lightweight "collections" to group images for reuse (e.g., a hero set).
- Minor improvements to the editor picker to surface collections and tags.

v2 (enhanced media library)

- Introduction of a lightweight **media library** beyond Unlayer, supporting collections of images (e.g., event photo galleries).
- Ability to browse and search assets more easily, with AI auto-tagging and **AI-generated alt text** for accessibility and SEO.
- Optional integration of the media library into Unlayer, so editors can pull assets directly while designing.

v3 (full DAM including documents)

- Expansion of the media library into a **full DAM** supporting documents (PDF, Word, Excel) alongside images.
- Support for metadata (title, description, tags) and preview where possible.
- Ability to display **document libraries** on websites (e.g., publications, reports, downloadable resources).
- Advanced DAM features (expiry dates, duplicate detection, quotas) may be added progressively depending on client demand.

This phased approach ensures we start with a **simple, practical solution** that covers 80% of day-to-day needs, while leaving a clear path to evolve into a more complete DAM for clients requiring richer media and document management.

5.6 Redirect Management

Redirects are essential both during **migration** and in the ongoing life of a website (e.g., when restructuring content, renaming pages, or consolidating URLs). The CMS should provide a simple but powerful redirect management system.

Note: The phases below describe the evolution of this capability. They do not represent the global release sequence of the CMS; prioritization beyond v1.0 will be set by Product Owners.

v1.0 (baseline)

- Automatic generation of **1:1 redirects** during migration, ensuring old URLs continue to work and preserving SEO.
- Ability for administrators to manually create **simple redirects** (301, permanent).

v1.1 (short-term enhancements)

- **Bulk import/export** of redirects (CSV/Excel) to support large-scale restructuring.
- Redirect reporting: detect and list broken links or loops.

v2 (advanced features)

- **Redirect rules** (wildcards, regex) for more complex scenarios.
- Redirect analytics: measure traffic to redirected URLs to help phase out legacy paths.
- Integration with site maps and SEO tools to monitor the impact of redirects.

This ensures that redirect management is not treated only as a one-off migration need, but as a **core part of site governance** over the long term.

5.7 Themes and Variations

5.7.1 Why Themes (and Variations) Matter

Every client wants their website to reflect their own brand identity. The challenge is that our CMS must support very different needs: some clients expect a polished site "out of the box," while others want fine control over colors, fonts, and even subtle details like button shapes. To meet this diversity, we need a flexible but safe theming system.

A **theme** defines the overall look and structure of a site. It includes templates for big components like headers, footers, and menus, along with a base set of design parameters (colors, fonts, logos). Themes are "big pieces" that set the foundation, such as "a donation-first mini-site" or "a full association portal with login."

On top of themes, we introduce **variations**. A variation is a lighter adjustment of a theme that fine-tunes its style—rounded vs square buttons, black-and-white vs colorful, formal vs playful. Variations can usually be created by a designer without developer involvement and become part of our standard offer.

Beyond that, we allow **custom CSS** rules as an escape hatch. This gives clients the ability to tweak details that aren't covered by theme settings or variations—for example, aligning a logo, adjusting spacing on a specific section, or styling a partner logo differently. Because direct CSS edits are powerful and risky, we will enforce guardrails: size limits, validation, and version history.

Client scenarios help illustrate the balance:

- A **small NGO** can select a ready-made theme, apply its own colors and logo, and be online within hours without touching any code.
- A **large alumni association** might start from the same theme but commission a designer to create a formal variation, then add a handful of CSS rules to integrate partner branding on specific pages.

This layered approach balances **consistency and flexibility**. Themes and variations provide ready-made professional designs that keep sites looking coherent, while custom CSS gives advanced users freedom without undermining maintainability.

The trade-off is clear: too much flexibility leads to unmanageable sites and poor performance; too little flexibility frustrates clients who want their site to "feel like theirs." By clearly defining what a **theme**, a **variation**, and **custom CSS** mean in our system, and by offering the right guardrails, we can give clients choice without sacrificing reliability, scalability, or supportability.

While custom CSS can be a powerful tool, it also creates long-term support and performance risks if used without restraint. For this reason, custom CSS will be **disabled by default** in entry-level packages. It will only be available as an **advanced option**, explicitly enabled per site when there is a proven need. This approach ensures most clients benefit from consistent design tokens and guardrails, while still allowing flexibility for advanced cases under controlled conditions.

5.7.2 Advanced Client Scenario - Custom Themes

Some clients may want to go beyond the flexibility of theme variations and custom CSS, and instead build their own **full theme**. In this model, the client (or their agency) develops templates, layouts, and styling outside of our standard theme catalog. This gives them **total control** over design and behaviour, but it also means they are responsible for maintaining their theme against future changes in the CMS.

The trade-off is clear:

- **Pros:** maximum freedom, unique branding, ability to implement very specific layouts or interactions.
- **Cons:** any **internal changes** to the CMS (new APIs, component updates, design system adjustments) will not automatically apply. The client's theme may require updates, delaying their releases. They must accept this as an **extra cost of ownership**.

In practice, we would treat these as **"custom theme projects"**, with clear rules:

- The client owns the code and its maintenance.
- We provide stable APIs and documentation, but **backward compatibility is not guaranteed** for custom themes.
- If a release is delayed because their theme needs adjustments, this is their responsibility and part of the cost of going beyond our standard model.

This option gives clients a path to maximum independence, but with the understanding that it shifts responsibility from our product team to their own technical resources.

To manage expectations clearly, we could distinguish two levels of support:

- **Standard Themes and Variations**: fully supported by our product team. Updates, bug fixes, and improvements are applied automatically across all clients using these themes. Clients benefit from new features and system changes with no additional effort.
- **Custom Themes**: owned by the client or their agency. We provide documentation and stable APIs where possible, but **compatibility with future releases is not guaranteed**. Any adaptations required to keep their theme aligned with system changes are the client's responsibility. This may delay upgrades and generate additional costs.

This approach ensures that clients who stay within our supported model benefit from speed and reliability, while those who choose full independence accept the trade-off of slower releases and higher maintenance overhead.

5.7.3 Functional Overview of the Proposed Solution

The proposed theming system is designed to support three groups of actors: **our development team**, **designers/UX specialists**, and **clients**. Each group has clear responsibilities and boundaries, ensuring flexibility without losing maintainability.

For development (our team):
Themes are produced and maintained internally. Each theme is a packaged foundation that defines structure (header/footer variants, layout templates) and a base set of design variables. Developers ensure themes are technically sound, consistent with our design system, and compatible with the CMS architecture. They also publish the standard catalog of themes available to all clients.

For UX/UI and designers:
Designers focus on **variations**, not themes. Starting from a theme, they can adjust the style by changing parameters (colors, typography, button shapes, spacing) to produce "lighter" alternatives. A variation can be private to one client or, if broadly appealing, added to the standard catalog. Variations are created directly in the admin with an import/export mechanism, so a designer can duplicate an existing variation, adjust it, and reapply it without developer involvement.

For clients (administrators):
When creating a website, an admin selects a theme from the catalog. They then choose a variation — either one of the standard options or a custom one available to their tenant. From there, the admin can fine-tune parameters in a dedicated **Theme Settings panel**: brand colors, fonts, button styles, shadows, spacing, etc. To ensure safety, any change goes through a **draft and preview workflow**: admins can test a variation privately without impacting end-users, then publish once satisfied. Variations can also be reused across multiple websites in the same tenant, though only the creator can edit them.

Custom CSS as an escape hatch:
In addition to themes and variations, we allow direct CSS rules to be added for very specific adjustments. This is intended for small tweaks — aligning a logo, changing the look of a single section, integrating a partner's branding. Because this feature is powerful and potentially risky, it comes with guardrails: file size limits, validation against unsafe rules, live preview, version history, and rollback. The system will encourage admins to use Theme Settings and variations first, keeping custom CSS as a last resort.

Under the hood (simplified):
All these mechanisms work by updating a shared set of **design variables**. Themes define a default set of values, variations override them, and Theme Settings in the admin let clients adjust them further. Custom CSS then applies on top, scoped to the website. This ensures consistency across all components — headers, footers, blocks, and forms — while making it easy to preview and rollback changes.

The lifecycle of theming in practice:

1. Developers publish and maintain the theme catalog.
2. Designers create variations in the admin, either for a specific client or as part of the standard library.
3. Clients pick a theme and variation for each website.
4. Admins fine-tune parameters through Theme Settings and preview changes before publishing.
5. For exceptional cases, admins add custom CSS with safety limits.
6. If a client wants complete independence, they can develop a custom theme, accepting slower updates and higher maintenance costs.

This functional model gives everyone the right level of control: developers ensure stability, designers extend style options, and clients tailor the look of their site — all while keeping risks manageable and preserving long-term maintainability.

5.7.4 Technical Proposal for Theming

Core principles

- **Design tokens as the foundation**: all design properties (colors, typography, spacing, radii, shadows, motion) are expressed as tokens. Tokens are compiled into **CSS variables**, which are the single source of truth for styling.
- **Headless components**: Vue components provide logic and structure only. Styling is applied via tokens, so components stay consistent across themes and variations.
- **Themes = token sets + templates**: a theme is primarily a JSON of token values and a small set of structural templates (header/footer variants, section layouts). Themes are not separate codebases.
- **Variations over forks**: most customization happens by overriding token values. New themes are created only when page structure truly differs.

Token pipeline

- Tokens are stored in JSON (e.g., tokens.json).
- At build/deploy, tokens are transformed into CSS variables (--color-brand, --radius-md).
- On theme change in the admin, a **site-specific CSS override** is generated or injected dynamically, so updates don't require redeploys.
- Example:

```
JSON                                                                    :
{
  "color.brand":                                                "#0F6FFF",
  "color.brand-contrast":                                       "#FFFFFF",
  "radius.md":                                                      "8px"
}
CSS                                                                     :
:root                                                                   {
  --color-brand:                                                  #0F6FFF;
  --radius-md:                                                        8px;
}
```

Component styling

- Components reference CSS variables directly:

```
<template>
  <button                                          class="btn"><slot/></button>
</template>
<style>
  .btn                                                                  {
    background:                                           var(--color-brand);
    border-radius:                                          var(--radius-md);
    color:                                          var(--color-brand-contrast);
  }
</style>
```

- If Tailwind is used, tokens are mapped into Tailwind config (theme.extend) so utility classes stay aligned.

Theme provider

- On site init, the CMS loads the selected theme + variation.
- Token overrides are applied to :root (or via data-theme="theme-name").

Templates

- We ship a curated library of section templates (hero, grid, split, CTA band) and header/footer variants.
- Editors select variants; options change token sets and layout toggles, not raw CSS.

Governance and guardrails

- **When not to fork**: colors, typography, radii, logos, and spacing should be solved with token overrides or variations.
- **When to fork**: only if structure is fundamentally different (e.g., donation mini-site vs association portal).
- **Custom CSS**: limited to a small site-scoped file. Guardrails:
  - No !important overrides on core tokens.
  - No global resets or wildcard selectors.

- Lint checks before saving.
- File size limit (~10–15 KB).

Theming for plugins and widgets

- Plugins must style exclusively via tokens (CSS variables).
- We provide a **token map** documentation (color, typography, spacing) for plugin developers.
- If extra classes are needed, they must be prefixed (e.g., .na-job-*) to avoid collisions.
- Utility class map (.btn, .text-primary) ensures Unlayer content uses the same tokens.

Admin UX implications

- Admin panel offers a **Theme Settings panel** with:
  - Brand colors, typography, button shape, radius, shadows, spacing.
  - Desktop/Tablet/Mobile preview modes.
  - Reset to defaults + save as new variation.
  - Import/export variations as JSON.
- **Draft/preview workflow**: variations can be tested privately before publishing to live users.
- **Versioning**: keep rollback options for both variations and custom CSS.

Performance considerations

- Tokens → CSS = very lightweight; variations only override variables, no new bundles.
- Deliver one component library + many small token files. Avoid multiple theme bundles.
- Inline critical CSS for above-the-fold content; serve the rest via CDN.
- Custom CSS and variation files are cacheable, invalidated on publish.

5.7.5 More About Custom CSS

While themes and variations cover most branding needs, some clients will want finer control over the look of their site. To support this, we will allow administrators to add **Custom CSS** directly from the admin interface. This provides an "escape hatch" for advanced styling needs, but it comes with **strict guardrails** to protect maintainability, performance, and security.

Policy

- **Site-scoped only**: Custom CSS applies only to the specific site where it is added.
- **File size cap**: limit to ~10–15 KB of CSS.
- **Scoped selectors**: CSS must target elements inside a site root container (e.g., #site-root …) to avoid impacting admin tools or embeds.
- **Token-friendly**: encourage the use of design variables (tokens) instead of hard-coded values, so changes remain compatible with theme/variation updates.
- **No globals**: disallow resets and wide selectors like body { … } or * { … }.
- **No unsafe overrides**: block !important on token-based variables.
- **Safe hooks**: if we expose utility classes (e.g., .page--home, .section--hero), clients are required to target these instead of arbitrary deep selectors.

Admin Experience

- **Theme first, CSS last**: The admin UI will emphasize Theme Settings and variations as the primary tools. The "Advanced → Custom CSS" editor is a last step for fine control.
- **Live preview**: admins see changes in real time, with device toggles (desktop, tablet, mobile) and basic accessibility checks (contrast).
- **Inline linting**: warnings appear if admins attempt risky patterns (e.g., global resets, forbidden properties).

- **Documentation panel**: provides a list of safe class hooks (header, hero, CTA) to guide styling.
- **Preview & publish workflow**: admins can save a draft, preview it privately, and only publish once approved.
- **Version history & rollback**: keep the last 5 published versions for quick rollback.
- **Audit trail**: log who edited the CSS and when.

Technical Guardrails

- **Validation**: when saving, the system strips or rejects forbidden patterns (@import, global resets, unsafe positioning).
- **External resources**: block external URLs in url(http…) unless explicitly allowlisted (e.g., CDN-hosted assets).
- **Scoping enforcement**: optionally apply PostCSS to auto-prefix selectors with #site-root if missing.
- **Delivery**: custom CSS is served as a separate cached asset and CDN cache is purged on publish.
- **Performance budget**: warn if CSS exceeds the size cap or adds more than ~100 new rules.

When to Use Custom CSS

- **Use tokens/variations for**: brand colors, typography, button shape, spacing, shadows, links, and global look.
- **Use custom CSS for**: small, one-off tweaks like adjusting the opacity of a hero banner, aligning a logo, or modifying a single section layout.
- **Do not use custom CSS for**: redesigning headers, footers, or menus. Those changes require a new template variant or a custom theme project.

Optional Extras

- **Per-page CSS**: small textarea (max ~2 KB) scoped to a single page, with selectors prefixed by .page--<slug>.
- **Access tiers**: custom CSS could be read-only in standard packages and editable only for advanced or premium clients, with a “break glass” toggle.

Custom Fonts

Some clients will expect their website to use specific brand fonts. While our default catalog will include a curated set of safe, performant options (system and open-source fonts), we foresee the need to support **custom fonts** in the future.

The principle would be to allow clients to either **upload font files** (with licensing confirmation and size limits) or **reference a hosted service** (Google Fonts, Adobe Fonts). To protect performance and maintainability, we would enforce guardrails such as limiting the number of custom fonts, applying font-display: swap, and scoping fonts per site.

At this stage, the details of implementation (storage, validation, caching) are not finalized, but the direction is clear: support brand identity where needed, while ensuring speed, legality, and stability.

5.8 Analytics and A/B Testing

5.8.1 A Self-hosted Analytics Platform

We must integrate an analytics solution from the start.
We can offer Google Analytics, but its use in Europe raises compliance concerns in many configurations.

We should propose a solution like Matomo (https://matomo.org) mature and with a solid API that would allow us to seamlessly integrate analytics inside the administration and then redirect the user to Matomo for more

advanced usage.
The access rights for Matomo should be studied: we don't want to overcomplicate things and we simply want to inform the clients of the consequences of giving access to Matomo.

By using the self-hosted instance of Matomo we could provide, beyond the CMS, an analytic platform about the usage of the One-Platform which would be a great addition for our clients to understand and analyze their own usage of the platform (and the data would be available for us too of course).

To go further, we could also set up an A/B testing of pages or articles and measure the impact with Matomo, particularly useful for donation form.
It would require tracking statistics between the CMS and the donation module but the insights are worth it.

5.8.2 Consent Management Platform (CMP)

To ensure compliance with GDPR and support future personalization features, the CMS will integrate a Consent Management Platform. The CMP will allow end-users to grant or withdraw consent across categories (analytics/Matomo, personalization, marketing tags). This creates a consistent user experience, provides clear auditability, and prepares the platform for evolving consent requirements in the EU.

5.8.3 Events Specification

Alongside the integration of Matomo and CMP, we will define a **lightweight events specification** to ensure consistent analytics across the CMS and plugins. This spec will cover:

- **Event naming conventions and namespaces** (e.g. article.view, menu.click)
- **Standard payload fields** (site ID, user role if applicable, timestamp, locale)
- **PII policy** (no personal identifiers, only anonymous/session-level tracking)

By aligning on this minimal schema early, we avoid fragmented data models and ensure that analytics remain coherent and privacy-compliant as the platform scales. This events spec will also be **published as part of the developer documentation**, so plugin authors and integrators have a clear reference for instrumenting analytics correctly.

Sample events (illustrative):

- article.view → { siteId, articleId, locale, timestamp }
- form.submit → { siteId, formId, success: true/false, locale, timestamp }

5.9 A Focus on Performance and Security

Performance and security are not optional add-ons: they are **foundational requirements** for the new CMS. Fast, reliable websites directly impact business outcomes — higher donation conversion rates, better engagement, and stronger SEO rankings. At the same time, robust security is essential to maintain client trust and protect sensitive data.

The platform must also **scale to handle high-traffic websites**, which requires careful use of SSR, caching strategies, and a global CDN footprint. Finally, because our system is open to internal and external plugins, we must enforce strict security boundaries to prevent vulnerabilities. By treating performance and security as first-class concerns, we ensure the CMS can serve both small associations and enterprise-scale clients with confidence.

We therefore define **clear performance budgets and security rules** that apply consistently across all Pages, Blocks, and Plugins, ensuring the CMS is both high-performing and resilient.

An EU-based CDN (ideally France-hosted) should be used to serve all static files and media to preserve our servers and faster delivery to our clients.

5.9.1 Performance

To ensure the CMS delivers a consistently fast and reliable experience, we define clear performance rules. These guardrails make sure sites scale under heavy traffic, remain SEO-friendly, and provide a smooth experience for end-users.

A Few Principles

The performance of the CMS must be protected by clear operational principles. These define the baseline expectations for payload size, response time, and resource usage.

Route-level caching

- Every plugin route declares a caching policy in its manifest (public, private, or none) and an optional time-to-live (TTL).
- Example: a public news list can be cached for 2–5 minutes, while member dashboards remain private and uncached.
- This avoids unnecessary server load while keeping content fresh.

Edge and CDN strategy

- HTML and data responses respect the declared caching policy.
- A global, EU-based CDN ensures assets and images are delivered quickly under high traffic.
- Images are optimized automatically (WebP/AVIF formats, srcset, lazy loading) with on-the-fly resizing

Data-fetch SLAs

- Each page should collect all needed data from the backend in **under 250–400ms** (95% of cases).
- This keeps the **Time To First Byte (TTFB)** — the moment the browser first starts receiving the page — below **600ms for European users**.
- Low TTFB means pages feel responsive and load reliably even at scale.
- Any slower integration must provide **loading skeletons** and/or degrade gracefully.

Payload budgets

- Initial HTML responses should stay below **80 KB gzipped**.
- Initial JavaScript should stay below **200 KB gzipped** for public pages, with heavier code split and loaded only when needed.
- This keeps page loads light and avoids overwhelming users on slower connections.

List and pagination constraints

- Large lists (news, events, directories) are capped (e.g., 20–24 items per page).
- Server enforces limits to prevent oversized responses that slow down rendering.
- Pagination can still be offered, but always within these safe boundaries.

Observability and regression control

- We track **Core Web Vitals**, Google's standard for user experience:
    - **LCP (Largest Contentful Paint):** main content appears within 2.5s for 75% of users.
    - **CLS (Cumulative Layout Shift):** layout doesn't "jump" unexpectedly (target < 0.1).
    - **INP (Interaction to Next Paint):** site reacts to clicks and typing within 200ms for 75% of users.

- Automated checks (Lighthouse CI) run before release to catch slowdowns early.
- Real-user monitoring (RUM) in production provides continuous feedback from actual site visitors.

A Few Uses Cases

To illustrate how performance principles apply in practice, here are a few representative use cases that highlight caching, SSR, and personalization challenges.

Example: News Page Under Heavy Traffic

Imagine a client publishes a high-profile announcement and traffic spikes. Thanks to caching and CDN delivery, the first visitor request generates the page in under 400ms, and subsequent visitors are served cached HTML instantly. The initial page load stays under 200 KB, so even on mobile networks it loads quickly. Core Web Vitals confirm: the headline and image appear in under 2 seconds (LCP), the layout stays stable (CLS < 0.1), and interactions like "Read more" respond instantly (INP < 200ms). This ensures the page remains **fast, SEO-friendly, and resilient** even when thousands of users hit it at once.

Example: Agenda (Hybrid) Under Load

An association's agenda gets featured in a newsletter and traffic surges. The **base route** /agenda is SSR'd once, cached publicly at the edge for 2–5 minutes, and served instantly to subsequent visitors. Month views are child routes like /agenda/2025/09, each SSR'd on first request and **public-cached** with the same short TTL. The **event detail** /event/my-event-123 is also public-cached. Inside /agenda, client-side interactions (switch month, sort) update the URL (?month=2025-09&page=2) and fetch only the needed JSON, which is **API-cached** separately. Net effect: initial paint is fast, most traffic is served from the CDN, and interactivity stays snappy without re-rendering the whole page.

Personalization & Caching

Many pages mix **public content** (news, agendas, articles) with **user-specific elements** (greeting, favorites, notifications). To keep sites fast and scalable, we **cache the public shell** of the page at the edge/CDN and load **small, private widgets** after the first paint. This preserves SEO and "instant" delivery for most of the page while still providing personalized experiences for logged-in users.

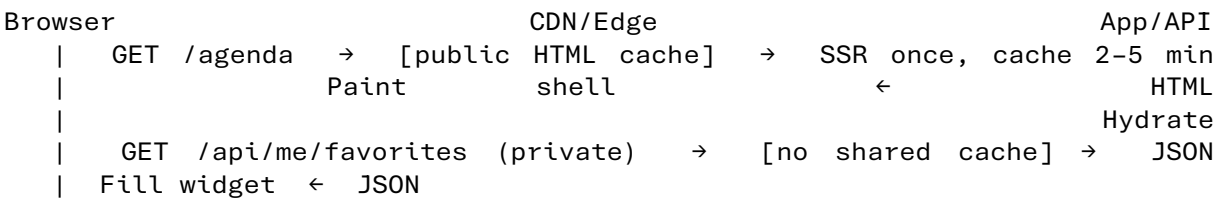
For the users:

- The page appears immediately (headline, body, images).
- Personal widgets fill in a moment later (e.g., "Hi, Alice", "3 new messages") without blocking the first paint.

Technical note

- **Pattern:**
  - SSR and **public-cache** the **page shell** (HTML for layout + public content).
  - Render **placeholders/skeletons** where personal widgets will appear.
  - After hydration, fetch **private JSON** (authenticated API) to fill those widgets.
- **What not to do:** avoid varying full HTML on cookies (Vary: Cookie)—it crushes cache hit rates.
- **API caching:**
  - Public JSON (e.g., agenda list) → edge-cache with short TTL + surrogate keys.
  - Private JSON (e.g., favorites) → no shared cache; keep payloads small and endpoints fast.
- **Plugin manifest (per route/endpoint):**
  - cacheability: public | private | none
  - ttl: <seconds> (for public routes)
  - api_cacheability: public | private (public agenda JSON vs private favorites JSON)
  - ssr: required | optional | never (to know if the shell must SSR)

Sketch



This approach keeps **SEO and first paint fast** (public shell at the edge) while delivering **personalization** via lightweight, private API calls.

5.9.2 Security Guidelines

Because most plugins are **internal** or **client-specific**, our plugin model can stay lighter than a full marketplace-grade sandbox. The priority is to prevent **simple but dangerous mistakes** (e.g., stale manifests, domain takeover) without slowing down development.

A Pragmatic v1.1

As long as plugins are only internals and no third-party, we can postpone that section, but it's required for the v1.1

For the first release with secured plugins, we will apply pragmatic security guidelines that balance speed of delivery with risk management. These include the following measures.

Trust but verify

- When a new plugin is declared in the CMS, it is associated with a **secret key** (provided to the plugin owner but not stored inside the CMS).
- At regular intervals, the CMS sends a **challenge string** to the plugin endpoint.
- The plugin returns its **manifest** plus the challenge encrypted with the shared secret.
- The CMS verifies the response; if it fails, the plugin is automatically disabled.

Basic guarantees

- This handshake prevents someone hijacking the plugin's domain or returning a forged manifest.
- It also ensures the manifest can't silently change without validation.
- No plugin can stay active if it stops answering correctly.

Scope of responsibility

- Plugins are assumed **trusted code** (internal or client-owned), so we don't enforce heavy sandboxing or CSP isolation by default.
- Public pages ship with a **strict CSP**. Plugins may load only the hosts declared in their manifest; any new host requires an admin allowlist update.
- Responsibility for secure coding (input validation, API calls, etc.) stays with the plugin owner.
- As the ecosystem grows, we can add stricter isolation (iframes, manifest-scoped capabilities) later if needed.

Design tokens / theming

Plugins and widgets must style via **design tokens (CSS variables)** provided by the theme; no global CSS overrides. v1.0 allows basic scoping; v1.1 formalizes a token contract.

Future tightening

Beyond v1, we will progressively tighten security as the platform matures. The following measures will be prioritized for later phases.

For now, plugins are either internal or client-specific, which lets us keep security lightweight. If in the future we open the ecosystem to third-party or cross-client plugins, we will add stronger measures such as **sandboxed execution (iframes or shadow DOM)**, stricter **Content Security Policies (CSP)**, **Subresource Integrity (SRI)** for third-party assets and **capability-based manifests**. This evolution path ensures that the CMS can safely grow from a trusted environment to a broader plugin marketplace without re-architecting the foundations.
When/if plugins become cross-client, we'll move to **asymmetric signing** (vendor private key, CMS-stored public key).

Manifest

The plugin manifest is the key mechanism for declaring capabilities and security posture. It must evolve over time to reflect caching, SSR, and SEO requirements.

Here is an example of a possible minimal manifest:

```
schemaVersion:                                                    1
name:                                                       JobCenter
routes:
  -                                path:                        /jobs
    indexable:                                                    true
    ssr:                                                      required
widgets:
  - id: job-counter
```

This baseline manifest makes sure every plugin at least declares:

- **Its name** (so we know what it is).
- **The routes it owns** (so the CMS can mount them and handle SEO/SSR).
- **Any widgets** (so they show up in the block picker).

As the ecosystem matures, we can include RBAC scopes, cacheability, endpoints, events...

```
schemaVersion:                                                    1
name:                                                       JobCenter
owner:                              "B2C                         R&D"
routes:
  -                                path:                        /jobs
    indexable:                                                    true
    ssr:                                                      required
    cacheability:                                               public
    ttl:                                                         300
widgets:
  -                              id:                       job-counter
    ssr:                                                          true
rbacScopes:                      [jobs.read,                jobs.write]
endpoints:
  -                               url:                        /api/jobs
    method:                                                       GET
    access:                                                     public
events:
  -             emits:             [jobs.viewed,             jobs.applied]
  - listens: [user.loggedIn]
```

5.10 Integration & Automation

Beyond content creation and theming, a modern CMS must also integrate seamlessly with the wider digital ecosystem. Instead of building and maintaining one-off connectors (e.g., “publish to Facebook”), the CMS will rely on **event-driven webhooks** that can be consumed by automation platforms such as **n8n**.

This approach is more than a convenience — it is a **first-level capability** that keeps the CMS lean, while opening the door to powerful no-code extensibility. Rather than hardcoding integrations inside the CMS, we centralize them in workflows that can evolve independently, be reused across clients, and be customized without requiring new CMS releases. For advanced clients, their own n8n instance will allow them to create entirely bespoke automations.

Looking forward, this also provides a natural **bridge with AI-driven scenarios**. By combining CMS events with AI services inside n8n, we can imagine automations such as: automatically generating and sending a newsletter from a batch of new articles, enriching metadata with AI tagging, or drafting social media posts tailored to each channel. With this approach, AI can be leveraged safely and incrementally, without bloating the CMS core.

Why this makes sense

- **Keeps CMS lean**: no hardcoded connectors inside core.
- **Generic by design**: the same event can power multiple downstream actions.
- **Future-proof**: new integrations or AI-powered flows can be added without changing CMS logic.
- **Client empowerment**: advanced customers can self-serve integrations using their own automation instance.

Planned scope for v1.5

- Emit webhooks for key lifecycle events (publish, update, delete, form submit, media upload).
- Operate a shared **n8n instance** to deliver standard workflows across all clients.
- Allow clients to connect their **own n8n instance** for custom needs.

Candidate n8n flows (illustrative):

- On article publish → auto-post to Facebook, LinkedIn, and Twitter/X.
- On new article batch → generate a **newsletter draft with AI** and push it to the mailing tool.
- On media upload → trigger AI moderation or automatic tagging before publishing.

This is not an optional add-on but a core strategy to avoid connector sprawl.

6 AI: VISION, GOVERNANCE, GUARDRAILS

6.1 AI at the service of our clients

Beyond what we already mentioned in section “3.4 AI-First”, AI could provide a lot of features. Here are some ideas:

Short-term – proven and expected features

- **AI-assisted content editing**: grammar, tone, readability improvements.
- **Automatic SEO optimization**: keyword generation, URL structuring, metadata recommendations.
- **Multilingual support**: high-quality translation of content into multiple languages.
- **Accessibility checking**: detect missing alt text, poor contrasts, heading issues.
- **Image optimization**: auto-resizing while preserving focal points (avoid cropped heads/objects).
- **Content-to-format conversion**: turn a long article into a short news item, newsletter, or social post.
- **Smart media tagging**: auto-generate tags and categories for faster organisation.
- **Draft mode everywhere**: easy preview/testing of changes before publishing.

Medium-term – differentiating features

- **AI-powered image editing**: integrated tool to transform visuals (“brighten this,” “remove background,” “add element”), using models like Nano Banana.
- **Automatic content summaries**: generate executive summaries or TL;DR versions of pages.
- **Interactive content generation**: suggest polls, quizzes, or calls-to-action based on the page topic.
- **Layout generation**: AI builds mini-sites or landing pages from APIs/content with CMS-compliant design.
- **Dynamic CSS transformation**: AI proposes alternative styles (colors, typography, spacing) within brand guidelines.
- **Brand guardrails**: enforce consistency with client style guides (logos, fonts, colors).
- **Content scheduling optimization**: AI recommends best publish times based on audience history.
- **Engagement prediction**: forecast performance of a piece of content before publication.
- **Automated A/B testing**: generate variants of headlines/images, test them, and recommend the winner.

Long-term – forward-looking opportunities

- **Personalization engines**: content and design adapt automatically to user profiles, behaviour, or segments.
- **Predictive recommendations**: AI suggests which content types, layouts, or channels will maximize engagement.
- **Full-page “design co-pilot”**: AI proposes alternative layouts or block structures for improved UX.
- **Adaptive design testing**: simulate experiences for different devices and accessibility profiles (e.g., color-blind, low vision).
- **Heatmap simulation**: predict where users’ attention will go on a page and suggest adjustments.
- **Content gap analysis**: identify missing topics compared to competitors or search trends.
- **Voice-to-content**: generate CMS-ready content directly from recorded speech (events, podcasts, interviews).
- **Compliance & policy guardrails**: detect sensitive or non-compliant content before publishing.

AI-Optimized SEO (Prospective)

By 2026, SEO will increasingly mean optimizing for **AI assistants** rather than just search engines. This shift implies:

- **Structured, semantic content**: machine-readable formats (schema.org, embeddings) so AI models can parse and reuse content.

- **AI-driven optimization**: copilots suggesting headings, summaries, FAQs, and metadata tailored for voice/chat queries.
- **AI-ready publishing**: exposing APIs and vectorized content that can be retrieved by LLMs directly.
- **Trust & compliance signals**: authorship, provenance, and transparency metadata so AI models select our content as reliable.

This is highly prospective, but it shows the CMS must be designed for a world where visibility depends on being **picked up and summarized by AI models**, not only ranked in Google.

To stay focused and reduce delivery risk, v1.0 will limit AI features to editorial assist, translations, alt-text, and basic SEO. More ambitious capabilities such as layout generation and personalization will be explored only once we have clear KPIs and proven ROI.

6.2 Our AI Posture

As we integrate AI into the CMS (and more globally our products), we need a shared posture that balances innovation with compliance, governance, and operational control. The following points are not final rules but **considerations and open questions** to be discussed in the following months.

6.2.1 Principles

- AI is an **assistant, not a replacement**: humans remain in control of publishing and validation.
- **Inference vs Training** must be separated:
    - **Inference (using existing models)** is the default mode of operation for the CMS. It is part of the platform baseline and not designed to be switched off, as maintaining a "non-AI" variant would create unnecessary complexity and cost. However, for clients with strict confidentiality requirements, we can offer (at a cost) the option to plug in their **own inference model** instead of the shared default.
    - **Training/Fine-tuning** on client data will be **opt-in only**, requiring explicit contractual consent. Here as well, clients may choose to provide their **own trained model** if they prefer full control over confidentiality and outputs.
- **No hidden training**: by default, we will not use client data to improve global models unless the client explicitly agrees.
- **EU-first compliance**: GDPR and EU AI Act alignment will guide all choices.

6.2.2 Data & Processing Flows

- We need to map and document **what data goes where**:
    - Editor text, metadata, and images processed by inference models.
    - Analytics or personalization signals only processed when explicitly enabled.
- **Inference location**: preference is EU-hosted services. If using external APIs, DPAs and SCCs must be signed.
- **Data logging**: open question whether prompts/outputs should be logged (for debugging and finops). If yes, define retention, anonymization, and access controls.

6.2.3 Consent & Governance

- **Training/fine-tuning** requires explicit client opt-in.
- **Inference** is considered part of the platform baseline, but certain modules (e.g., personalization, predictive analytics) may require admin consent toggles.
- Proposal: create a **client-facing AI settings dashboard** to manage which features are enabled and under which consent.

6.2.4 Compliance & Risk

- **PII handling**: we need safeguards, so member data does not leak into AI prompts or logs.
- **Auditability**: clients may expect a record of what AI has changed or suggested.
- **EU AI Act**: most features will likely be "limited risk," but we should keep a **lightweight risk register** documenting each use case, with owner and mitigation.

6.2.5 FinOps & Cost Control

- AI usage must be **metered**: track inference calls per feature, per site, per client.
- This enables **internal cost management** and **client billing transparency**.
- Open question: do we bill AI usage as a pass-through (usage-based) or as bundled "tiers" of functionality?

6.2.6 Testing & Quality Framework

- To prevent **model drift** and ensure quality over time, we need a repeatable testing approach:
  - Define a set of **reference inputs** with known "good outputs."
  - Run new or alternative models against these inputs and **score outputs** against the reference set.
  - Use a mix of **automated metrics** (BLEU/ROUGE for text, structural similarity for images) and **cross-model grading** (multiple models score the candidate output).
- This framework should be part of **release QA**, so we can swap or upgrade models with confidence.

6.2.7 ROI & Phasing

- New AI features should be gated by **adoption and measurable ROI** (e.g., time-to-publish, SEO lift, editor satisfaction).
- Expansion to advanced features (personalization, predictive engagement) will only happen once early features show traction and compliance risks are resolved.

6.3 Future Outlook: AI-driven Site Generation

Several actors in the industry are already experimenting with AI-driven website creation, where an entire landing page or microsite can be drafted from a text prompt or a visual reference. While these tools are still limited to simple use cases, we need to acknowledge the direction of travel: AI-assisted site generation will become a mainstream expectation within the next few years.

To prepare for this horizon, our CMS must be designed with the right foundations. In practice this means:

- A **contract-first API** with stable schemas for Pages, Articles, Media, Taxonomies, and Blocks, so AI tools have a predictable way to create and arrange content.
- A **declarative block system** rather than ad-hoc layouts, so AI can reliably map sections to components.
- **Design tokens and themes** that enforce brand consistency, accessibility, and performance standards regardless of how layouts are generated.
- **Governance hooks** (SEO rules, accessibility checks, performance budgets) embedded into the publishing workflow, ensuring AI-generated outputs remain compliant.

If we achieve this, we can unlock scenarios such as:

- Drafting a new page or microsite from a natural language prompt ("Build me a donor campaign site with a hero, a 3-step explainer, and a form").

- Cloning the structure of a reference design (e.g. screenshot, Figma file, or competitor's site) directly into block layouts.
- Generating alternative layouts or localized variants automatically, while staying within approved design tokens.

This is **not part of v1.0 or v2 scope**, and human validation will remain mandatory for quality and compliance. However, by thinking about these requirements now, we ensure our system will not block such innovations in one, two, or three years' time. In short: we want to build a CMS that is **ready for AI-First experiences when the market and our clients are**.

Roadmap position: AI-driven site generation is an **exploration beyond v3,** to be revisited once the CMS foundations (APIs, blocks, design tokens, governance) are mature and adoption KPIs confirm client demand.

7 RISKS

7.1 Dependency on Unlayer (Editor)

Unlayer is positioned as the core editor of our CMS, powering page and article creation as well as block-based design. This is a deliberate choice: it accelerates delivery and allows us to benefit from Unlayer's ongoing R&D (including AI-assisted editing). However, relying heavily on a third-party component also creates a set of strategic risks that need to be acknowledged.

Pros

- **Time-to-market**: Leveraging Unlayer allows us to deliver a mature drag-and-drop editor quickly, without rebuilding the full editing stack from scratch.
- **Feature breadth**: Unlayer continuously adds features (AI text, email editing, block marketplace) that we can reuse, accelerating innovation.
- **Consistency**: Using one editor across websites, emails, and landing pages unifies the user experience and reduces training/support overhead.

Cons / Risks

- **Vendor lock-in**: Our CMS depends on Unlayer's roadmap, licensing, and business stability. If Unlayer changes direction, pricing, or terms, our flexibility is limited.
- **Limited control**: Some advanced features (e.g., custom schema-driven fields, deep AI integration) may not align with Unlayer's roadmap.
- **Integration cost**: Tight coupling makes it harder to swap editors later, should we need to.

Mitigation (future option)

- Keep in mind the concept of **adapter boundaries**: an internal contract/interface between the CMS and the editor.
- For now, we deliberately **do not build this abstraction** (to avoid slowing delivery), but it remains a potential safeguard if vendor dependency becomes critical.
- Establish regular checkpoints to evaluate:
    - Cost vs value delivered by Unlayer.
    - Ability to meet client-specific feature requests.
    - Emerging alternatives (open-source or in-house).

7.2 Lack of clear KPIs to guide roadmap decisions

Without clear KPIs, we risk prioritizing features that do not meaningfully improve client outcomes or editor satisfaction. To guide the roadmap, the product team will need to define a small set of **north-star metrics**.

Candidate areas include:

- **Editorial productivity**: time-to-publish, editor satisfaction (NPS), support ticket volume.
- **Website performance & SEO**: Lighthouse / Core Web Vitals scores, indexing coverage.
- **Business outcomes**: donation conversion rates, engagement with content.

We do not yet have these metrics in the current CMS, but the new platform gives us an opportunity to introduce them. The exact KPIs and targets will be defined and tracked by the product team.

Mitigation: ensure telemetry hooks (basic logging of editor actions, publishing times, and site performance metrics) are included in v1.0 so KPIs can be measured later without costly rework.

Telemetry hooks to include in v1

Area	Hook (example)	Purpose
Editorial actions	Log timestamps for "create page/article" and "publish"	Measure time-to-publish, editor workflows
Editor feedback	Simple in-app survey after publish (1–5 stars)	Early signal for editor satisfaction (proxy for NPS)
Site performance	Capture TTFB, LCP, CLS from real users (RUM)	Track Core Web Vitals on production sites
Publishing health	Count zero-result or failed publish attempts	Detect friction points, reduce support tickets

Product must commit to defining 3–4 north-star KPIs in parallel with v1.0 delivery.

7.3 Migration of existing clients

Migrating existing clients to the new CMS is part of a broader transition toward the One-Platform. This typically involves not just the CMS but also CRM migration (from Eudonet CRM or Netanswer) and, for GiveXpert, moving from the stand-alone version to GiveXpert V2 fully integrated into the platform.

There will be no "magic wand" to perform these migrations automatically and perfectly. However, we can significantly reduce the manual workload through a combination of:

- **Conversion scripts**: when the source data is standardized (e.g., a configuration form on GiveXpert, or homepage setup on Netanswer).
- **AI-assisted transformation**: converting free-HTML content into Unlayer blocks/pages with minimal human intervention.
- **Client participation**: clients will still need to validate and adjust migrated content, but our role is to make the process efficient enough to be worth their time.

First version (baseline)

- **Self-service migration module**: clients initiate the migration themselves, essential for scaling across ~400 clients.
- **Supported sources**: GiveXpert pages and Netanswer CMS content.
- **Automatic redirects**: 1:1 redirects generated for migrated URLs to preserve SEO and traffic continuity.
- **AI as a standard step**: AI automatically converts free-HTML into Unlayer pages to cover the bulk of non-standard cases.
- **No rollback**: since the old and new platforms remain distinct, rollback is not necessary; clients can continue using the old system until migration is complete.

As part of the migration process, the platform will provide a **pre-flight diff report** before import. This report will highlight the number of pages, broken links, and the planned redirect map, giving clients visibility on what will change. Even if deeper automated validation only comes in later versions, this lightweight check ensures a safer, more predictable migration from day one.

A possible example:

- Pages detected: 154 (of which 12 unpublished)
- Broken links identified: 8 (list provided in CSV)
- Redirects to be created: 27 (legacy → new URL map)

Second version (if deemed necessary)

- Support for richer validation: content diffing, broken link detection, or URL parity reports.

- Broader source coverage (other modules beyond CMS pages).
- Pre-migration analysis reports to help clients scope the effort before starting.

This approach aims for **~95% quality at a fraction of the manual effort**, accepting that migration will never be fully deterministic but can be made manageable and repeatable.

7.4 Other future risks

Other potential risks such as long-term search quality drift or the need for stricter plugin isolation have been identified in the document but are not included in this register at this stage. They are considered lower priority or further in the future. These topics will be revisited once adoption scales and the ecosystem grow beyond the initial scope.

8 IMPLEMENTATION NOTES & OPEN TOPICS

8.1 Pricing Model

We previously raised the topic of **AI usage metering** in the AI section, but the same concern applies more broadly. Our CMS will consume resources whose costs can vary significantly between clients and over time. To ensure scalability and sustainability, we need to instrument the platform to measure not only AI usage but also other key drivers of infrastructure cost.

At a minimum, we should expose usage metrics for:

- **Storage** (media library size, number of files, total GB stored)
- **Bandwidth/CDN traffic** (requests, volume served, cache hit ratio)
- **Image transformations** (resizing, optimization, format conversion)

These metrics are not meant to enforce strict limits at launch, but to provide **visibility** into consumption patterns. This will allow us to design simple **plan tiers** (e.g., basic vs premium packages), define **guardrails** to prevent abuse, and provide **transparent reporting** to clients.

From an implementation perspective, this means adding lightweight metering hooks at the service level and exposing them through internal dashboards or APIs. Beyond pricing, these same hooks can serve as **operational KPIs** (e.g., query success rate, cache efficiency, storage growth), ensuring that product, support, and engineering teams have the data needed to monitor health and continuously improve the service.

8.2 APIs

A first-class API is at the heart of our headless CMS. It must serve both as the backbone for our own front-ends and as a reliable integration point for other tools in the ecosystem. This means providing a clear and stable contract that defines how content and related objects can be created, read, updated, and deleted.

At a product level, the API needs to cover the core entities from the start — **Pages, Articles, Taxonomies, and Media** — and expose them consistently. We must define how listings work (sorting, filtering, pagination), what error responses look like, and how changes are versioned and communicated over time. This consistency is what will let teams build safely on top of the CMS without risk of silent breaking changes.

The API should also reflect our governance goals: predictable query performance, reasonable pagination limits, and clear rules for deprecation when objects evolve. As the solution grows, the API will become the primary surface for extensibility and integration, so it needs to be treated as a product in its own right, with documentation, examples, and usage guidelines.

For the initial release, we will provide a **REST API** that is easy to consume and widely supported. In the future, we may expand to **GraphQL** if and when we see strong use cases for more flexible querying.

Non-negotiables for the API design:

- Pagination must be consistent and limited to avoid performance issues.
- Errors must follow a predictable format with clear codes and messages.
- Versioning policy must be explicit, with deprecation notices and migration guidance.
- Sorting and filtering semantics must be standardized across all entities.
- Documentation must be maintained alongside the API itself (living contract).

Finally, these specifications will need to be developed in **close coordination with the One-Platform teams** to ensure coherence and alignment across the broader R&D landscape.

8.3 Technical Foundations & Work Organisation

To deliver the CMS efficiently and coherently, we will treat **front-end and back-end as two distinct projects** with a shared API contract and living documentation. This separation allows teams to move in parallel while maintaining alignment through regular integration checkpoints. Documentation, including API definitions, plugin manifests, and block usage guidelines, will be treated as a product in itself and kept up to date as part of the delivery process.

A **library of reusable blocks** will be developed and governed centrally to ensure design consistency and reduce duplication. Each block should be reviewed for compliance with design tokens, accessibility, and performance budgets before being made available across projects. Collaboration with the One-Platform teams will be essential to keep APIs, theming, and design guidelines coherent across the wider R&D ecosystem.

We also set several **foundational technical requirements** for work organisation:

- **Time management:** all stored times must be in GMT; the front-office should detect user time zone automatically, and back-office users should be able to select their preferred time zone.
- **Media storage:** all media and images must be stored in an **S3-compatible storage** system to ensure scalability.
- **Image processing:** support **on-the-fly resizing via URL parameters**, with transformed images cached temporarily and cleared periodically from S3 to save space.
- **Security IDs:** no consecutive numeric IDs may be exposed in APIs visible to end users, to prevent IDOR vulnerabilities.
- **Testing:** all features must comply with the **standard testing requirements** defined in our Development Approach Plan for this year.
- **Delivery pipeline:** the CMS must follow a **continuous delivery** approach to allow rapid iteration and safe deployments.

8.4 MVP Scope for v1.0

Inclusions (must be delivered in v1.0)

- **Pages**: SSR, draft/versioning, i18n slugs, canonical URLs, and redirect management.
- **Articles**: flat categories + tags, with auto-generated category landing pages.
- **Themes & variations**: based on design tokens; variations selectable; custom CSS disabled by default (unlockable as advanced).
- **Search v1**: index Pages and Articles; simple filters; capped pagination.
- **Plugin manifest v1**: route declaration, cacheability/SSR flags, 1 route-owning app integrated.
- **Performance budgets**: HTML/JS caps enforced; EU-based CDN configured.
- **AI assist**: editorial help (rewrite/summarize), basic SEO (title/meta), translations, alt-text generation.
- **Analytics & compliance**: Matomo integration, CMP, lightweight events spec for consistent plugin telemetry.
- **Migration tooling**: pre-flight diff report (page counts, broken links, redirect map), self-service redirects.
- **Delivery pipeline**: continuous delivery for the CMS; all features documented as part of delivery.

Exclusions (not in v1.0)

- Editorial workflows beyond Draft/Published (scheduling, audits → v1.1+).
- Search governance (synonyms, analyzers, no-results dashboard → v1.1).
- Media metadata/tagging and DAM collections (→ v1.1/v2).
- Integration & automation via n8n/webhooks (planned for v1.5).
- AI layout generation, personalization, advanced analytics (v2+).
- Plugin marketplace or third-party plugins (future exploration).

Constraints

- **Performance**: HTML ≤ ~200 KB, JS ≤ ~300 KB, initial TTFB ≤ 200ms at CDN edge.
- **Accessibility**: RGAA compliance required; exact level (AA/AAA) to be confirmed at group level.
- **Security**: only trusted internal plugins in v1.0; no unvetted third-party code.
- **Testing**: all features must comply with group's Development Approach Plan for automated testing.
- **Time handling**: all stored in GMT; FO auto-detects timezone; BO users can select.
- **Media**: stored in S3-compatible storage with on-the-fly resize by URL parameters.
- **IDs**: no numeric consecutive IDs exposed via APIs (IDOR protection).

Dependencies

- API contracts and schemas must be coordinated with **One-Platform teams**.
- Design system alignment required with **One-Platform design guidelines**.