ChatGPT

Documentation & Knowledge Base System Design

1. Docs Architecture and Governance

A **docs-as-code** approach will be used, keeping all documentation under a `/docs` directory in the monorepo alongside the code. This ensures docs version in lockstep with code changes and allows collaboration via Git (pull requests, code review, branching) [1]. The documentation site will be structured into logical sections reflecting the multi-module architecture: - **Module Sections:** Each major module (Frontend UI, AI/ML Engine, Forecasting Model, Budget Planner, Revenue Tracker, Infra/DevOps, etc.) gets its own subsection with module-specific docs (design overviews, how-to guides, API docs). - **Global Sections:** Cross-cutting documentation like Project Overview, Architecture & Data Models, Glossary, and Compliance lives in top-level sections. Also include sections for **Decision Log** (ADRs), **Runbooks/Playbooks**, and **API Reference** separate from module areas. - **Content Ownership:** Assign clear ownership for docs per area. For example, module leads or "owners" are responsible for updating their module's docs; the AI module agent team maintains AI/ML documentation, etc. A **Docs Steward** role (could be a rotating developer) oversees the overall structure and standards compliance. All docs changes go through PRs that are reviewed for accuracy and clarity (at least one technical reviewer and optionally a docs steward). - **Versioning Strategy:** Use Git branching and tags to manage doc versions. For now, maintain a single **"Latest"** version (matching the bleeding-edge in `main`). As the product matures, introduce versioned documentation for releases (e.g. v1, v2) using tools like Docusaurus versioning or MkDocs versioning, so users can access docs relevant to their app version [2]. Each version's docs will be built and preserved on the site. In Git, you might keep version branches or a `/versioned_docs` folder if using Docusaurus. - **Review & Maintenance Cadence:** Institute a regular cadence for docs maintenance. For example, every sprint's **Done** criteria includes updating relevant docs, and schedule a monthly or quarterly **docs audit** where stale content is flagged. Use PR templates to prompt authors with "Have you updated or reviewed documentation for this change-set?" to integrate docs into the development workflow. Major architectural changes (like new features or module refactors) should not be considered complete until documentation (design docs, API changes, user guides) is updated – enforced via the PR review process. Also plan periodic **content reviews** (e.g. each quarter) where owners re-read their sections for accuracy and clarity, ensuring the knowledge base stays current.

Docs Layout: Organize the `/docs` directory into subfolders by content type and module for navigability:

```
/docs
   /overview/        (project intro, vision, architecture)
   /modules/
       /frontend/
       /forecasting/
       /budget/
       ... (one per module, containing that module's design, setup, etc.)
   /adr/             (architecture decision records)
   /runbooks/        (operational runbooks)
```

```
/playbooks/       (persona scenario playbooks)
/api/             (API specifications or references)
/glossary.md
/index.md         (landing page for docs)
```

This hierarchy will reflect the taxonomy in the published site's navigation (mirroring Project → Module structure). A logical taxonomy makes it easy to find content by module or topic [3]. The sidebar or menu will group pages by module and doc type for quick scanning. Each page will have a consistent header structure (using H1 for title, H2 for major sections, etc.) to enforce uniformity.

Roles & Workflow: All documentation is handled through Git, so the same **CI pipeline** that tests code will also build and validate docs. This enables **live previews** for each docs update (more on CI in section 6 and publishing in section 9). Contributors write in Markdown (or MDX for advanced content) with YAML front matter for metadata. There will be **CODEOWNERS** for the `/docs` folder (e.g. the docs steward or senior engineer) to ensure at least one knowledgeable reviewer on every docs PR. By treating docs as code, we leverage version control for accountability (every change logged) and can even rollback or diff documentation changes easily [1]. The entire team is encouraged to contribute improvements (typos, clarifications), fostering a culture that documentation is a first-class artifact of development.

Review Cadence & Governance: To avoid docs lagging behind code, adopt a **"no merge without docs"** policy for features: if a change-set implements a new user-facing feature or module, the PR must include docs changes or an ADR/decision record explaining why docs aren't needed (rare). CI can enforce some of this (e.g. failing if required doc files are missing), but primarily this is a cultural/policy rule. Additionally, schedule **doc review meetings** each sprint or milestone where recent docs are quickly checked for quality. An **owner matrix** (mapping sections to owners) is maintained so everyone knows who to consult for a particular document.

Finally, maintain **transparency** and historical traceability: Keep old decisions and older version docs accessible (perhaps in an archive folder or via the versioned docs system) so nothing is lost. This ensures new contributors can trace how the system evolved. The goal is a well-structured, up-to-date knowledge base that grows with the project and is easy to navigate for both humans and AI agents.

2. Content Model & Templates

All documentation content will follow a structured model with **YAML front-matter** at the top of each Markdown file to consistently categorize and label documents [3]. The front-matter will include common fields (like `title`, `type`, `module`, `author`, `date`, `status`) and type-specific fields. Standardizing this metadata allows automated checks and easier indexing for search (both UI search and AI retrieval). We will define **templates** for each major document type to ensure consistency:

- **Product Requirement Docs (PRDs):** Used for new features or modules (e.g. a PRD for a *"Tax Intelligence"* module or *"Calendar Heatmap"* feature). **Front-matter** fields might include: `type: PRD`, `feature_name`, `module`, `author/product_owner`, `stakeholders`, `created_date`, `status` (Draft, In Review, Approved), and `target_release`. The body template will include sections like **Background**, **Objectives/Goals**, **User Stories & Use Cases**, **Scope (In/Out)**,

Requirements Detail (functional and non-functional), **Acceptance Criteria** and **Impact Analysis**. For example, a PRD might start like:

```
---
type: PRD
feature: "Tax Intelligence Module"
module: "Revenue"
author: "Product Manager"
stakeholders: ["Finance Team", "Dev Lead"]
status: Draft
created: 2025-08-10
target_release: "2025-Q4"
---
```

And the content would then provide context (why a Tax Intelligence feature is needed), specific requirements (e.g. "The system shall auto-calculate quarterly tax obligations based on categorized transactions"), assumptions, and criteria for completion. This ensures new features are well-defined in docs before or during development.

- **Architecture Decision Records (ADRs):** ADRs capture significant technical decisions in a lightweight format [4]. They will serve as our **Decision Log** (section 3 details the system). Each ADR is a single Markdown file (stored in `/docs/adr/`) that records one decision along with its context and reasoning. **Front-matter** for ADRs might include: `type: ADR`, `id` (a sequential number or unique ID like ADR-0001), `title` (short summary of decision), `status` (Proposed, Accepted, Superseded), `date`, `deciders` (people who approved), and optionally `tags` (for relating to modules or topics). The body follows a template: **Context** (the problem or issue being addressed, and any forces or requirements), **Decision** (the choice made, e.g. "Adopt Postgres for the main database"), **Consequences** (implications of the decision, positive/negative, and any follow-ups) [5]. We will use a format consistent with Michael Nygard's ADR template for clarity [5]. For example:

```
---
type: ADR
id: "ADR-0005"
title: "Use Docusaurus for Documentation Site"
status: Accepted
date: 2025-09-26
deciders: ["CTO", "Dev Lead"]
tags: ["Docs", "Tools"]
---
```

Context: We need a static site generator for our docs that supports Markdown, versioning, and integration with our React/Next.js toolchain…

Decision: Chosen to use **Docusaurus** as the docs framework, because … (rationale) …

Consequences: We will set up a Docusaurus site under `/docs`. All docs will be written in MDX. This decision means contributors should be familiar with React-based docs, but it provides a robust theming and versioning system. Alternatives considered were MkDocs and ReadTheDocs…

ADR files will be named with an index and slug (e.g. `ADR-0005-use-docusaurus.md`) and stored chronologically. This yields a human-readable log of decisions. The front matter helps index them by status and module. ADRs provide a time-ordered, **immutable** record of why certain key choices were made, so future maintainers or AI agents can understand the system's evolution [4].

- **Runbooks:** Runbooks are step-by-step operational guides for specific tasks or incidents in the system (often maintained by DevOps or SRE roles). They target **technical processes** and detail the finer steps to, say, restart a service, recover from an outage, or perform routine maintenance [6]. Each runbook will have front-matter like: `type: Runbook`, `id` (e.g. RB-001), `title` (the task name), `system` or `module` (e.g. "Forecasting Service" or "Gmail Hub Ingestion Pipeline"), `owner/team` (responsible team), and `last_reviewed` date (to ensure they are periodically verified). The content format includes: **Overview/Purpose** (what this runbook is for, e.g. "How to restore the Forecasting Model service from a failure"), **Scope/Audience** (who should use it – e.g. "DevOps engineer on-call"), **Prerequisites** (permissions or tools needed), **Procedure** (the ordered steps to execute), **Verification** (how to confirm the step succeeded), **Troubleshooting** (common issues and fixes), and **Rollback** if applicable. Steps should be clearly enumerated (using ordered lists or bullet points) and written in imperative language. For example, a runbook's procedure section might say:
- **Step 1:** Navigate to the serverless function dashboard for the **ForecastingModel** service.
- **Step 2:** If the service is not running, click "Deploy Latest" to restart it. *Expected Result:* status changes to Running.
- **Step 3:** Verify recent logs for errors. If errors like "DB connection failed" appear, follow the Database Reconnection sub-steps…

By structuring runbooks with clear titles, scopes, and numbered steps, we make them not only human-friendly but also **AI-accessible** – an AI agent can easily parse the steps or even execute them if integrated with automation. We will ensure runbooks are written consistently (consider using a **runbook linter** to check for missing sections or overly long steps). Runbooks will be stored in `/docs/runbooks/` and perhaps named by task (e.g. `runbook-restart-forecasting-model.md`).

- **Playbooks:** Playbooks provide higher-level guidance for **broader scenarios or processes** that might involve multiple runbooks or decision points [6]. They are often persona-based – for example, an **"Onboarding Playbook for a New Finance Team Member"** or an **"Incident Response Playbook for Data Breach"**. They differ from runbooks in scope: a playbook might orchestrate which runbook to use when, or describe a workflow with decision branches. Front-matter: `type: Playbook`, `title`, `applicable_persona` (e.g. "Customer Support Agent", "Finance Officer", "SysAdmin"), and possibly `related_runbooks` (list of runbook IDs referenced). The playbook content is usually narrative with conditional steps or checklists. For instance, a **"Financial Close Playbook"** for the *Tax Intelligence* module could have sections: **Preparation** (data needed before close), **Steps** (which might reference specific runbooks like "Runbook: Generate Tax Report"), and **Post-action Review**. We will template playbooks to include an **Overview** of the scenario, **Roles Involved**, **Trigger** (when to run this playbook, e.g. "end of quarter closing procedure"), **Procedure** (with substeps or references to runbooks: e.g. "If the tax calculation fails, execute RB-010 'Recompute Tax Obligations'"), and **Outputs/Reports** generated. Playbooks thus act as an umbrella for multiple runbooks, providing a

broader context or strategy. By documenting them, we ensure even complex operations (spanning multiple modules or teams) are standardized and can be followed by anyone or even guided by an AI assistant.

- **Deep Research Reports & Design Docs:** These are in-depth technical investigation documents – exactly the kind of document we are producing right now. They compile analysis, comparisons, and recommendations on complex topics (for example, a **Blueprint for Forecasting Financial Time Series** or a **Deep Dive on Multi-currency Accounting**). They often include context, detailed findings from research, and proposed designs or decisions. We will treat these as a content type for long-form technical documentation. **Front-matter**: `type: ResearchReport` (or `DeepResearch`), `title`, `author`, `created`, and possibly `related_module` or `initiative`. These docs may also list `sources` or have an appended references section, since they often cite external info. The body is usually structured in a narrative form with headings, but we can have a loose template: **Introduction** (problem statement or questions), **Findings** (organized by subtopic), **Recommendations/Conclusions**, and **References**. These documents benefit from being in the knowledge base because they preserve context and rationale for complex features. For example, the project already has comprehensive guides (like the unified engine design guide covering data modeling, logic, trade-offs, and diagrams [7]) – those would be converted into this format under `/docs/research/` or within relevant module sections. By including them, the documentation system serves as the historical memory and design archive of the project.

- **Diagrams & Architecture Graphics:** Architectural diagrams, flowcharts, and other visuals will be stored and managed as part of docs-as-code. We propose using text-based diagram tools (like Mermaid or PlantUML via Markdown) so that diagrams are version-controlled and can be reviewed in PRs. For example, an architecture overview might include a Mermaid diagram code block in the Markdown, which the site can render. If binary images are necessary (PNG/SVG), we store them under `/docs/assets/` or alongside the doc, with descriptive file names. We will also maintain a simple front-matter for standalone diagram pages if needed (e.g. `type: Diagram`, `title`, `description`, and maybe `related_docs`). But generally, diagrams accompany other docs (like included in design docs or runbooks). We will ensure every diagram has a caption or description in the text so that an AI agent (which might not parse the image content) still gets context. We will also incorporate an **automated diagram build** step in CI: e.g. if using PlantUML, the CI can generate/verify the UML diagrams from source to catch syntax errors or update images. Diagram source files (for complex diagrams drawn in an external tool) should also be stored (e.g. draw.io .drawio files) for easy updates by any developer.

- **API Contracts & Reference Docs:** The system's APIs (internal module APIs or external endpoints) will be documented either via integration of an OpenAPI spec or via markdown templates. For instance, if the *Gmail Hub* module exposes an API or if the front-end calls backend endpoints, we maintain an **OpenAPI (Swagger) YAML/JSON** file describing them. That file can be stored under `/docs/api/` (or generated from code annotations). We'll generate human-friendly reference pages from it – possibly using a tool like Redocly or Docusaurus plugin to render OpenAPI, or simply manually writing endpoint docs. Template for API endpoint docs (if done manually) could include: **Endpoint** (method and URL), **Description**, **Request format** (parameters, body schema), **Response format**, **Examples**, and **Error codes**. We will ensure these stay in sync with the code by either automation (preferred, generating from code or tests) or by mandating that API changes require updating the OpenAPI file plus docs (enforced in code review and CI schema checks; see Doc Quality

Gates). Having a clear API contract in the knowledge base allows internal developers and future integrations (and AI agents interfacing with the system) to know how modules communicate. This content type may have front-matter like `type: APIReference`, `module`, `endpoint` for indexing, but often these will be collated in one section rather than separate for every endpoint.

All templates will be stored as examples (for instance, provide Markdown template files in a `/docs/templates` directory or documented in a contributor guide). New documents should be created by copying the appropriate template and filling in the front matter and sections. This ensures consistency across the documentation set.

Summary of Front-Matter Schema: To formalize, we will publish a JSON Schema or at least a documented schema for the front-matter of each type. For example, a simplified schema snippet for an ADR might look like:

```
{
  "type": "object",
  "required": ["type", "id", "title", "status", "date"],
  "properties": {
    "type": { "const": "ADR" },
    "id": { "type": "string", "pattern": "^ADR-[0-9]{4}$" },
    "title": { "type": "string" },
    "status": { "enum": ["Proposed", "Accepted", "Rejected", "Superseded"] },
    "date": { "type": "string", "format": "date" },
    "deciders": { "type": "array", "items": { "type": "string" } },
    "tags": { "type": "array", "items": { "type": "string" } }
  }
}
```

Similar schemas will exist for PRD, Runbook, etc., with their respective fields. These schemas can be used in CI to **validate** that docs have correct front-matter (see Doc Quality Gates). By enforcing structured metadata, our docs become **machine-readable** to a large extent, which is invaluable for search, navigation, and AI agent consumption.

3. Decision Log System

To track and index all important decisions made during the project, we will implement a robust **Decision Log** centered around Architecture Decision Records (ADRs). The **Decision Log** will be a living collection of ADR documents that record why and how key choices were made, forming an easily navigable history of the project's architectural evolution [4].

Format and Storage: As described above, each decision is an individual ADR markdown file under `/docs/adr/`. We will adopt a naming convention like `ADR-0001-descriptive-title.md` to ensure alphabetical sorting also sorts by chronological order. Each file contains the decision details in the ADR template format. The ADR front-matter includes a `status` field so we can mark decisions as **Accepted (current)**, **Superseded** (replaced by a later ADR), **Proposed** (still under review), or **Rejected** (decided

against an option). ADRs are immutable once Accepted – if a decision changes, a new ADR is written to supersede the old one, linking to it.

Indexing & Navigation: We will maintain an **ADR Index** page (or sidebar category) that lists all ADRs in reverse chronological order (newest first) or grouped by topic. This could be a manually maintained `index.md` in the `/docs/adr/` folder that contains a list (bulleted or table) of all decisions, e.g.:

- ADR-0001: Use Nx Monorepo Tooling – *Accepted 2025-07-01*
- ADR-0002: Database Choice (PostgreSQL vs Firebase) – *Accepted 2025-07-10*
- ADR-0003: Forecasting Model Approach (classical vs ML) – *Superseded by ADR-0008*
- ADR-0004: Module Communication (Event Bus) – *Accepted 2025-08-01*

This index page is the quick reference for anyone to see what decisions have been made and their status. We will update it with each new ADR (this could be automated with a script that scans the ADR folder and updates the index – adr-tools or similar can generate an index, but doing it manually is fine for a small team). The Decision Log index helps **link related decisions** (e.g. show an ADR that supersedes another) and provides a one-stop overview.

Each ADR document will also explicitly link to related ADRs (using hyperlinks) if applicable – for instance, ADR-0008 might have a note "*This ADR supersedes ADR-0003*". This cross-linking, combined with the index, ensures **traceability** of decisions over time.

Lifecycle and Maintenance: The typical lifecycle: - A significant issue or choice arises (perhaps identified in a planning meeting or as part of a WBS Job). - A draft ADR is created (status = Proposed) describing the options and recommendation. It may be discussed in PR or design reviews. We encourage writing the ADR in early stages so the discussion is recorded. - Once the team decides, the ADR is updated to status = Accepted (or Rejected if we decided not to do something). - If down the line a decision is revisited, a new ADR is written. The old one gets status = Superseded and references the new one. - All ADRs remain in the log. Even rejected ideas are kept (with status Rejected) because it's useful to know "we considered X and decided against it at the time and why".

By following this process, the Decision Log acts as the project's memory. New developers (or AI assistants reviewing the docs) can read through to understand why things are the way they are, avoiding repeating past discussions. This is especially important in a distributed or evolving team where not everyone has the historical context in their head [8] [9].

Linkage to WBS and PRs: Each ADR will be linked to the Work Breakdown Structure (WBS) and implementation artifacts to connect decisions to execution: - We will reference relevant **WBS IDs** in ADRs. For example, if the WBS has a Job for "Decide on Forecasting Algorithm" that resulted in ADR-0003, the ADR can mention "WBS: Project-Forecast-Job5". Conversely, the WBS entry or ticket can link to the ADR "Decision recorded in ADR-0003". This creates a two-way trace: from a planning item to the decision doc. - **Pull Requests** that implement or follow up on an ADR will reference the ADR in their description (e.g. "Implements ADR-0001" or "See ADR-0002 for context"). This is facilitated by training team members to mention ADR IDs in commit messages or PR templates. We could even automate checking that if an ADR is referenced, it is in Accepted state, etc., but mainly it's for human tracking. - In the ADR front-matter or footer, we may include a **Related PRs** field or a list of PR links once the decision is enacted. For instance,

ADR-0001 (choosing a tech stack) might list the PRs that did the initial setup. This is optional but can be helpful for detailed auditing.

Decision Log Indexing Rules: We treat the ADR folder as **append-only chronological log** [4] . To preserve integrity, consider using an **ADR numbering tool** or simple convention to not re-use numbers. The index page is updated whenever a new ADR is merged. The sidebar in the docs site will list ADRs in numeric order by default (Docusaurus can auto-generate sidebar from files). We might prepend the ADR number to titles for clarity in the UI (e.g. "0005 – Use Docusaurus for Documentation Site").

We will also incorporate decisions that are not purely architecture – e.g. process decisions or even high-level product decisions – into the log for completeness. The ADR concept can extend to any important project decision ("any decision record" beyond just architecture) [10] . For example, a decision to change the **release cadence** or a decision to adopt a **monorepo strategy** can be ADRs too.

By having this Decision Log system in place, we ensure every significant change in direction is documented and approved openly, which is crucial for governance and onboarding. It also makes our knowledge base agent-friendly: an AI can be directed to the ADRs to answer "why did we choose X?" type questions with direct quotes from the decision rationale.

4. Runbooks & Playbooks (Operational Guides)

Runbooks and playbooks form the operational backbone of our knowledge base, enabling both humans and AI agents to perform operational tasks reliably. We will develop a library of runbooks and playbooks covering both routine procedures and incident responses, tailored to the personas who will use them (DevOps engineers, support staff, end-users via AI, etc.). The focus is on clarity, completeness, and accessibility to AI-driven assistance.

Runbook System: Each runbook is a concrete procedure, usually technical. For instance, we'll have runbooks like "**Restarting the Forecasting Model Service**", "**Troubleshooting Gmail Hub Email Ingestion Failures**", "**Regenerating Revenue Reports**", etc. Runbooks will follow the template described in Section 2. To reiterate key parts: - A **clear title** stating the action or problem (e.g. *"Runbook: Gmail Hub Outage Recovery"*). - Metadata (front-matter) denoting who and what it's for (module = Gmail Hub, owner = Infrastructure team, last updated date). - Step-by-step **Instructions** that are actionable and in logical order. Each step will be concise (1-2 sentences), starting with an imperative verb (e.g. "Check the service status on Cloud Run" or "Delete all stuck Pub/Sub messages in the queue `gmail-intake`"). - **Expected results** or verification for steps as needed (possibly in italic or as sub-bullets: "(Expected: response HTTP 200 from health check)"). - **Troubleshooting notes** for steps where something commonly can go wrong ("If the service does not restart, check X logs for Y error"). - **References** or links to related docs or external references (like linking to Google's documentation if needed, or linking to an ADR if the procedure is implementing a specific design).

Runbooks are written assuming some technical knowledge, but we also consider that an **AI Agent** might use them to guide an operator. For example, if our system includes an AI assistant that can answer DevOps questions, the assistant might fetch steps from a runbook to help the engineer resolve an alert. Because of this, **formatting matters**: - We will use **consistent markup** for step numbering and bolding important keywords (e.g. *"Note: ensure the API key is present"*) to make it easy for an AI to parse key info. - Avoid

ambiguous language – steps should be deterministic ("do X then Y") rather than verbose prose. - We might include a **Quick Summary** or **Flowchart** for complex procedures for quick scanning (maybe an ASCII or Mermaid flow diagram).

Each runbook will be indexed by module and topic. We'll create a top-level page listing all runbooks by category (e.g. "Infrastructure Runbooks", "Application Module Runbooks"). Additionally, within each module's docs section, we'll list the runbooks relevant to that module. For example, in the Gmail Hub module docs, link to "Ingestion Pipeline Restart – see Runbook RB-003".

Playbooks for Scenarios: Playbooks handle broader scenarios that involve multiple tasks or decision points. Examples: - **"Full System Recovery Playbook"** – when the whole system experiences an outage, coordinating multiple runbooks (DB restore, services restart, reprocess backlog). - **"New Client Onboarding Playbook"** – steps support staff or AI should follow when a new small-business client starts (like verifying initial data imports, enabling modules like Calendar Heatmap, Gmail integration for them, etc.). - **"Quarter-End Tax Filing Playbook"** – guiding a user or accountant through using the Tax Intelligence module outputs to file taxes, with pointers to relevant help articles or runbooks if something is off.

Playbooks will often be used by **persona-defined agents**. For example, a *"Finance Advisor AI"* persona might rely on an internal playbook to walk a user through reducing expenses: the playbook might instruct to review certain dashboards, or use the Forecast module to simulate budget changes. We create these documents so that even if the person interacting is an AI, it follows a vetted process.

In format, playbooks may be more narrative but will still use **ordered lists and conditional logic**. We might incorporate simple conditional statements like: - "If X happens, do Y (refer to Runbook RB-010). Otherwise, proceed to step 5." This explicit branching helps both humans and AI follow the logic. We could also delineate sections for different tracks in the playbook.

We will ensure **cross-referencing** between playbooks and runbooks: A playbook should reference the specific runbooks for detailed steps. Conversely, a runbook can note which playbooks it is part of (so someone looking at a runbook knows the higher context). This creates a web of operational knowledge.

AI-Accessible Formatting: To further optimize for AI agent use (retrieval and interaction): - Use descriptive headings within runbooks/playbooks, like "Steps to Resolve", "Verification", "Next Actions". If a user asks the AI "How do I verify it's fixed?", the agent can jump to the "Verification" subsection. - Keep individual steps relatively atomic and avoid long paragraphs. This not only aids readability but ensures any snippet retrieved by a vector search is self-contained and useful. - Incorporate **FAQs or Q&A style** info where relevant. For example, at the end of a runbook, we might have a small **FAQ** section ("Q: What if the service still fails after restart? A: Check environment variables..."). This can preempt common follow-up questions an AI or human might have. - Include **metadata tags** in front-matter like `persona: "DevOps"` or `severity: "Critical"` for incidents. This could allow an agent to filter or prioritize answers (e.g. if an incident is critical, the agent might respond with more urgency or escalate to a human).

By developing thorough runbooks and playbooks, we not only empower team members to handle operations consistently but also lay the groundwork for **AI augmentation** of operations. Our AI agent (or "module agents" for each module) can use these as a knowledge source in a RAG pipeline to troubleshoot issues or guide users through processes. For example, if an AI module agent is dedicated to the Forecasting model, it will have access to the Forecasting-related runbooks (like how to retrain the model, how to

interpret forecast results, etc.) and can answer or act accordingly. The runbook content will effectively become the **script** the AI can follow or present to users, ensuring the AI's guidance is accurate and in line with our documented best practices.

5. RAG Pipeline & AI Agent Integration

To enable AI agent interoperability with our knowledge base, we will establish a **Retrieval-Augmented Generation (RAG) pipeline** that indexes the documentation and allows agents to fetch relevant information from it in real-time. This ensures our AI assistants (whether developer-facing or end-user-facing) always have up-to-date, authoritative answers based on the docs. Key components of this system:

Embeddings Pipeline: We will implement an automated process to convert our documentation into vector embeddings for semantic search. Using an embedding model (such as OpenAI's text-embedding-ada-002 or a local model), each document (or document chunk) is transformed into a vector representation. These vectors will be stored in a **Vector Index** (could be a vector database like Pinecone/Weaviate, or even an on-disk index using libraries like FAISS or in-memory if small scale). The pipeline likely runs as part of CI/CD or a periodic job: - On every docs change (merge to main) and optionally on a schedule (e.g. nightly), a script will run to re-index changed documents. We can integrate this into a GitHub Action. - The script will scan the `/docs` content, skip any files marked non-indexable (for compliance reasons, see below), and chunk the content, then call the embedding model to get vectors, and update the index.

Document Chunking Rules: It's important to chunk documentation into semantically coherent pieces for embedding, rather than embedding whole documents or random splits. We will use a hybrid chunking strategy: - **By Sections:** We will try to chunk by logical sections or headings. For example, each top-level section (H2 or H3) of a doc could be one chunk, possibly combined with the title. This preserves context (like an entire ADR's text might be one chunk if short, or a long runbook's each step group as chunks). Markup (like headings) will guide splits [11]. We'll leverage the fact that our docs have structured headings to do content-aware chunking rather than arbitrary fixed size. - **Size Constraints:** To respect token limits and ensure efficient search, keep chunks in a range of about **200 to 500 tokens** (roughly a few paragraphs) [12]. Technical documentation can often benefit from slightly larger chunks (to capture context like an API method with its entire description) [12], so we will aim for the upper end (~400-500 tokens) when the content is highly connected. For example, an ADR might be one chunk ~300 tokens, whereas a runbook with 10 steps might be split into two chunks of 5 steps each if needed. - **Overlap:** Implement a small overlapping window between chunks (around 10-15% overlap, e.g. repeating the last sentence or title in the next chunk) to avoid losing context [13]. For instance, if a section is split across chunks, a sentence or clause from the end of chunk 1 may be prepended to chunk 2. This helps the AI not miss transitional info when reading a single chunk. - **Metadata tagging:** Each chunk, when indexed, will carry metadata: the source document title, type, module, and potentially an internal pointer (like "Budget Module Runbook #2, Steps 6-10"). This metadata can be stored alongside the vector in the index and is crucial for filtering and output (so the agent can cite the source). We will include the document reference or URL in metadata so the agent can refer back (or even present a link to the user if needed). - If using an existing solution like Azure Cognitive Search or similar, we can leverage built-in chunking (like their Text Split skill) which chunks by paragraphs or headings [14]. Otherwise, a custom script with a library like LangChain's text splitter will be used, configured for our needs.

Index and Query Workflow: The vector index will allow semantic queries. Our AI agents (e.g. a chatbot integrated into the app, or an internal CLI agent) will use the user's query or task to retrieve relevant doc

chunks: - An agent query like “How do I handle an overdraft alert in the budget module?” will be embedded and matched against vectors. The metadata might find a chunk from a **Budget module Playbook** about handling negative balances, and a chunk from the **Glossary** defining overdraft. The agent then assembles those to formulate a correct answer or guide. - The system will ensure multiple relevant chunks can be retrieved and passed to the LLM (perhaps 3-5 top results) to give it enough context.

Front-Matter Schema for AI Indexing: We will use the front-matter metadata when indexing to enforce **permissions and context filters**: - A field like `visibility` or `access` in front-matter can mark a document as **internal** or **public**. The embedding pipeline will respect this by either skipping internal docs for a user-facing agent’s index or tagging them so that an agent knows not to reveal that content to end-users. For example, an internal runbook might be indexed only for a DevOps assistant agent, not for the end-user-facing financial advice agent. - **Module and Type tags:** The `module` and `type` from front-matter will be attached as metadata. This allows scoping searches. For instance, if the “module agent” concept is used (where each module has a specialized AI agent), that agent can restrict retrieval to docs where `module = Forecast` (plus any global docs) so it doesn’t get unrelated content. This improves precision and also aligns with permission (e.g. a Forecast module agent wouldn’t fetch confidential info from another module). - **Redaction and Sensitive Info:** If a document or section is marked sensitive (see Compliance section), the pipeline might exclude it or sanitize it. We can use markers in docs like `<!-- sensitive -->` … `<!-- /sensitive -->` to denote content that should not be embedded. Before embedding, the script can strip or replace such sections with a note “[REDACTED]”. This ensures, for example, that secret credentials or personal data aren’t accidentally stored in the vector index. We might also decide not to index certain docs like audit logs or compliance docs if they aren’t needed for Q&A. - Each chunk will also store a reference (like `source_url` or `doc_id` with a stable anchor) so that when the agent answers, it can cite the source if required (like we do here with 【source†】 style, although user-facing might not show it exactly like that, but internal usage could).

Agent Query Permissions: We envision multiple AI agents: - Developer-facing agent (internal, can access all internal docs, ADRs, runbooks). - End-user-facing agent (in-app help assistant, should only use user-facing docs like how-to guides, maybe playbooks that are for end users, and definitely exclude internal runbooks or confidential data). - Module-specific agents (like a “Budget Module Agent” that helps with budget questions, “Forecast Agent”, etc., which have access to their module’s docs and general docs).

The retrieval system will support **filtered search** by agent identity. For example, an internal agent might query the entire index; an end-user agent’s queries will include a filter `visibility:public` and perhaps `type != ADR` to avoid exposing internal decisions. This way, the same documentation repository serves multiple audiences safely.

Re-index Triggers: We will integrate re-indexing into our CI/CD: - On every merge to `main` that affects the `/docs` directory, a GitHub Action will run a job (maybe called “Update KB Index”). This job can run a script or use a service hook to update the vector store. In a simple setup, we might just rebuild the whole index and upload it (since docs size is not huge), or do a smarter diff-based update (detect which files changed and only re-embed those). - Additionally, if using Vercel or another deployment, we can trigger re-index on successful deployment of docs (ensuring the index and published content are in sync). - We will also allow manual re-index (maybe a makefile target or CLI script for maintainers to run if needed). - If the vector index is stored in a remote service (like a managed vector DB), the CI job will have credentials to update it. If it’s stored in-repo (like a JSON with embeddings), the CI can commit the updated index (though that can bloat the repo, so external storage is preferred).

Chunk Storage and Search: The index will store chunk content or an identifier. For certain agent interactions, we might want to return not just the answer but allow the agent to show the relevant document section. So storing a snippet of text with each vector is useful. Many vector DBs allow storing a payload (which could be the text chunk itself, capped at say 2000 characters). We will do that, along with doc title and maybe section header. This way, the agent can directly quote the documentation in its answers (ensuring high accuracy and the same phrasing as the docs).

Example: Suppose a user asks, *"What does 'Calendar Heatmap' mean in this app?"*. The end-user agent will embed the question, search the index. Because we have a **Glossary** entry for "Calendar Heatmap", or a module overview that defines it, that chunk will surface. The agent retrieves the definition (e.g. "Calendar Heatmap – a visualization module showing spending by date in a heatmap calendar format") and can present that answer, possibly with a citation or offer to show more. If the user asks *"How do I enable the Calendar Heatmap?"*, the agent might retrieve a Setup Guide chunk for that module or a relevant PRD section and instruct accordingly.

The RAG pipeline thus turns our documentation into a **queryable knowledge base**. It will be critical to keep it up-to-date, so we'll monitor indexing logs for errors and possibly include a **sanity check**: e.g. run a scheduled query test to ensure the index returns results for known queries (to catch if something went wrong in indexing).

Security and Privacy in Retrieval: (This overlaps with compliance but important here too) We will design the system such that certain data is never exposed by the AI. For example, if an internal doc had actual email addresses or secrets (which ideally it shouldn't due to redaction), even if embedded, the agent should be instructed not to output such strings. Policies like "do not answer with anything from a doc marked confidential" can be enforced at the agent level. We can also implement a **fallback**: if a query hits only confidential info, the agent might respond with a polite refusal or a high-level summary without details.

In summary, the RAG pipeline ensures our AI helpers are always armed with the latest knowledge. This makes the documentation truly "living" – not just read by humans, but actively used by AI to assist in development, operations, and user support, fulfilling the vision of an agent-readable documentation system. Our front-matter schemas, chunking strategy, and CI triggers form the technical basis for this pipeline, while compliance controls (next section) will govern *what* goes into the index.

6. Documentation Quality Gates (CI Enforcement)

To maintain a **high-quality bar** for our documentation, we will enforce a series of **"Doc Quality Gates"** in the CI pipeline. These are automated checks that run on each pull request and/or on the main branch to catch common issues and enforce standards before docs are published. Similar to how code quality gates prevent substandard code from deploying, doc quality gates will prevent broken or non-compliant docs from merging [15]. Key quality gates include:

- **Broken Link Checker:** All internal and external links in the markdown should be valid. We'll use a link checking tool (for example, a GitHub Action or npm package like `markdown-link-check` or Docusaurus's built-in link checker) to scan each doc for 404s. If any link is broken (e.g. pointing to a removed page or a typo in URL), the CI will fail the check. This is crucial because broken links erode trust and usability of a knowledge base [16]. By catching them early, we keep docs "fresh" and user-

friendly [16]. We will configure the checker to ignore certain patterns (like placeholder links or anchors that might resolve on the live site). External links can be checked for HTTP 200 response or at least not obvious 404. This gate runs on every docs-related PR, and periodically (maybe weekly) on the whole site to catch any rot in older docs.

- **Front-Matter Validation:** Using the schemas for front-matter defined in section 2, we will run a script that parses the YAML front-matter of each Markdown file and validates required fields and allowed values. For example, ensure every file has a `type` and `title`. If a runbook is missing the `last_reviewed` date or an ADR missing `status`, the CI fails and reports which file and field is problematic. This ensures consistency and prevents omissions. We can implement this with a small Node.js or Python script using a YAML parser and our predefined schema (like using a JSON Schema validator). This gate helps enforce that everyone follows the template (no skipping metadata). It's especially useful as the number of docs grows, to quickly spot if someone forgot to set a status or used an unknown type keyword.

- **Linting and Style Checks:** We will incorporate a **Markdown linter** (like Markdownlint or Vale for prose) to catch style issues: e.g. heading levels in order, no trailing spaces, consistent spelling of terminology (we can maintain a glossary or dictionary for the tool). This gate is more about quality and consistency. For instance, we might enforce that sentences in documentation are <= a certain length for readability, or that passive voice is avoided in instructional steps (Vale can be configured for such rules). Another aspect is checking that code blocks or JSON examples in docs are valid (if we have JSON snippets, we could validate their syntax; if we embed Mermaid diagrams, ensure they compile correctly). Essentially, a **doc lint** job will run and fail on any formatting or style violations. We will adopt a baseline rule set and then refine as needed (possibly relaxing rules if too noisy).

- **Diagram Build Tests:** If we rely on generating diagrams from source (e.g. PlantUML files or Mermaid in Markdown), the CI will attempt to build these diagrams. For Mermaid in Docusaurus, the site build itself will catch syntax errors (failing the build). We can also explicitly run a Mermaid CLI to parse `.md` files for Mermaid fences to ensure no errors. For PlantUML, we might run a PlantUML jar on any `.puml` files to ensure they compile to SVG. By automating this, we avoid situations where a diagram fails to render on the live site due to a typo. We also might generate the diagrams as part of CI and store them as artifacts or commit them (some workflows commit the generated images to avoid runtime generation). The quality gate is that all diagrams must successfully generate.

- **Schema Sync (API & Data Schemas):** One challenging but valuable gate is ensuring documentation that describes schemas or APIs remains in sync with the actual schemas in code. We can implement targeted checks:

- For API docs: if we use an OpenAPI spec file in docs, we can validate it against the running code using a tool in CI (e.g. spin up the service and hit an endpoint to verify response shape, though that's complex for CI). More simply, ensure the OpenAPI YAML is well-formed and perhaps run `swagger-cli validate` on it.
- For data model docs: perhaps we have a JSON schema or TypeScript interface examples in the docs (for e.g. the Budget JSON structure shown in docs). We might automate comparing these with actual code definitions if possible. For example, if we maintain TypeScript types for certain config, we could generate JSON schema from code and diff with what's in docs (or have docs import from code if we set up that tooling).

- At minimum, this gate will flag if known schemas have changed in code without docs update. One approach: keep a reference of checksums. For instance, if the `budgets` SQL schema is documented, we store a checksum of the CREATE TABLE statement from docs. If in a future migration PR the actual table changes, the developer must update the docs and the checksum. This is somewhat manual but at least surfaces the need. More directly, we could integrate documentation in the dev workflow: perhaps require that any PR touching certain directories (like `/db/migrations` or `/api/routes`) includes a mention of docs or triggers a docs check. While full automation is tough, raising awareness is key. We will put in place lightweight scripts that search the repo for obvious mismatches (e.g. number of columns in a CREATE TABLE vs documented columns count).

- Also, we'll maintain a policy: any change to API or data models should either update an example in docs or confirm no docs impact. Code reviewers will have this in their checklist.

- **Documentation Coverage:** We want to ensure that for each significant code change or new feature, appropriate documentation is added or modified. While this is partially procedural (code reviews enforce it), we can automate some aspects:

- **Module coverage:** For each module in the codebase, ensure at least an overview doc exists. If a new module (say a new package or directory) is added in code, CI can warn if there's no corresponding doc. We can maintain a list of modules in a config, or derive it (e.g. if code has `packages/forecasting`, expect `docs/modules/forecasting.md`). This prevents completely undocumented modules.
- **Feature labels:** If using conventional commits or PR labels, we can key off those. For example, if a PR is labeled `feature` or touches certain areas, we could enforce a label like `docs-included`. Some teams use a PR checklist item "- [ ] Documentation updated". We might not fail CI automatically on that (to avoid false positives), but we can use GitHub's **merge checks** or branch protection requiring that checkbox to be ticked or a label to be set by a maintainer indicating docs are handled.
- Another approach: maintain a **doc TODO list** in the WBS for upcoming features. If a feature is completed but its doc task is not done, the project management process catches it. This is more outside CI but part of Quality Gates as a concept (quality gate can be process, not only tooling).

In summary, while we may not have a perfect metric for "coverage", we will implement at least a **"No feature left undocumented"** guideline with tooling to remind. Additionally, we can track the ratio of docs changes to code changes in a release – if it's low, that's a smell we address.

- **Consistent Terminology and Links:** Using our glossary, we can add a check that certain terms are linked or explained on first use. For example, the first time "Change-Set" appears in a document, ensure it's either defined or hyperlinked to the glossary. A simple regex check could find terms from a glossary list that are not linked. Or we could enforce italicizing or capitalizing standard terms. This is a softer gate; we might use Vale with a custom rule: e.g. "If the term 'Work Breakdown Structure' appears, suggest using abbreviation (WBS) after first mention and ensure glossary entry exists." While not a hard fail, these help maintain consistency especially important for AI parsing (consistent terms means easier retrieval).

- **Build and Preview:** Obviously, we treat a successful site build as a gate. The docs site should build without errors or warnings. CI will run `npm run build` (for Docusaurus or MkDocs build) and fail if

any warnings (broken link, unknown reference, etc.) occur. We will treat warnings as errors to maintain rigor.

All these quality checks run in CI via GitHub Actions. We might create a dedicated workflow (like `docs-ci.yml`) that triggers on PRs changing docs or on a daily schedule for full site scan. They can also run as part of the normal build if integrated.

Audit Trails for Docs: We will also have an **auditability** aspect: using Git history is one form (who changed what doc when). We can enhance this by requiring meaningful commit messages for doc changes (like "Docs: Update runbook X for new threshold limits"), and perhaps generating a **changelog** for documentation (point 9 covers publishing a changelog). This way, improvements or changes in docs themselves are tracked and visible to the team, not just silently updated.

Gate Maintenance: The CI rules themselves will be documented (in a contributor guide) so devs know what to expect. If a gate is too strict or causing false failures, we adjust or allow overrides (e.g. a broken external link that's temporary could be allowed via an ignore list in config). The goal is to **prevent common documentation issues** automatically, thus keeping the knowledge base high quality without requiring heroic manual effort for proofreading everything. Over time, as the project grows, these gates ensure scaling documentation without loss of consistency.

In essence, just as code has tests and linters, our docs have their own "tests". By failing fast in CI, we instill good practices (fix broken link now, not later) and keep the docs site professional and reliable. When these gates are in place, the team gains confidence that if the docs build passes CI, the published docs will be in good shape – no dead links, all sections properly labeled, and schemas consistent. This reliability is also crucial for the AI aspects: it ensures the AI isn't trained on outdated or incorrect documentation.

7. Documentation Search & Navigation Design

A well-structured knowledge base is only as useful as it is navigable. We will design the docs site's search and navigation features to make finding information intuitive, closely aligning with the project's module structure and terminology. This includes a clear taxonomy, user-friendly sidebar navigation, cross-references, and a glossary of terms.

Information Architecture & Taxonomy: The documentation will be organized in a hierarchy that mirrors the **Work Breakdown Structure (WBS)** and module architecture of the project: - **Project-Wide Docs:** At the top level, we'll have introductory and high-level docs (overview, architecture, glossary, how all modules fit together). - **Module Sections:** Each module (Frontend UI, AI/Intelligence, Forecasting, Budget, Revenue, Infra, etc.) has its own section. Within a module section, docs might be further grouped by type: e.g. *Overview*, *Design & Data Model*, *User Guide*, *API Reference (if module provides APIs)*, *Runbooks related to that module*. This grouping makes it easy for a developer or stakeholder interested in, say, the Budget module to see everything about it in one place. - **Operational Guides:** Separate sections for Runbooks and Playbooks (which might also be cross-linked from modules). Alternatively, we could integrate runbooks into module sections if they are module-specific. A likely approach: have a top-level **Operations** section that contains all runbooks/playbooks categorized by module or function, and also list each runbook under its module page as a reference. This way, ops folks can browse the centralized list, while devs focusing in a module's area also see relevant runbooks. - **Decision Log:** The ADR section will be outside the module grouping, since

decisions often span modules. It will probably live under a top-level **"Decision Log"** or **"Architecture"** category. - **API Reference:** If the system has a unified API reference (for external API), that can be a top-level section. If APIs are internal per module, they stay in module sections. We can do both: e.g. a top-level "API" page linking to subpages per module API. - **Glossary:** A single glossary page (or possibly one per domain, but likely one page) listing definitions of key terms (like "Change-Set", "WBS", "Gmail Hub", "Calendar Heatmap", etc.). This will be under an **Appendix** or **Reference** category.

We will implement the sidebar navigation accordingly. If using Docusaurus, we can define multiple sidebars or a nested sidebar. For example:

```
module.exports = {
  docs: [
    {
      type: 'category',
      label: 'Overview',
      items: ['overview/intro', 'overview/architecture', 'overview/glossary'],
    },
    {
      type: 'category',
      label: 'Modules',
      items: [
         {
           type: 'category',
           label: 'Forecasting',
           items: [
             'modules/forecasting/overview',
             'modules/forecasting/design',
             'modules/forecasting/api',
             // etc.
           ]
         },
         // ... other modules
      ]
    },
    {
      type: 'category',
      label: 'Decision Log',
      items: [...ADR files...]
    },
    {
      type: 'category',
      label: 'Operations',
      items: [
         'runbooks/index',
         // possibly categories by persona or module
         { type: 'category', label: 'Runbooks', items: ['runbooks/rb-001',
'runbooks/rb-002', /*...*/] },
```

```
        { type: 'category', label: 'Playbooks', items: ['playbooks/
onboarding', 'playbooks/incidents', /*...*/] }
      ]
    }
  ]
};
```

This results in a sidebar that clearly delineates sections. Users can collapse/expand modules. The **flow matches the mental model**: first an intro, then by module, plus global references.

We should also implement **prev/next navigation** within logical sequences, like within a module or within a multi-part tutorial (if any). But since most docs are reference-style, users will likely jump via sidebar or search.

Search Functionality: We will provide a full-text search for the knowledge base. If using Docusaurus, Algolia DocSearch can be set up, or Docusaurus' built-in Lunr-based search for smaller sites. For internal usage, we might use the built-in search which indexes all content. We need to ensure our content is easily searchable: - The **front-matter metadata** can enhance search results (some search systems can boost based on tags or title). - We will use consistent terminology so that searches yield relevant results (e.g. avoid too many synonyms for the same concept; pick one and stick with it, while listing synonyms in the glossary). - Search results should show the context (the page and a snippet). We'll verify that key pages (like important runbooks) are discoverable by common queries. If needed, we can add search aliases via hidden keywords in front matter (some systems allow a list of keywords for search).

We will also integrate the search with our **AI agent** approach: for example, a user query to the AI may use the same index as the site search behind the scenes. But for direct user search, we rely on the static site search.

Cross-linking and References: We will liberally cross-link documentation pages to each other to help navigation: - Within text, whenever a module or feature is mentioned, if there's a relevant doc, we'll hyperlink it at first mention. For example, "As described in the **Forecasting Model** module design, …" with a link. - Use relative links so they work on the site and in GitHub markdown preview. - Each runbook/playbook should link to any prerequisite docs (like "see API Key Setup for preparing credentials"). - At the end of pages, we can have a **"See Also"** section if appropriate (for manual curation of related content). - The Decision Log entries will often link to design docs or PRDs that motivated them, and vice versa (e.g., a design doc might say "Decision XYZ was finalized in ADR-0007").

Glossary and Terminology Linking: We will maintain a **Glossary** page listing key terms and definitions, especially important domain-specific terms and project jargon (like "WBS", "Change-Set", "Module Agent", "Tax Intelligence", etc.). Each entry provides a concise definition. To make these terms easily accessible: - The first time a glossary term appears in a page, we will highlight it and link it to the glossary. For example: "The system uses a **Work Breakdown Structure (WBS)** [17] approach to organize tasks…". In this snippet, "Work Breakdown Structure (WBS)" would be a link to the glossary entry for WBS, which might explain it in our context (project → module → job → run → step hierarchy). - We might use a special formatting or a plugin for glossary. Some documentation frameworks allow tooltip on hover that shows the definition. If we can, we'll enable that so users (and possibly AI) get quick context for terms. - We will ensure no term remains ambiguous: e.g. if "Change-Set" is a key concept, its glossary entry will define it (like "**Change-Set** –

A collection of code changes, configuration, and documentation that implements a distinct feature or fix, managed as a single unit of work (often a PR). Each Change-Set is associated with a WBS Job and can produce a changelog entry."). This clarity helps not just humans but also AI, which can then use the glossary definition to answer questions like "What is a Change-Set in this project?"

Sidebar Design: The sidebar will be structured with clear indentation and headings as described. We will also include perhaps icons or badges if supported (for example, tag internal-only pages with a small icon, though if the site is entirely internal, that's less needed). Each page's title will show in the sidebar for context.

For mobile or smaller screens, ensure the navigation is still accessible (Docusaurus handles this by a drawer menu). For MkDocs, similar.

Module Overviews: At the start of each module section, we'll have an overview page that briefly describes the module and lists relevant documentation pieces. Think of it as a "README" for the module docs. For example, the **Gmail Hub** module overview might list: its purpose, a diagram of its components, links to configuration instructions, links to runbooks for Gmail integration issues, etc. This overview serves as a hub for that module's info and can be a landing page from which one can dive deeper.

Navigation Aids: - A **global nav bar** can be enabled (like top menu) for quick jumps to key sections (like "Modules", "Runbooks", "ADR", "API"). This complements the sidebar. - Breadcrumbs will be shown at the top of pages (e.g. Home / Modules / Forecasting / Design) to orient where the user is. - We'll also incorporate a **prev/next** at page bottom for sequential reading in some sections (like an ADR might have prev = older ADR, next = newer ADR, or a multi-step tutorial have a sequence). - Possibly include **tags or indices** pages: Docusaurus supports tags for pages. We could tag docs by things like "compliance" or "performance" and have an index if it makes sense, but not crucial. More useful might be a **FAQ page** if needed, but likely the playbooks and runbooks cover that territory.

Matching Navigation to WBS: Since WBS is project → module → job → run → step, our docs navigation mirrors at least the project → module parts. We might even create a page that explicitly maps WBS items to docs: e.g. for each WBS Job, list if there's a PRD or design doc for it, which could be a way to ensure traceability (this might be overkill but could be an appendix or just managed via links in WBS tickets in issue tracker rather than in docs).

Search Engine (external): If any portion of docs is public, ensure that search engine indexing (SEO) is okay for those pages (proper meta tags, etc.). But likely this is internal, so not a big concern.

AI Considerations: The structure and linking also support the RAG pipeline. Because docs are categorized by module, if an AI knows a question is about the Budget module, it might bias search to that folder. The glossary linking ensures that even if an AI is unsure of a term, it can find its meaning easily by retrieving from the glossary page. We will treat the Glossary page as a high-value target in vector index (maybe boosting it so definitions are found first for single-term queries). Also, cross-links in content help because if the AI retrieves one chunk, that chunk might have links or context pointing to other relevant docs (which the AI can then follow up on by another retrieval using the linked page title as query, etc.).

User Testing: We will test the navigation scheme with sample tasks – e.g., "Find how to configure the Forecast model's parameters" – and ensure a user can either search "Forecast parameters" or navigate

Modules -> Forecasting and see a relevant page. If not, we adjust labeling. Our taxonomy labels (module names, doc titles) should use terms that users (developers or ops) naturally use. For instance, if “Calendar Heatmap” is a feature name but users call it “Calendar view”, maybe mention both. We might list synonyms in front matter keywords to aid search [18].

Finally, we will gather feedback on the docs site once up: see what pages are most visited (if using analytics internally) or what questions get asked repeatedly that maybe docs didn’t answer easily, and improve navigation or content accordingly. The goal is a self-service knowledge base where information is organized logically, reducing the time to find answers.

8. Compliance, Security & Privacy in Documentation

Since our documentation may contain sensitive information (designs related to financial data, possibly example data, or operational details), we need to enforce compliance and privacy measures. This ensures we protect confidential information and adhere to regulations (especially in finance domain, potential compliance like GDPR for any personal data). Key considerations:

Document Classification & Access Control: We will classify each document by sensitivity and intended audience, and handle accordingly [17]: - **Public:** Safe to share outside the organization (e.g. a user guide or API reference that might go to customers). Likely, most of our docs are internal, but perhaps some high-level ones or help docs could be public. Public docs will contain no secret or personal data. - **Internal:** Meant for internal team use only (e.g. most design docs, runbooks). They might contain technical details that we don’t want exposed but are not highly sensitive either. - **Confidential/Secret:** Contains highly sensitive info like security procedures, credentials, personal data examples, etc. These should be tightly controlled. Every doc’s front-matter will include a `classification:` field (or `visibility:`) with values like Public, Internal, Confidential [17]. By default, we’ll label all docs Internal unless explicitly needed outside. Confidential classification will be used sparingly, perhaps for security runbooks or anything with user data references. - The docs site (if deployed) should enforce that confidential docs are not accessible to unauthorized users. If the site is entirely internal, we already gate by login; if we ever publish a subset publicly, we will **exclude** confidential docs from the public build. We can achieve this by filtering in static site generation (some tools allow building only certain content) or simply by keeping two sets of docs. - The RAG pipeline will also use this classification to avoid feeding confidential docs to agents that serve users.

Redaction Patterns: We will adopt strict rules on including any sensitive data in documentation: - **No real personal data** in examples. For example, if showing an email from Gmail Hub, use a fake email like `user@example.com` instead of an actual user’s email. If showing a bank transaction, anonymize names (use Example Corp, John Doe, etc.). This prevents accidentally leaking customer or personal data in docs. - If real data must be referenced (say a case study), we’ll mask it thoroughly (names, IDs, etc.). Possibly use placeholders like `[REDACTED]` for anything sensitive. - **API keys, passwords, secrets**: Never include actual secrets in docs. Instead, show mock values (e.g. `API_KEY=XXXX-XXXX` or mention “stored in secret manager” rather than revealing). - We will define a set of patterns to scan for in CI (like anything resembling an AWS key pattern, etc.) in docs and fail the build if found. GitHub has secret scanning which might catch if committed, but we can double ensure by scanning markdown text for common secret regex (like `AKIA[0-9A-Z]{16}` for AWS keys). - For **financial data or PII** in examples, use synthetic or aggregated data. Possibly include a note in docs that all sample data is fictitious and for illustration.

Document Retention and Archival: As the project evolves, some docs (like early design drafts or deprecated feature docs) might become obsolete. We need a policy on how to handle them: - We won't delete ADRs or decision records – those are kept for historical record (but marked superseded if so). - For design docs of deprecated features, we can move them to an **Archive** section or mark clearly as obsolete at top (and exclude them from main navigation). Possibly front-matter `status: deprecated` and have our site generate a warning banner "This document is deprecated". This prevents confusion. - If there's data retention requirements (like remove data after X years), this typically applies more to production data than docs, but if any personal data found its way into docs, we should remove it immediately. Since we plan to avoid that, retention is more about knowledge relevance. We might decide to prune or archive content that is >N years old and no longer relevant, to reduce noise. But keep it accessible somewhere (like a separate branch or folder). - For compliance like GDPR: If any doc inadvertently contained personal data of an EU person and they had a right to erasure, we'd have to purge that from the docs (which is version-controlled, tricky because git history is append-only). This is yet another reason not to store such data in docs at all.

Privacy in AI Indexing: We have touched on ensuring the AI index doesn't use content that's sensitive for user-facing agents. Concretely: - The indexing script will skip `classification: Confidential` docs entirely, or include them only in an internal index that is not accessible to user-facing queries. - We might maintain two separate vector indexes: one "public/support" index and one "internal/devops" index. The agent will choose which to query based on context. This segmentation prevents any chance of leakage. - Also, even internal, if a doc is extremely sensitive (e.g. "Security Incident Response Playbook" might be classified confidential), we might not even want it in the vector DB in case that gets compromised. In such cases, we could decide those docs are only searched via direct keyword in a secure system, not broadly embedded. But given our scenario, likely okay if internal and secure.

Compliance Standards: Given this is a financial domain project, we should align doc practices with any standards: - If following something like ISO27001 or SOC2 compliance, documentation of processes (like runbooks) is actually a plus. But we also need to control access to them. Ensure only authenticated team members can see internal runbooks, etc. If the knowledge base is internal on the company network or behind login, that's fine. If using Vercel, we might use password protection for the preview domains or limit access. - Mark confidential docs with a header like "**CONFIDENTIAL – Do Not Distribute**" just to make sure if someone exports a PDF or copy-pastes it, it's labeled. - We'll adhere to data classification guidelines typical in finance: Public/Internal/Confidential categories as mentioned [17]. Training the team to classify content accordingly is part of our docs contributor guide.

Masking Policies: For logs or outputs included in docs (sometimes runbooks include sample log lines), ensure any IDs or personal info in those logs are masked. E.g. `user_id=12345` might be fine, but `email=john.doe@gmail.com` should be changed to a placeholder. If we include screenshots (less likely in a text-based approach, but if we did for UI guides), verify they contain no personal or live data (use demo accounts). - If any doc needs real data (like illustrating how a real bank statement looks), better to store that as a separate secure file or an image with redactions, rather than raw text.

Contributor Training: We will include in our docs guidelines a section about compliance: - A checklist for authors: "Before committing docs, ensure: no secrets, no personal data, classification set, etc." - Possibly a **pre-commit hook** that scans for common issues (just like secret scanning). - This reduces the chance of something slipping in.

Audit Logging for Documentation Access: If the knowledge base might be used by external parties or broad internal teams, consider logging access to confidential docs (this is advanced; not usually done for docs, but we can if needed). For example, if docs site is behind SSO, ensure that we can track who accessed what if it contains sensitive info.

Retaining History vs. Purging: One tricky thing: if we remove sensitive info from a doc, it may still exist in Git history. If something really sensitive got in and we needed to purge for compliance, we would have to rewrite git history (BFG Repo Cleaner, etc.) and force push, which is disruptive. So our strategy is prevention (not let it in). For safe measure, we can run a **one-time scan of the entire repo** for any 16-digit numbers, emails, etc., to ensure nothing slipped in from initial docs.

Compliance with Regulatory: If any regulatory documentation is needed (like if we have to document data flows for GDPR, or risk assessments), those docs too should be labeled and stored. Possibly an **Compliance** category in docs for things like "Privacy Policy" (internal) or "Security Architecture". Those might be confidential and audience-limited.

Data Masking in AI: When the AI uses docs to answer, we can program the agent to mask or paraphrase if needed. For example, if an internal runbook contains internal server names or IPs, the AI if talking to an end-user should not reveal those. We'll rely on classification filters to avoid that scenario entirely (the user-facing agent wouldn't have that runbook at all). For an internal agent, revealing an IP might be fine for a devOps context. But we'll caution the team to consider what they ask the AI to avoid inadvertently logging or sharing secrets. This goes a bit beyond docs into AI usage policy.

Conclusion on Privacy: We treat documentation with the same level of care as code or data when it comes to secrets and personal info. By classifying docs, redacting sensitive content, scanning for leaks, and controlling access, we build a documentation system that is **safe and compliant by design**. This means developers can document freely without risking a compliance breach, and the AI can leverage the docs with appropriate safeguards in place.

9. Publishing, Hosting & Preview Workflows

To make the documentation easily accessible and continuously up-to-date, we will set up an automated publishing pipeline, leveraging GitHub Actions and Vercel (the chosen deployment platform) for live previews and production hosting. The goal is that every change to documentation is instantly viewable in a shareable environment, and the main docs site is always in sync with the main branch.

Documentation Site Generator: We will use either **Docusaurus 2** or **MkDocs (Material theme)** to build the static site. Both are viable; Docusaurus fits well with our Next.js/React stack (enables using React components if needed, and has out-of-the-box support for versioning, search, and a plugin ecosystem), whereas MkDocs is Python-based but very simple and has a polished Material theme. We lean towards Docusaurus for its rich features and our familiarity with Node. Thus, we'll create a Docusaurus project in `/docs` (it can live inside the monorepo or as its own package in the repo).

CI/CD Integration with Vercel: We will take advantage of Vercel's Git integration for continuous deployment: - Connect the GitHub repository (or specifically the docs site subdirectory) to Vercel. We may set up the docs as a separate project on Vercel, using the `/docs` folder as the root. Alternatively, if the

whole monorepo is one Vercel project (since Next.js app might also be on Vercel), we might treat docs as a route in it. But likely cleaner to have a separate Vercel project for docs (e.g. docs.finosapp.com domain). - Vercel provides **Preview Deployments** for every branch/PR by default [19] [20]. When someone opens a PR with doc changes, Vercel will build the docs site on that PR's commit and provide a unique URL (like `https://docs-git-fix-typo-user.vercel.app`). This is immensely useful for review: the team can click the preview link and see the site exactly as it will look, including all navigation, images, etc., rather than trying to parse Markdown diffs. We will ensure this is enabled. GitHub can show a status check from Vercel and we can configure Vercel to leave a comment with the preview URL or mention it in the PR's checks. - **GitHub Actions:** While Vercel covers the deployment, we might still use GitHub Actions to run our quality gates and possibly to trigger some tasks. For example, after a PR is merged to main, we might want an Action to generate a **CHANGELOG.md** from change-sets (discussed soon) or to ping the search index update. But Vercel itself can be the one that triggers on merge to redeploy production docs.

Production Hosting: The main branch docs will be deployed to a production URL. We can use a custom domain like `docs.myfinosapp.com` or simply a Vercel domain. Possibly, if the app itself has a domain (like `finosapp.com`), we could host docs at `docs.finosapp.com`. Vercel makes domain aliasing easy. We'll also set up appropriate access: if internal only, perhaps behind login or at least obscure URL. But since it's likely internal, perhaps we restrict it by network or not worry if repo is private.

Docusaurus/MkDocs Setup: We will configure the site's theme for usability: - Ensure the **sidebar** is configured as per section 7. - Set up **search** (DocSearch might require the site to be public for their crawler, so if internal, use local search or consider self-hosting an Algolia server or use Typesense. But for a small team, Docusaurus local search is fine). - Enable **versioning** features in Docusaurus if needed in future; initially might not activate until first official release version. - Use plugins as needed: e.g. mermaid plugin for diagrams, footnote plugin if required, etc. - Possibly integrate an **announcement bar** or **banner** if we want to highlight things (like "Docs are in beta" or internal notes).

Previews for Diagrams and Content: Some documentation changes like updating a diagram might need visual verification. The Vercel preview serves this need: the author can open the preview and confirm the Mermaid diagram renders correctly or the new sidebar item appears in the right place.

Changelog from Change-Sets: The request mentions generating a changelog from Change-Sets. Here's how we approach it: - We define a *Change-Set* as a collection of changes usually corresponding to a feature or improvement, often represented by a PR or set of PRs. It might map one-to-one with WBS Jobs or epics. - We propose maintaining a file or directory where each significant change (feature or fix) has an entry (maybe a YAML or MD snippet). For example, under `/docs/changes/` we could have files like `2025-10-01-change-calendar-heatmap.yml` with fields: `id, date, module, description, issue/pr links`. - Using this, we can generate a **Changelog** page in the docs that collates these. Possibly group by release or time. If our project has releases (versions), tie them: e.g. "Release 0.5 – 2025-10-15" then list changes (with the descriptions from those files). - We can automate part of this: a GitHub Action could on each merge to main parse new change-set files and update a `CHANGELOG.md`. Or we could manually update one file. But a nice approach is to treat each change as a data item (like how many projects use towncrier or similar for managing changelogs via fragments). - For simplicity, maybe we maintain `CHANGELOG.md` by hand (with sections for unreleased and then release headings). But since the prompt explicitly says "from Change-Sets", they might want a more systematic approach. - We might incorporate the WBS integration: maybe each WBS Job completion corresponds to a changelog entry. The developer writes a short summary of the change in a YAML (with references to WBS and PR). CI validates its format (schema

validation as said) and once merged, a script appends it to CHANGELOG.md under “Unreleased” section. - Periodically (or each release), we fold unreleased into a release heading with date. - The docs site can then include the `CHANGELOG.md` as a page (nice for internal tracking and for users if it’s a user-facing product). - Also, these change-set files are a place to capture the documentation impact perhaps (“docs updated: yes, in XYZ page”), but since by merging they've presumably updated docs already, the change-set entry mostly serves to summarize.

In our design, the **Change-Set schema** might be:

```
id: CS-2025-001
title: "Added Calendar Heatmap visualization for budgets"
date: 2025-09-30
module: "Budget"
description: "Introduced a new Calendar Heatmap UI to visualize daily spending.
Accessible in the Budget module. Includes color-coded heatmap of spending
intensity."
wbs: "ProjectX-Budget-Job12"
pr: 42
```

From these, the changelog might render: - 2025-09-30 – **Added Calendar Heatmap visualization for budgets** (Budget module). Introduced a new Calendar Heatmap UI to visualize daily spending… (PR #42).

We will implement a docs page that lists these in descending date order or grouped by version if we add a `version` field.

Previewing Changelogs: Because docs are updated alongside changes, the changelog will also organically update. We might decide to only show released changes in the user-facing changelog and keep unreleased in a draft state. But since this is internal, it’s fine to see everything.

Integration with Git Workflow: - We might encourage using **Conventional Commit messages** or PR titles that we can parse for an initial changelog entry. But requiring manual entry is more straightforward and ensures context (plus allows non-code changes to be recorded if needed). - GitHub Actions could possibly auto-generate a draft release notes by scanning merged PR titles, but customizing it via our docs seems cleaner.

MkDocs Option: If we went with MkDocs Material: - We could use its built-in search and have a similar structure via nav in mkdocs.yml. MkDocs also supports versioning (via separate site instances per branch or plugins). - Deployment to Vercel works (just `mkdocs build` and serve static). We might choose Docusaurus anyway for easier React integration and possible future interactive docs (embedding our Next.js components in MDX if needed).

PDF or Export: Sometimes teams want to export docs as a PDF or other format for compliance or sharing. We might consider enabling a PDF export (Docusaurus doesn’t have out-of-box, but one could use something like `pandoc` or some plugin). Not a core requirement here, but we can mention that our docs-as-code approach allows generating such artifacts if needed.

PR Preview for diagrams: We covered that the preview environment will show diagrams. If, however, Vercel has trouble building (maybe if something not configured), we might have a fallback: a GH Action artifact or a Netlify preview. But likely Vercel is sufficient.

Notifications: We can set up that when docs are updated, maybe post in Slack a link to the preview or mention major updates in a channel so team is aware. This is optional but fosters documentation visibility.

One-click Publishing: After a PR is merged to main, Vercel will auto-deploy to production (docs main site) [21] [22]. So essentially publishing is continuous – no separate manual step. If we want to coordinate with code releases, we could lock docs deployment to releases, but it's usually better to just have docs always live (especially for internal use, up-to-date is crucial). If we cut version branches for docs, then merging to a version branch could trigger that version's site build.

Fail-safe: If a docs build fails (e.g. due to a bug in markdown or config), Vercel will report it. We mark the docs check as required for merging, so you can't merge a PR that breaks docs build. This ties back to quality gates.

Docusaurus vs. Integration with Next.js: Another approach could be embedding the docs into the Next.js app (like using MDX and rendering within the app's UI). But that complicates the separation of concerns. We prefer a standalone docs site so that the app remains focused on product functionality and docs can iterate and deploy separately. We can cross-link from app to docs (like a "Help/Docs" link opening the docs site). If SSO needed, we might integrate auth if internal (Docusaurus has some swizzles or we could put behind basic auth on Vercel if needed).

Summaries on PRs: Another small nice-to-have: if using a tool like `zentered/gh-pages-preview` GitHub Action, it could comment on the PR with "View docs preview here: <url>". But since Vercel does that natively in the checks, it might be enough.

Changelog Publication: If the changelog is mainly for internal awareness, just having the page is fine. If it's user-facing, we could also generate a markdown for release notes or GitHub Releases. Possibly automate that by pushing changes to GitHub Releases via API. But might be overkill; simply having a "Changelog" page in docs and maybe an entry in the project CHANGELOG.md (which can link to docs for details) is fine.

In summary, the publishing pipeline will give us: - **Instant Previews:** every docs update PR gets a live site for review [22]. - **Auto Deployments:** merges go live to the docs site within seconds. Team members can always refer to the latest docs at the known URL. - **Continuous Changelog:** the docs site itself serves as the hub for changes and updates, meaning our documentation also documents itself (changes to system features). - **Integration with Code Releases:** if we do official product releases, we can freeze docs versions. For now, continuous deployment is agile and fitting.

This modern docs deployment setup ensures the documentation is not a static afterthought but an actively developed component of the project, with the same agility as the code. It lowers friction to update docs (since you get immediate feedback and publishing), encouraging contributions and thereby keeping the knowledge base current.

10. WBS Integration and Seeded Backlog

To ensure the documentation & knowledge base system is implemented efficiently and comprehensively, we will integrate it into the project's **Work Breakdown Structure (WBS)**, breaking down the work into Project → Module → Job → Run → Step hierarchy. Below is a **seeded backlog** for the Documentation Platform (as a project module itself), including tasks with testable acceptance criteria. This WBS-aligned plan will guide a single developer (or small team) to implement the documentation system in iterative steps:

- **Project:** *Financial OS Documentation & Knowledge Base*

- **Module:** *Documentation Platform Infrastructure* – (This module covers all work to create the docs system and integrate with CI/AI. WBS items below are under this module.)

    - **Job 1: Documentation Site Scaffolding** – *Set up the skeleton of the documentation site.*
    - **Run 1.1:** Initialize Static Site
        - **Step 1.1.1: Choose and install docs framework** (Docusaurus 2) in the `/docs` directory. *Acceptance:* Running `npm run start` in `/docs` opens a dev server with the default Docusaurus homepage.
        - **Step 1.1.2: Configure basic site metadata** (site name, logo placeholder, navbar links). *Acceptance:* The site's title is "FinOS Docs" and a placeholder logo appears, with a navbar link to "Modules" and "ADR" sections.
    - **Run 1.2:** Implement Docs Structure
        - **Step 1.2.1: Create folder structure** under `/docs` for modules, adr, runbooks, etc., and add sample markdown files (e.g. one per section). *Acceptance:* Sidebar shows intended sections (Overview, Modules list, Decision Log, Operations) with placeholder pages for at least one module and one ADR.
        - **Step 1.2.2: Configure sidebar navigation** in Docusaurus (`sidebars.js`) reflecting our taxonomy (section 7 layout). *Acceptance:* Sidebar on site matches the planned hierarchy (module sections nested, etc.), and links work (even if content is minimal now).

    - **Run 1.3:** Theming and Layout

        - **Step 1.3.1: Apply styling to match project branding** (ensure Tailwind or custom CSS for colors that align with app, integrate Tailwind if possible or use Docusaurus theme customization). *Acceptance:* Documentation site color scheme and font reflect the app's design (e.g. uses Tailwind's config or custom CSS) and is responsive.
        - **Step 1.3.2: Add Home/Landing page content** (`docs/index.md` or Docusaurus homepage) with an introduction and quick links. *Acceptance:* The docs homepage greets the user with project name and sections overview, and no default Docusaurus filler content remains.

    - **Job 2: Content Model & Templates Implementation** – *Define and implement documentation content standards.*

- **Run 2.1:** Define Front-Matter Schemas
  - **Step 2.1.1: Write YAML front-matter conventions** for each doc type (PRD, ADR, Runbook, etc.) – document this in a `CONTRIBUTING.md` or in the docs contributors guide. *Acceptance:* A "Docs Style Guide" page exists in docs explaining front-matter fields required for each type.
  - **Step 2.1.2: Create example templates** in a `/docs/templates/` directory (or as comments in the style guide) that authors can copy. *Acceptance:* E.g. `docs/templates/adr-template.md` and others are present with placeholder fields and instructions.

- **Run 2.2:** Populate Initial Content

  - **Step 2.2.1: Migrate existing design guides into docs** – e.g., take content from provided PDFs like the forecasting blueprint, budget module guide, etc., and convert to markdown under appropriate sections. *Acceptance:* At least one sample (e.g. Budget Module spec) is available in the docs site, properly formatted with headings and maybe front-matter (type: DesignDoc).
  - **Step 2.2.2: Create stub for glossary** and add 5-10 key terms with definitions. *Acceptance:* `glossary.md` exists with definitions for "WBS", "Change-Set", "Gmail Hub", etc., and those terms are hyperlinked in at least one place in docs.
  - **Step 2.2.3: Add one example ADR** (e.g. ADR-0001 for choosing docs-as-code) in `/docs/adr`. *Acceptance:* ADR appears in Decision Log section, with proper front-matter and content using the template.
  - **Step 2.2.4: Add one example Runbook and Playbook** (maybe a simple one each, like "Restart Service X" and "Onboard New User Playbook") to demonstrate format. *Acceptance:* They show up under Operations section and have numbered steps, etc., as per template.

- **Job 3: CI Pipeline & Quality Gates** – *Automate validation and deployment of docs.*

- **Run 3.1:** Continuous Integration for Docs
  - **Step 3.1.1: Set up Markdown linting** (using Markdownlint CLI or a GitHub Action) to run on PRs. *Acceptance:* A PR with a lint issue (like trailing whitespace or bad header) gets a CI failure pointing out the line.
  - **Step 3.1.2: Implement front-matter validator** script. *Acceptance:* If a doc is missing a required field or has an invalid value, the CI job fails and outputs the file and field error.
  - **Step 3.1.3: Configure broken link check** (use a tool or Docusaurus's build with link checking). *Acceptance:* CI intentionally break a link in a doc on a test branch, see the CI fail highlighting the broken link.
  - **Step 3.1.4: Add secret scan for docs** (maybe using trufflehog or built-in GH secret scan on the repo). *Acceptance:* If someone accidentally writes an AWS key in docs, CI catches it (simulate with a fake pattern for test).
- **Run 3.2:** Preview & Deployment Integration
  - **Step 3.2.1: Add GitHub Action for docs build** (or rely on Vercel). Since using Vercel, ensure project is set and environment variables if any (like Algolia keys) are configured. *Acceptance:* On pushing a docs branch, Vercel posts a preview link in PR checks.

        - **Step 3.2.2: Protect main branch with docs CI** – mark the docs build and checks as "required" in GitHub branch settings. *Acceptance:* If docs build or checks fail, the PR cannot be merged.
        - **Step 3.2.3: Verify production deployment on merge**. *Acceptance:* After merging to main, the docs site at official URL updates within a couple minutes with the changes.

    - **Run 3.3:** Doc Coverage Enforcement (Process)

        - **Step 3.3.1: Create PR template section for docs**. *Acceptance:* When opening a PR, the template asks "Documentation updated? (Y/N, explain)" as a checklist item.
        - **Step 3.3.2: Implement simple module-doc mapping check**. (Optional) e.g., a script to list all modules from code and see if corresponding doc exists. *Acceptance:* For test, if a dummy "moduleX" appears in code without docs, the script warns (can be in CI logs but not failing).

- **Job 4: RAG Pipeline Implementation** – *Enable AI indexing of docs.*

    - **Run 4.1:** Build Initial Embedding Index
        - **Step 4.1.1: Select embedding method** (OpenAI API vs open-source). Implement a script `embed_docs.py` that reads markdown files, uses front-matter to decide inclusion, splits text using rules. *Acceptance:* Running the script generates an embeddings JSON or uploads to vector DB, logging each processed file.
        - **Step 4.1.2: Chunk by heading algorithm** – ensure the script splits by our chunking rules (maybe use a library with config). *Acceptance:* Check output chunks: no chunk exceeds 512 tokens and chunks align with doc sections (visually inspect a few).
        - **Step 4.1.3: Include metadata in index** – ensure each chunk record has doc title, type, module, classification. *Acceptance:* The output JSON entries clearly show these fields alongside the vector.
    - **Run 4.2:** Integrate with CI
        - **Step 4.2.1: Secure API keys** (if using external embedding API) in GitHub Actions secrets, or set up local model container if offline. *Acceptance:* The embedding script can run in CI without exposing keys (e.g. using GH encrypted secrets).
        - **Step 4.2.2: Add CI job to update embeddings on docs changes**. *Acceptance:* After merging a docs change, the CI pipeline runs `embed_docs.py` – if using an external DB, confirm new content is searchable; if storing file, ensure it's updated.
        - **Step 4.2.3: Test query to agent** – (Manual step) simulate an agent querying the index for a known answer. *Acceptance:* For example, ask the dev agent "What is WBS?" and confirm it finds the glossary definition. This tests end-to-end retrieval.
    - **Run 4.3:** Permission Controls
        - **Step 4.3.1: Implement filtering by metadata** in the retrieval function (if we have a custom retrieval code or ensure chosen vector DB supports metadata filtering). *Acceptance:* Query using a filter (e.g. only `module:Forecasting`) returns only chunks from that module's docs.
        - **Step 4.3.2: Partition indexes if needed** – e.g. create two separate indices for internal vs public. *Acceptance:* Confirm that the public agent's config points to index without internal docs by trying a known internal query (should get nothing).

- **Run 4.4:** Documentation for RAG
    - **Step 4.4.1: Document the RAG pipeline** in the repository (how to run embed script, how to add new docs and re-index). *Acceptance:* A readme or section in docs describes how the embedding update works and how developers can use the AI agent with it.
- **Job 5: Compliance & Privacy Measures** – *Apply security controls to docs.*
- **Run 5.1:** Implement Classification Labels
    - **Step 5.1.1: Add `classification` field to front-matter where needed** (maybe globally default to Internal). *Acceptance:* All ADRs and Runbooks get `classification: Internal` by a batch edit, and one doc is marked Confidential as a test.
    - **Step 5.1.2: Update site UI to indicate confidential docs** – e.g. if `classification: Confidential`, add a badge or watermark via custom Docusaurus component. *Acceptance:* A "Confidential" ADR shows a red badge at top.
- **Run 5.2:** Redaction & Scanning
    - **Step 5.2.1: Integrate secret scanning** (we already did in CI gates) – verify it runs on docs too. *Acceptance:* (Test with a dummy secret in a draft PR, see it blocked).
    - **Step 5.2.2: Set up regex find for PII** (like emails, credit card formats) within docs for periodic audit. *Acceptance:* Running a script finds no real personal data; add one fake email in a doc for test and see script flag it.
    - **Step 5.2.3: Review existing docs for sensitive info** – manual task: ensure things like API keys in examples are obviously fake. *Acceptance:* Developer confirms all examples sanitized (no live keys).
- **Run 5.3:** Access Control
    - **Step 5.3.1: Restrict public access if needed** – if hosting on Vercel, configure password or allow only certain email domains if we want to limit. *Acceptance:* Hitting the production docs URL prompts for login if not authorized (if decided necessary).
    - **Step 5.3.2: Test classification filtering in RAG** (tie with Run 4.3) to ensure e.g. Confidential docs are not in public index. *Acceptance:* A query for something only in a confidential doc yields nothing in the public agent.
- **Run 5.4:** Compliance Documentation
    - **Step 5.4.1: Write a short "Docs Security Policy"** page in docs outlining how we classify and handle docs (for transparency and team reference). *Acceptance:* Page exists under maybe Overview or an Appendix, stating: "All documentation is Internal unless labeled Public. Confidential docs are limited… etc."
    - **Step 5.4.2: Train team on procedures** (perhaps an internal email or meeting). *Acceptance:* All contributors are aware of how to classify docs and the do's/don'ts of adding content (this is more process than a deliverable).

Each **Step** above has an acceptance criterion that is testable, either by observing the docs site behavior or CI output. As the developer works through this WBS, they will deliver a fully functional docs system: - Job 1 and 2 deliver the structure and initial content standards. - Job 3 integrates it into our development workflow (no broken builds, easy previews). - Job 4 connects it with our AI layer, making docs queryable. - Job 5 ensures we're not exposing what we shouldn't and that authors follow privacy rules.

This plan is designed to be executed incrementally (some jobs can be parallel, but roughly this order works – e.g. scaffold before worrying about CI, CI before RAG, etc.). By completing these WBS items, we achieve a production-grade documentation and knowledge base system aligned with our multi-module financial OS, ready for immediate use and future-proof for growth.

---

1 2 20 21 22 Docs as code hosting with standard tools - Read the Docs
https://about.readthedocs.com/docs-as-code/

3 18 Front matter | Writers' Toolkit documentation
https://grafana.com/docs/writers-toolkit/write/front-matter/

4 10 Architectural Decision Records (ADRs) | Architectural Decision Records
https://adr.github.io/

5 8 9 Why you should be using architecture decision records to document your project
https://www.redhat.com/en/blog/architecture-decision-records

6 How to Create a DevOps Runbook Template | FireHydrant
https://firehydrant.com/blog/runbook-template-devops/

7 6-Unified Revenue and Expenses Engine_ Implementation and Design Guide.pdf
file://file-MWYdtvR31y9Gk4NVw1MB3N

11 13 14 Chunk documents in vector search - Azure AI Search | Microsoft Learn
https://learn.microsoft.com/en-us/azure/search/vector-search-how-to-chunk-documents

12 What is the optimal chunk size for RAG applications?
https://milvus.io/ai-quick-reference/what-is-the-optimal-chunk-size-for-rag-applications

15 The Importance of Pipeline Quality Gates and How to Implement Them
https://www.infoq.com/articles/pipeline-quality-gates/

16 How to find and fix broken links on your knowledge base - Document360
https://document360.com/blog/how-to-find-and-fix-broken-links-on-your-knowledge-base/

17 Document Classification Confidential: Levels and Protocols - Deasy Labs: Efficient Metadata Solutions for Scalable AI Workflows
https://www.deasylabs.com/blog/document-classification-confidential-levels-and-protocols

19 How can I use GitHub Actions with Vercel?
https://vercel.com/guides/how-can-i-use-github-actions-with-vercel