Discord CM_Deep Research - 19 july 25

Discord-Based Hierarchical Context and Prompt Generation System

Table of Contents:

---

0. Executive Summary

We propose a comprehensive **AI-supported work environment** within Discord that manages knowledge, context, and prompt generation across a nested hierarchy of **Client** → **Product** → **Project** → **Subproject** → **Task**. The system ingests all project-related information (documents, Markdown notes, meeting transcripts, etc.) via a dedicated **“dump” channel**, automatically classifies the content using metadata, and routes it into structured **context vaults** scoped to each level of the hierarchy. At each level, an AI **bot** provides contextual assistance: retrieving relevant information, answering questions, and helping to compose high-quality prompts or summaries appropriate to that level.

Key Features of the System:

- **Hierarchical Context Management:** Information is organized and abstracted at multiple levels of granularity. Lower levels (Tasks, Subprojects) retain rich **nuance** (detailed discussions, exploratory thoughts, edge cases), whereas upper levels (Projects, Products, Clients) contain distilled **basis** information (final decisions, requirements, outcomes). As one moves up the hierarchy, context becomes more synthesized and high-level. This **abstraction gradient** ensures that each level gets the right amount of detail: Tasks have all the fine-grained context needed for execution, while Product/Client views see only consolidated insights and decisions.
- **Multi-Bot Architecture:** A network of bots operates at different scopes. For example, a **Task Bot** assists within a task-specific Discord thread, a **Subproject Bot** oversees a subproject channel, up to a high-level **Client Bot** that can provide an overview for client-facing queries. Each bot “knows” the objectives and constraints of its scope and can pull in relevant context from its own vault and related levels. The bots also coordinate: e.g., a Task Bot can fetch higher-level constraints from the Product Bot’s knowledge, and the Project Bot can aggregate updates from Subproject Bots. This layered bot approach ensures that any query or prompt at a given level is answered with information that is both relevant and appropriately detailed for that level.
- **Automated Ingestion & Classification:** All raw information (Markdown files with YAML frontmatter, Word/PDF docs, transcripts, etc.) is first dropped into a **central ingestion channel** (managed by a “Dump Bot”). Using a predefined **metadata schema** (provided via Markdown frontmatter or other cues), the Dump Bot parses each item and determines its placement: which client/product/project it belongs to, whether it pertains to a specific segment (e.g., “Research” or “Timeline” within a project), and whether it’s a draft or final version. The content is then stored in the corresponding context vault for that level and category. When metadata is missing or ambiguous, the system applies fallback heuristics (such as keyword analysis or manual confirmation) to classify the

content. This automation reduces organizational overhead and ensures information is available in the right context without manual sorting.

- **Task Context Lifecycle:** Tasks are the smallest unit of work and have a clear lifecycle. During a task's execution (often happening in a Discord thread), there is a **divergent phase** of exploration where team members discuss, brainstorm, and iterate, generating ephemeral context (scratch notes, intermediate outputs, Q&A with the Task Bot, etc.). Once the task reaches a conclusion, the valuable results are extracted as a **finalized task artifact** (e.g., a decision record, summary of findings, or deliverable). This artifact is then **promoted upward**: it gets stored in the parent Subproject's vault (and possibly summarized further for the Project level). The Task's thread can then be archived. In this way, each Task produces a permanent, distilled contribution to the project's knowledge, while the noisy transient discussion can be left behind or referenced only if needed. Over time, this process builds a knowledge base of task outputs that feed into higher-level progress updates.
- **Context Propagation with Abstraction:** The system enforces a disciplined approach to context sharing:
  - At the **Task level**, nearly all relevant nuance from the subproject and any pertinent higher-level info is available to ensure no detail is missed.
  - At the **Subproject level**, context is a mix of some nuance (to understand how results came about) and synthesized outputs from tasks.
  - At the **Project level**, context is further abstracted: mostly summaries of subproject outcomes and key decisions, categorized by segments like Operations, Research, etc.
  - At the **Product level**, only high-level basis information is retained (core requirements, final designs, metrics, key decisions across projects).
  - This controlled flow prevents information overload at high levels and context starvation at low levels. Each bot is responsible for pulling the **right amount of context** for its level. For example, when generating a prompt for a Task, the Task Bot might automatically include a recent decision from a relevant past task and a pertinent design principle, whereas the Product Bot answering a question would only reference the final outcomes of those tasks, not the play-by-play.
- **Design Insights Library:** In addition to project-specific knowledge, the system maintains a library of **Design Insights**. These come in two flavors: **Local Insights** tied to a particular project (lessons learned, user feedback, retrospective notes for that project), and **Global Frameworks** that represent universal principles or best practices (e.g., general UX rules, design heuristics the organization follows). All bots have access to the global insights and will inject them into conversations or prompts when appropriate. For instance, if a task is about designing a login flow, the Task Bot might automatically surface a global design guideline about authentication UX. Local insights are used when similar situations arise within the same project or product. This integration ensures that hard-earned lessons and corporate knowledge inform ongoing work continuously.

- **Personal Workspaces and Person Bots:** Each team member (especially interns or individual contributors) has a personal context vault and a **Person Bot** that helps them manage it. In a private Discord channel or via integration with a personal Obsidian vault, an individual can take daily notes, experiment with prompt ideas, jot down thoughts, or maintain a to-do list. The Person Bot acts as their personal assistant – it can answer questions using both the person's private notes and the project data that the person has access to. Personal notes can remain private or, if the individual decides, be shared with the team by pushing them to the dump channel with appropriate metadata. For example, an intern might draft an analysis in their space, then publish it to the relevant subproject by tagging it and sending it through the Dump Bot. The Person Bot also tracks the individual's assignments and deliverables (e.g., which tasks they are responsible for) and can remind them or help fetch relevant context for their work specifically. This personal layer ensures individuals can work things out in a sandbox before sharing and have a tailored view of the project that matters to them.
- **Access Control and Context Privacy:** The hierarchical design inherently provides **data segmentation** by client and by role, but additional access controls ensure the right people see the right level of detail. Each content item carries permission metadata (e.g., default vs. restricted) and inherits visibility rules from its level. For instance, detailed internal discussions (nuance) might be restricted to the team and not visible to a client-facing summary. Bots are aware of these permissions and will **redact or omit sensitive details** when providing context upward or to users without clearance. For example, if a Project Bot is generating a summary for a stakeholder who shouldn't see internal budget details, it will exclude or generalize that information. All promotions of content upward can be logged, with attribution to who approved or initiated it, creating a clear audit trail of how information flows and transforms. This ensures confidentiality and clarity are maintained throughout the knowledge base.
- **Discord-Native with Optional GUI:** The entire workflow is designed to function within Discord, leveraging channels for hierarchy (e.g., a channel per Project, threads for Tasks) and bot interactions for assistance. This meets users where they already collaborate. Optionally, we envision a lightweight **dashboard GUI** that provides an overview of the active projects and tasks, hierarchy navigation, and perhaps controls for administrative actions (like archiving a subproject or reviewing pending summaries). The GUI could show a tree view of Client→Product→Project→Subproject→Task, highlight which subprojects or tasks are currently active, list team members and their person spaces, and indicate data freshness (e.g., last update time for each context vault). This is mainly to give managers and team leads a quick visualization of the workflow state; all content creation and discussion still happen in Discord.

Conclusion of Summary: This architecture is aimed at **maximizing relevant context for each task while minimizing overload**. By structuring information by level and intelligently retrieving it, the system ensures that team members (and the AI assistants) are always informed by the latest and most pertinent knowledge. It blends automated organization, AI summarization, and human oversight to create a continuous knowledge lifecycle: from raw data ingestion, through active task work, to refined knowledge at higher levels, and back to supporting new tasks. In the following sections, we detail each aspect of this system, including metadata design, storage strategy, context retrieval algorithms, bot behaviors, lifecycle management, and more, and provide concrete specifications and examples to guide implementation.

---

1. System Scope & Boundaries

This section defines what is included in our system design and what is considered out-of-scope, to clarify the focus of the architecture.

In Scope:

- **Discord as the Primary Interface:** The system will be implemented within a Discord workspace, using Discord channels, threads, and bots for all user interaction. All team members already use Discord for communication, so our solution augments this existing workflow with AI capabilities. There is no separate application for daily use (the optional GUI is just a read-only dashboard); Discord remains the hub for dumping information, discussing tasks, and receiving AI assistance.
- **Hierarchical Project Structure:** We assume a fixed hierarchy of **Client** → **Product** → **Project** → **Subproject** → **Task** to organize work. This reflects a consulting or multi-product environment where a company (client) has one or more products or initiatives, each product has projects, projects may be broken into subprojects or workstreams, and each subproject consists of granular tasks. The system is built around this structure:
  - *Client:* Top-level entity, possibly representing an external client or internal division. Contains one or more products or programs.
  - *Product:* A major product, program, or workstream under a client. Contains projects.
  - *Project:* A specific project or engagement with a defined goal, under a product. Projects are further divided into segments for organization (like Operations, Research, Timeline, Design, etc., which are categories of information within the project context).
  - *Subproject:* A smaller scope of work or phase within a project. Only one subproject is actively in focus per project at a given time (as an assumption to simplify context switching), while others might be on hold or completed.
  - *Task:* The smallest unit of work, usually executed as a Discord thread under a subproject channel. Many tasks can be executed (often sequentially or with limited parallelism) to complete a subproject.
- **Project Segmentation:** Each Project's information is categorized into **segments** such as Operations, Research, Timeline, Design, etc. These represent different facets of project data:
  - *Operations:* Might include planning, meeting notes, logistics.

- *Research:* User research findings, benchmarks, exploratory analyses.
- *Timeline:* Schedules, gantt charts, milestone tracking.
- *Design:* Design documents, wireframes, design decisions.
- *(Extendable:* The system allows adding more segments as needed.) These segments help filter and retrieve information (e.g., a query about schedule should search the Timeline segment of the project). In implementation, segments could correspond to tags or folders within a project's context vault.

- **Central Dump Ingestion:** A designated Discord channel (or set of channels) acts as the **intake point for all raw information**. Team members will upload or paste:
  - Markdown files (with YAML frontmatter metadata, as detailed later),
  - Other documents (Word, PDF) which the Dump Bot will OCR or convert to text and attempt to classify,
  - Meeting transcripts (likely as .txt or .md files after using a transcription service like Whisper),
  - Images or other media (these might be handled via references or manual steps if text extraction is needed). The Dump Bot monitors these channels, processes incoming files, and then **routes them to the appropriate context vault** in the hierarchy based on their metadata or content.
- **Automated Metadata Parsing and Classification:** The system relies on a **metadata schema** (proposed in section 5.2) to guide classification. For example, a Markdown file might declare itself as level: project, project: RedesignWebsite, segment: research. The Dump Bot uses this to store the file's content in the "Research" segment of the RedesignWebsite Project vault. If metadata is incomplete or a file has no frontmatter, the bot will apply some intelligent heuristics: it might look at the channel it was posted in (e.g., if someone drops a file in a subproject-specific channel, assume it belongs to that subproject), examine the file name or content for keywords (project names or known terms), or as a last resort, ask the user to clarify via a prompt. The classification process also involves setting statuses (draft vs final) and linking the content to any related items (for instance, if a transcript corresponds to a specific meeting or a task ID, that link is recorded).
- **Context Vaults for Each Node:** A **Context Vault** is a logical storage area for all content relevant to a particular node in the hierarchy (client, product, project, subproject, task, or person). It can be implemented under the hood with a combination of storage backends (discussed later), but conceptually:
  - Each Client has a vault containing high-level documents (e.g., contract, high-level reports) and aggregated knowledge of its products.
  - Each Product has a vault with documentation of that product (specs, roadmap) and distilled outputs from its projects.
  - Each Project has a vault that includes all project documents, organized by segment, as well as summaries of subprojects.
  - Each Subproject vault contains detailed information from tasks and any subproject-level docs (plans, open issues list, etc.).

    - Each Task vault (which might just be the Discord thread plus some scratchpad storage) holds the running conversation and any attachments or notes specific to that task, as well as pointers to the relevant subproject context.
    - Each Person has a personal vault for their notes and data (accessible primarily by their Person Bot). These vaults not only store raw documents, but also indexes or embeddings to facilitate retrieval. They provide **scoped views** of the knowledge base so that queries can be answered with localized information (e.g., a search within a project's vault yields project-specific results).
- **Task Threads and Lifecycle:** For each Task, a dedicated Discord thread is used for discussion and AI assistance. The **Task Bot** participates in this thread, providing relevant context from the subproject and higher levels as needed. Team members can ask the Task Bot questions ("What do we already know about X?") or ask it to summarize interim findings. The Task Bot also keeps track of important points made in the thread (like decisions or discoveries). When the task is completed (manually marked or inferred after a period of inactivity or via a specific command), a **final artifact** is produced. This could be done by the Task Bot automatically synthesizing the discussion into a summary or by the human posting the outcome (e.g., "Conclusion: We will implement feature A because…"). The final artifact is then extracted from the thread, saved to the Subproject vault (possibly in a distilled form), and the thread can be closed or archived. This mechanism prevents the accumulation of massive, unstructured chat logs—useful information is systematically captured and elevated.
- **Progressive Upward Synthesis:** The system supports a **layered upward flow of information**:
    - Task outputs feed into **Subproject updates**.
    - Subproject outcomes are summarized into **Project updates**.
    - Project updates roll into **Product-level** knowledge (ensuring the product context stays current with all project changes).
    - Finally, product snapshots can inform **Client-level** reports. Each step involves abstraction: the content is summarized to fit the higher scope and audience. This may be initiated by bots but will typically require human review or trigger (especially for significant updates). The "upward synthesis" can be on-demand (e.g., a project lead requests a project summary after several subprojects complete) or event-driven (e.g., when a subproject is marked finished, automatically generate a project segment update). The result is that at any given time, a manager or stakeholder can ask the higher-level bot "What's the status of Project X or Product Y?" and get an up-to-date answer drawn from the accumulated knowledge of all sub-parts, without needing to dive into low-level details.
- **Design Insights Integration:** Two stores of design knowledge are maintained:
    - Project-specific **Local Design Insights**, which are typically documents or notes stored in the project vault (possibly in a special "Insights" segment). These could come from retrospectives, QA feedback, or mid-project evaluations.

  - A **Global Design Frameworks** library, which is a collection of general principles, guidelines, and best practices the organization wants to reuse across projects (for example, a style guide, UX heuristics, coding standards, etc.). This is logically separate from any one project (perhaps stored in a top-level “Global” vault or associated with the organization or client overall). Both sets of insights are indexed such that bots (especially Task and Subproject bots) can retrieve them contextually. For instance, if a task involves designing a sign-up flow and there’s a global principle about “Minimize user friction during sign-up”, the Task Bot should surface that principle at the right time. Likewise, if in a previous subproject the team learned “Users preferred single sign-on,” and that was recorded as a local insight, a new task in the same project dealing with login should retrieve that tidbit. The integration of these insights ensures that both organizational memory and project memory inform current work.
- **Personal Context and Person Bots:** Each collaborator can maintain a **personal workspace**, e.g., an Obsidian vault or a private Discord channel, which is integrated via their Person Bot. In scope is the syncing or ingestion of personal notes when the user chooses. For example, an intern might write a personal journal entry about a problem they solved; if they tag it with a project ID, their Person Bot could automatically share it to the project’s context after confirmation. The Person Bot can also pull in information from the shared space for the user: an intern could ask their bot “What did I work on last week?” and it can compile their task summaries, or “Show me all my notes related to Project Z” and it will filter their personal notes. This ensures individuals benefit from the knowledge base personally, and can contribute back to it seamlessly.
- **Single-Organization Deployment:** The architecture assumes we are deploying this for *one organization* that might handle multiple clients internally. Multi-tenancy (i.e., a SaaS that multiple unrelated orgs use with strict siloing) is **out of scope**. Instead, our focus is that within this org’s Discord, there may be multiple clients handled, but all under this one installation. We will leverage Discord’s existing permission model (roles, private channels) as needed to separate client-specific info if needed (for example, if clients are external and could have access to some channels, though likely this is an internal tool, so primarily the team sees everything and just conceptually separates client data). We won’t design features like separate billing or admin consoles for multiple organizations – those are beyond our current needs.
- **Moderate Scale (Cost-Aware):** The solution will be designed for a moderate scale of data and users. Perhaps dozens of projects, hundreds of tasks, and a team of, say, 5-20 active users (interns, leads, etc.). We will prioritize using **open-source and cost-effective technologies**. For instance, we plan to:
  - Use the open-source **Whisper** model (possibly hosted on our server) for transcribing audio to text, avoiding recurring API costs for transcripts.
  - Use a local or open-source **vector database** (like FAISS, Chroma, or an embedded library in our app) for semantic search, instead of paid services.
  - Where possible, run **LLMs on-device or on our server** (using models like LLaMA or other efficient models) for tasks like summarization or prompt generation, to

minimize calls to expensive APIs. We may, however, call high-end models (GPT-4 or similar) for critical tasks where quality is paramount (like final document summarizations for clients), but we will design the system to do as much as possible with cheaper alternatives.

- Keep data storage solutions simple and within our control (using either a small database or even markdown files in a repository for now) to avoid cloud costs at the early stage. The architecture will be scalable in design (so we could swap in more robust technologies as needed), but our immediate implementation will aim to be **lean in cost** and complexity.

Out of Scope (for now):

- **Enterprise Integrations:** We are not focusing on things like SSO/enterprise authentication integration, external project management tool integrations, or enterprise analytics. The system is largely self-contained around Discord and a backend store. We assume team members will use Discord credentials to access, and we won't develop separate user management beyond what Discord provides.
- **Advanced Analytics & Dashboards:** While we provide a simple GUI idea for hierarchy visualization, we are not building full data analytics or reporting dashboards. For example, we won't generate burndown charts or productivity metrics in this phase. Our priority is context management and prompt assistance.
- **Multi-Org or Commercial SaaS Features:** We are not packaging this as a product for multiple organizations yet. So, things like multi-tenant databases, usage billing, or an admin portal to manage multiple client organizations are not considered. Our design can accommodate multiple clients within one org's data, but that's different from multiple totally separate orgs.
- **Heavy Parallel Usage or Huge Scale Performance:** Our design will handle the expected load of our team, but we are not specifically optimizing for thousands of simultaneous users or extremely large document sets. If the organization grows the usage significantly, we might need to revisit scaling (like sharding the vector index or optimizing database queries), but initially, we assume a manageable volume of data (maybe a few gigabytes of text at most, which is still large but not massive in terms of what modern search can handle).
- **Billing and Cost Tracking:** Aside from being mindful of cost in choosing technologies, we are not building billing systems or cost-tracking logic. For example, if we use OpenAI API calls, we'll monitor costs manually rather than building an internal chargeback system.
- **Fully Autonomous Agents without Oversight:** While bots assist and automate many steps, we plan for **human-in-the-loop** at key points (especially when moving information upward or making decisions with external visibility). We are not aiming to let the AI make major decisions or publish content without human review. The AI's role is assistive: summarizing, retrieving, suggesting. Humans will confirm promotions of content or final prompt wording for external outputs.

By establishing these scope boundaries, we ensure that our design remains focused on the core problem: managing and using context effectively in a Discord-driven workflow. Next, we define key terms in the system (Glossary) and then address specific research questions and design solutions for each part of the problem.

---

2. Glossary of Core Concepts

To avoid ambiguity, here are definitions of key terms and concepts used in this system design:

- **Hierarchy:** The structured levels of work: **Client** → **Product** → **Project** → **Subproject** → **Task**. This defines parent-child relationships (e.g., each Product belongs to a Client, each Project to a Product, etc.) and scopes of context. Higher levels encapsulate broader scopes (more abstract information), while lower levels are more specific (detailed information).
- **Project Segments:** Pre-defined categories of information **within a Project**. Examples include *Operations*, *Research*, *Timeline*, *Design*, etc. Each segment acts like a sub-folder or tag for project content, so that one can retrieve specific kinds of info (like all research findings for the project). All Projects share a common set of segment types (which can be extended if needed). They are not separate projects, but organizational categories for content in a project's context vault.
- **Dump Bot:** The bot (or process) responsible for **intake of raw information**. It monitors designated Discord channel(s) where team members "dump" files and text. The Dump Bot reads file metadata (frontmatter in Markdown) or uses other cues to classify the content's level (client/product/project/etc.), segment, and other attributes. It then routes the content to the appropriate storage (context vault) and may send notifications (e.g., confirming ingestion or alerting relevant project bots of new content). The Dump Bot essentially is the gatekeeper that transforms unstructured inputs into organized knowledge.
- **Context Vault:** A logical repository for all content and knowledge related to a specific entity in the hierarchy (or a person, or the global insights library). It can be thought of as a folder or database partition where everything about that project or subproject, etc., is stored. Vaults hold not just original documents but also processed forms (summaries, embeddings, metadata indexes). They provide an API or mechanism to query the content within that scope. For example, the Project vault for Project X contains all docs and summaries for Project X, and the Project Bot will retrieve information from that vault when answering questions or generating prompts.
- **Nuance vs. Basis:** Two ends of a spectrum for information detail:
  - **Nuance:** Highly detailed, contextual, and sometimes tentative information – including edge cases, alternative ideas considered, reasons behind decisions, and any granular discussion points. Nuance is valuable for understanding context

deeply but can be overwhelming or unnecessary at higher abstraction levels. In our system, nuance is largely kept at the Task and Subproject level. For example, a task brainstorming session transcript is full of nuance.
    - **Basis:** Solidified, confirmed facts or decisions, and high-level information that forms the foundation of knowledge moving upward. Basis information is what gets passed upward in the hierarchy. It’s the distilled essence without the debate. For example, the chosen design direction after brainstorming, or final requirements, are basis. At the Product or Client level, almost all context should be basis (key results, final decisions, metrics) with minimal or no low-level nuance.
- **Active vs. Passive Context:** This refers to whether information is **automatically surfaced (active)** or only retrieved on demand (passive) in a given context.
    - **Active Context:** Information that the system/bot will proactively provide because it’s deemed immediately relevant. For instance, when starting a new Task, the Task Bot might automatically post the subproject’s goals and any recent decisions (active context essential to the task). Active context is included in prompt generation by default.
    - **Passive Context:** Information that exists and is accessible but is not automatically shown unless a user asks or a specific query triggers it. For example, an older task’s detailed log might not be pushed to every new task (passive), but if a user specifically searches for it or if the bot detects a strong similarity or reference, it can bring it in.
    - The system’s challenge is determining what should be active vs passive at each level. Generally, recent and critical basis info is active, whereas older or highly detailed info is passive.
- **Short-Term Task Context:** The ephemeral working memory of a task while it’s ongoing. This includes the Discord thread messages (conversation among team and Task Bot), any scratch notes or partial outputs generated during the task, and transient knowledge that might not yet be stored permanently. It’s essentially the “working set” of information the task participants are juggling. The Task Bot can help manage this by summarizing as it grows, because this context can become large quickly (e.g., a long brainstorming chat). Short-term context is pruned or summarized once the task ends, to extract the useful bits and let the rest fade or be archived.
- **Finalized Task Artifact:** The end product of a task’s efforts, in a concise form. It could be a written summary, a design file link, a piece of code, a decision record, or any deliverable that represents the outcome of the task. Crucially, it should be **distilled** – capturing the conclusions or contributions of the task, not all the deliberation. This artifact is marked as “final” (even if the task was exploratory, there might be a note like “No conclusive result was reached on X” – that note itself is the final artifact indicating that the exploration ended without a decision). Finalized artifacts are what get **archived upward** to inform the subproject and beyond.
- **Upward Synthesis:** The process of taking content from a lower level and **abstracting/summarizing** it for inclusion at a higher level. This can involve merging

multiple inputs and generalizing. For example, if three tasks under a subproject each produced a result, the subproject bot (or human lead) will synthesize these into a single update: what was accomplished, what was learned, what decisions made, etc., for the subproject as a whole. That subproject summary then undergoes further synthesis when updating the project (perhaps combining with other subproject summaries). Upward synthesis typically **reduces nuance** (you won't include every detail from tasks, just the key outcomes and perhaps a note on any major issues encountered) and ensures that the context at the higher level is concise and relevant to broader objectives. This process may be assisted by bots (automatic draft summaries) but will likely be reviewed or initiated by humans.

- **Design Insights (Local):** Reflections, lessons, and feedback specific to a particular project or subproject. They often come from retrospectives, design reviews, user testing sessions, or simply epiphanies during the work. For instance, "In Project Apollo, we discovered that first-time users prefer a guided tutorial over a manual". These insights are recorded (in the project's vault, perhaps tagged as insight) so that anyone working on Project Apollo in the future can recall them, and so that similar projects might reference them. Local insights help avoid repeating mistakes and build an internal knowledge base of project-specific wisdom.
- **Design Insights (Global Frameworks):** Overarching principles, guidelines, or frameworks that apply across all projects in the organization. These could include the company's design philosophy, standard UX guidelines, coding standards, accessibility checklists, etc. They exist in a global library (accessible to all bots). Unlike local insights, which are experiential, global frameworks are more like rules or best practices that the organization believes in. They are maintained separately (and likely more static, though they can evolve too). The system uses these to inform tasks so that every solution aligns with the broader guidelines. For example, a global principle might be "Our brand tone is casual and friendly"; if a task is writing UI text, the Task Bot might remind the user of this principle.
- **Person Bot:** An AI assistant dedicated to an individual team member. It operates in that person's private channel or DM. The Person Bot has access to the person's **personal context vault** (their notes, journal, tasks they worked on, etc.) and a subset of the shared context (whatever projects the person is involved in, as permitted). Its functions include:
  - Answering the person's questions by combining personal notes and project knowledge (e.g., "What's the status of my tasks on Project X?" or "Did I ever document the design rationale for feature Y?").
  - Helping the person organize their thoughts or plan work (the person can brainstorm with their bot privately).
  - Facilitating the transfer of personal work to the team space: if the person has a draft or idea in their notes, the bot can help format it and send it to the Dump channel with correct metadata when asked.
  - Logging the person's daily activities or important points (if the user likes to "journal" with the bot, these logs can later be retrieved or summarized). Essentially, the

Person Bot is like a personal assistant that also knows about the user's role in the projects. It respects boundaries – it will not share the user's private notes with others unless explicitly told to, and it only queries project data that the user has access to (ensuring permissions alignment).

With these terms defined, we can now confidently discuss the system design without confusion. Next, we outline the research goals and then dive into specific design questions and proposed solutions.

---

3. Research Goals

The primary goal is to design a system that **streamlines work management and context utilization** in a Discord-based environment. Breaking that down, the system must achieve the following objectives:

- **Reliable Ingestion and Organization of Information:** All materials (documents, transcripts, notes) should be captured and stored without loss, and correctly classified so they can be found later in the relevant context. We aim for a metadata schema and ingestion process that can handle the variety of inputs (with minimal manual intervention) and build a structured knowledge base.
- **Hierarchical Context Maintenance:** Maintain a hierarchy of contexts (from granular to high-level) that remains up-to-date as work progresses. When tasks complete and projects evolve, the context at each level should reflect the latest information appropriate for that level. This includes not just storing data but also abstracting it at each tier.
- **Effective Context Retrieval for Prompting:** Enable the generation of high-quality prompts and answers by providing the right context to the user or AI at the right time. This means implementing retrieval algorithms that can filter and rank information based on relevance, recency, and hierarchical level. Whether using similarity search, keyword search, or rule-based recall, the system should surface what the user needs for a given query or task.
- **Avoiding Information Overload:** Present comprehensive information **without overwhelming** users (or the AI models) with unnecessary details. The system should smartly **filter and summarize** context, especially as it aggregates upward. People at a higher level (or bots acting at a higher level) should not need to read through raw transcripts or long chats unless absolutely needed; they should receive concise updates. Conversely, at the lower level, a user shouldn't have to hunt around multiple places for relevant details – those should be readily available. Balancing depth and brevity is a key goal.
- **Task Lifecycle Management:** Implement a robust process for task contexts: allowing free-form exploration while active, then capturing results and cleaning up after completion. This lifecycle ensures ephemeral info isn't lost if important, but also that once a task is done, its context doesn't clutter active work. The outcome of each task should

seamlessly integrate into the broader project context, creating a cumulative knowledge base over time.

- **Integration of Personal and Shared Knowledge:** Allow individuals to maintain personal notes and have personal AI assistance, while still contributing to shared knowledge when appropriate. This dual system should improve individual productivity (by giving them personalized help and space to think) without siloing valuable information (since they can publish anything relevant). The goal is to harness personal work and align it with team objectives, with careful control by the individual.
- **Incorporation of Design Insights:** Ensure that both project-specific lessons and global best practices are easily accessible to those who need them. The system should **not rely on humans remembering** those insights; instead, it should actively remind or include them in relevant contexts. The goal is a smarter knowledge system that learns from the past and from general principles, improving the quality of outputs (like designs or decisions) by applying that wisdom.
- **Access Control and Confidentiality:** Enforce that sensitive information stays within the appropriate audience. For example, internal cost discussions might be at the project level but should not leak into a client-facing summary generated by the client bot. The system must be aware of such boundaries and either avoid or sanitize content when aggregating or answering queries at different levels. This also includes respecting personal content privacy.
- **Seamless Discord Workflow:** The bots and tools should enhance, not disrupt, the normal workflow on Discord. Users should be able to interact in channels and threads naturally, with bots augmenting the conversation. The design should minimize the need to switch contexts or use out-of-Discord tools (aside from the optional dashboard). Ideally, asking a bot in Discord for information or summarization should be as easy as asking a colleague.
- **Traceability and Explainability:** The system should maintain lineage of how information flows. If a summary is generated, one should be able (at least internally) to trace which inputs (task artifacts, etc.) fed into it. This is important so that if questions arise (like "where did this metric come from in the product report?"), one can drill down to the source (perhaps the project data or a specific task). This fosters trust in the AI outputs and helps with debugging any mistakes (like if a wrong info was propagated upward, we can find its source).
- **Adaptability and Scalability:** While our immediate use is for a single team, the architecture should be adaptable to more projects or slightly different hierarchies. The design should allow adding new segment types easily, or possibly adjusting hierarchy depth (e.g., if in the future we needed one more level like "Milestone" between project and subproject, it should be feasible). Also, as the volume of data grows, the retrieval methods and storage should scale or be replaceable with more powerful solutions without changing the core logic.

In summary, the goal is an integrated system that **captures knowledge, preserves its structure, and delivers it intelligently** when needed, all in the flow of teamwork on

Discord. With these goals in mind, the next sections will explore specific components and challenges, providing detailed recommendations and architecture proposals for each.

---

4. Solutions by Domain

In this section, we address specific aspects of the system design, each corresponding to the domain-focused questions listed in the prompt (4.1 through 4.11). For each domain, we discuss challenges and propose solutions or approaches, often referencing how it aligns with the goals above.

4.1 Ingestion & Metadata

Metadata Schema and Usage: We propose a minimal but expressive **YAML frontmatter schema** for all Markdown documents and transcribed text files ingested. This schema will be placed at the top of each file between --- lines. The preliminary fields (further detailed in section 5.2) include:

- **id:** A unique identifier for the item (could be a UUID or a human-readable composite key).
- **level:** The hierarchy level of the content (one of: client, product, project, subproject, task, person, global-framework).
- **client/product/project/subproject/task/person:** Identifiers linking the content to a specific entity. For example, product: AlphaApp or project: WebsiteRedesign. Only the fields up to the specified level are needed (e.g., a project-level doc would have client, product, and project IDs filled, but not subproject).
- **segment:** For project-level (or subproject-level) documents, which segment it belongs to (operations, research, timeline, design, etc., or a custom tag if needed). If not applicable (e.g., a product-level or client-level doc might not use segments), this can be omitted or set to a default like “general”.
- **status:** The content’s status in its lifecycle (draft, working, final, archived, etc.).
- **nuance_level:** An indicator of how nuanced vs basis the content is (e.g., high, medium, basis). This might be provided by the author or inferred (e.g., transcripts might default to high nuance; an executive summary doc might be basis).
- **active_importance:** A flag for whether this content should be considered **active** context by default or just passive. E.g., active for a key requirements doc that should always be surfaced in context, vs passive for a long meeting transcript that’s stored but not auto-injected.
- **authors:** List of Discord usernames or IDs who authored or contributed.
- **created/updated:** Timestamps for creation and last update.
- **source_channel:** Where it came from (like “dump” or “meeting-transcript” or “personal-space”).

- **meeting_ref:** If this is a meeting transcript, an identifier to tie it to a calendar event or a unique meeting ID.
- **linked_items:** References to other items (by id) that are related (e.g., a task artifact might link to the tasks it summarizes, or a meeting transcript might link to a task if it was a meeting for that task).

Example: See section **5.10 Prototype Data Examples** for an illustration of this metadata on a sample document.

This metadata is designed to be both human-readable/editable (so team members can write it in Obsidian templates, for instance) and machine-parseable.

Routing Logic: When the Dump Bot receives a file, it will:

1. **Parse the frontmatter** (if present). If the file is not Markdown or doesn't have explicit metadata (like a PDF or DOCX), the bot may look for an accompanying metadata file or rely on the context (for instance, the channel it was posted in might imply some fields).
2. **Validate the metadata:** Check required fields and possibly correct obvious errors (for example, if level: project is given but no project: name provided, that's inconsistent; the bot might ask for clarification).
3. **Classify the item's target:** Using the level and identifiers, determine which context vault(s) it belongs to. E.g., a file tagged as level "subproject" with project: WebsiteRedesign and subproject: LoginFeature goes to the WebsiteRedesign project's sub-folder, specifically under the subproject "LoginFeature". If a segment is given, file it under that segment category in that vault.
4. **Handle missing/ambiguous metadata:**
   - If no frontmatter exists, the bot might attempt to infer. For example, if the file was dropped in a channel named "#project-WebsiteRedesign", it likely belongs to that project. If it's in a general dump channel, the bot might scan the text for keywords (like project names or product names that it knows).
   - The bot could also look at the author: if an intern's personal bot forwarded it, maybe it includes a hint or it's likely related to what that intern is working on.
   - In difficult cases, the Dump Bot will post a message tagging the user, like: "I'm not sure how to categorize this document. Could you specify the project or tag it with a metadata header?" This human confirmation ensures nothing gets misfiled.
   - Another heuristic: If the document mentions certain key terms (like a client name or project code), it can be matched with known entities.
   - We will design a small fallback classification function that tries these approaches in order.
5. **Store the content:** Place the file's content into the storage system. This likely involves:
   - Saving the raw file (or extracted text) in a directory or database under the correct hierarchy (e.g., as Markdown in a folder structure like Client/Product/Project/Segment/...).

    - Storing the metadata in an index (perhaps a database table or a search index for metadata) so we can query documents by attributes quickly.
    - (Optional but recommended) Breaking the content into smaller chunks and updating a **vector index** for semantic search. Each chunk would carry the same metadata tags, so we can filter by project or segment later when doing similarity search.
    - If the content is an update of an existing item (detected by some ID or title match), handle versioning: possibly mark the old version as archived or superseded. The metadata might have a field for supersedes or version. We could also incorporate a naming convention, like keep the same id but update the content and move the old content to an archive store with an incremented version number. This part needs clear rules to avoid confusion (see Versioning below).
6. **Acknowledge/Notify:** The Dump Bot should provide feedback. For example, reply in the dump channel "Document X has been classified under Project Y (Research segment) as draft." If relevant, also notify the Project or Subproject bot (which might then alert users in that project's channel like "New research document added that might be of interest."). This keeps everyone informed that new info is available.

Encoding Project Segments: The metadata field segment explicitly covers this. We will define a controlled vocabulary for segments (to avoid typos causing divergent tags). e.g., operations, research, timeline, design, other. We might allow an "other" or custom value in case something doesn't fit, but by using metadata we keep it flexible. Internally, each project vault can have sub-containers or tags for these segments so, say, the Project Bot can restrict a query to one segment (if the user asks specifically for timeline info).

Meeting Transcripts: These are a special case for ingestion. We assume we have a separate process (like recording a meeting and then using Whisper to produce a transcript text or markdown) which then needs to be ingested:

- Ideally, the transcript file should have metadata indicating at least the project or subproject it belongs to, and maybe source_channel: meeting-transcript to flag it.
- If transcripts come without metadata, perhaps they're dropped in a dedicated "transcripts dump" channel and the Dump Bot could infer context by the participants or by keywords (for example, project name might be mentioned in the meeting or calendar info).
- We might encourage users to name the transcript files with a project code or mention it in the first lines so the bot can pick it up.
- Another approach: If the meeting was scheduled in a calendar with a known title (e.g., "Project X design review"), the transcription service could attach that info.
- Once identified, transcripts are stored like any doc. Possibly under a segment like "operations" or "research" depending on the meeting type, or just "meeting-notes" if we create that category.
- **Linking:** The meeting_ref in metadata can store an identifier of the meeting (like an event ID or just a date/time). If actions or tasks came out of the meeting, the transcript file's metadata linked_items could list those tasks' IDs. Conversely, the task metadata

might reference the meeting. The system can use these links to later retrieve “the transcript of the meeting where this task was discussed.”

Handling Versioning & Updates: We need a strategy so that updated information replaces or augments old info without confusion:

- If a document is an updated version (say “Project Plan v2”), ideally the metadata id could remain the same as the original (which was maybe “ProjectPlan”) but then include a version number field or simply use the updated timestamp. We could then mark the previous content as archived or keep it as history.
- Alternatively, generate a new id and use linked_items to point to the old one with a relation like “supersedes: oldID”.
- The metadata status could also reflect something like final (for current) vs deprecated (for old versions).
- Implementation: We might maintain a table mapping a logical document name to the latest version’s ID, or use file naming conventions (e.g., have a folder for the doc and multiple dated versions inside).
- For tasks, versioning is less about documents and more about the content of contexts. A subproject summary might be updated multiple times as tasks complete; each time we could either overwrite the summary file or keep snapshots. It might be wise to keep an archive of previous states (for audit trail), but the Project Bot should use the latest by default.
- Ultimately, we’ll likely have to identify specific content that gets versioned (plans, designs, etc.) and design a simple approach (like always include a version number in the file name and have the bot recognize that and link them, or manually retire old ones).
- **Heuristic:** If a new doc comes in with a title or ID that matches an existing one, assume it’s an update. The bot can then ask “Replace the old version or keep both?” if uncertain.

Metadata Missing/Ambiguous – summary:

- Use channel context or user instructions if available.
- Use content scanning (project names, key terms).
- Possibly use a text classification ML model if we have enough training data on docs, but given moderate scale, heuristic rules plus occasional user queries should suffice.
- Always fail safe by asking a human rather than misfiling in silence.

By establishing a robust metadata-driven ingestion, we set the foundation for everything else. It means later components can trust that, for example, if they query the Project X vault, they won’t accidentally get Project Y’s data, etc. The next section discusses how and where this data is stored.

4.2 Storage Architecture

The storage solution must accommodate different types of data: raw text (from documents and transcripts), structured metadata, embeddings for semantic search, and relationships (hierarchies and links). We also want it to be reliable and queryable by the bots in real-time as they generate prompts or answer questions. Here's our recommended approach:

Hybrid Storage Layers:

1. **File Storage for Raw Content:** Maintain the original files (or their text) in a structured filesystem or object storage (could be simple directories on disk, or something like an S3 bucket with a folder structure). For example:
   - ClientName/ProductName/ProjectName/Segment/filename.md as a path.
   - MeetingTranscripts/ProjectName/2025-07-19_meeting_topic.md for transcripts, etc. This makes it easy for humans to browse if needed and provides a fallback (the raw source of truth). Using a version control system (like a Git repo for all markdown files) is an interesting idea for change tracking; however, concurrency might be tricky if bots also write. Initially, simple directory storage suffices.
2. **Metadata Index (Database):** Use a lightweight database (even SQLite or a small SQL/NoSQL DB) to store the metadata of each item (as parsed from frontmatter). This would be a table or collection where each entry includes fields like id, level, client, product, etc., plus perhaps a pointer to the file location or an internal content reference. This index allows queries like:
   - "Give me all final status documents in Project Y's design segment" or
   - "Find all items linked to Task 123" or
   - "What's the latest summary for Subproject Z?". A relational schema might have separate tables for each level or a single table with all and columns for each possible field (with a lot of nulls). Alternatively, a document-oriented DB (like MongoDB or an embedded one) could store each metadata as a JSON object. Given the moderate scale, even an in-memory index at runtime loaded from parsing all files might work, but a persistent DB simplifies maintenance. This metadata DB also helps enforce uniqueness and handle updates (e.g., if a new doc with id=X comes, you can find if X exists already).
3. **Semantic Vector Index:** Use an embedding-based vector store for semantic similarity search. Each chunk of content will be embedded (using an LLM embedding model) into a high-dimensional vector. We will store these vectors in a vector database keyed by an ID and accompanied by metadata (so we can filter by project, segment, etc. on queries). There are many options: FAISS (in-memory or on-disk), Annoy, or an open-source vector DB like Chroma or Milvus. For simplicity and cost, an in-memory index with periodic saving might be fine to start.
   - **Chunking strategy:** For large documents or transcripts, break into chunks (e.g., 500 tokens or by paragraph) to embed. Each chunk gets an ID (like docID_chunk1) and retains the parent doc's metadata.
   - **Index per level or global?** We could create separate indexes for each major scope (one per project, one per product, etc.) to reduce search space and naturally isolate

    data. This way, if you know you only need to search within Project X, you query its index. However, maintaining many indexes could be overhead. Alternatively, one global index for all content where queries must include metadata filters (like a filter on project) so that vector search is restricted to relevant vectors. Some vector DBs support filtering natively.
    - Given moderate scale, a single index with filtering by metadata should be fine. But for clarity, having per-project or per-product indexes might also mirror the vault concept (like each vault has its own index).
    - **Freshness:** When new content is added, its embeddings should be generated and added to the index promptly. If content is updated, the old embeddings should be removed or flagged, and new ones added.
    - **Semantic search usage:** This enables the bots to do things like: "find me content similar to query Q among those marked relevant to this task's project" easily.
4. **Graph or Relationship Store (Optional):** The hierarchy itself (client-product-project-subproject-task) is naturally a tree. We might not need a graph database explicitly, as we can infer relationships from the metadata (project X belongs to product Y, etc.). However, if complex queries arise (like "show all tasks that two particular people collaborated on" or "find insights applicable to both Project A and B"), a graph could help. For now, that's likely overkill. Instead:
    - We will maintain mappings in memory or via the metadata DB for parent-child relations (like each project record lists its parent product, etc., so one can traverse up/down).
    - Linked items (like tasks linking to transcripts) are a cross-reference we can store in the metadata DB (maybe as a join table or as an array in the JSON).
    - If this gets complex, we could later introduce a graph DB like Neo4j to capture all entities and relationships (projects, tasks, people, concepts) for advanced querying. But initially, the hierarchy is straightforward and can be navigated with the fields we have.

Preserving Linkage between Raw and Derived Content: It's important to tie summaries to their sources:

- We can include in the metadata of an upward summary a linked_items list of the IDs of all tasks or documents that contributed. For example, a subproject summary might list the task IDs it summarized.
- We might also inversely update the source items to link to the summary (like each task artifact could list which subproject summary it was included in).
- The metadata DB, or a separate lineage log, can record these relations.
- This way, if someone is reading a project summary and wants more detail, the system can present the specific tasks behind each point (drill-down).
- Technically, a simple way is: when a bot generates a summary, it knows which docs it pulled in (from its retrieval step). It can then attach those references. If a human writes

the summary manually, they might include references or at least mention tasks; the bot could parse those mentions and populate the linked_items.

Storage Tech Choices:

- We want to keep costs low, so likely:
  - Use the **file system** or a self-hosted Git repo for markdown storage (the latter if version tracking is important).
  - Use **SQLite** for metadata (good enough for moderate concurrency with careful writes, and super simple to deploy). Or if we prefer a more scalable path, a small Postgres instance (especially if we might have complex queries).
  - Use an **in-memory vector index** to start (FAISS) with periodic dumps to disk for persistence, or a lightweight vector DB if one exists that we can self-host for free.
  - All these can run on a single server since load is not huge (but we ensure to design the abstraction such that switching out components is possible when scaling up).

Content Access Patterns:

- Bots will frequently query by project or subproject for recent items, by segments, by similarity to a query, etc. The combination of metadata filtering and vector search (and sometimes keyword search for exact fields) covers these:
  - If a query is specific (contains names or terms we know exist), a direct filter + keyword search via metadata DB might suffice (very precise).
  - If a query or needed context is more abstract (“related to user onboarding experience”), an embedding search is better.
  - Often a hybrid: e.g., Task Bot might filter the index to only subproject=XYZ (so only that context) and then do vector similarity with the task question to pull relevant chunks.
  - The storage must support these quickly. Filtering in memory on, say, a few thousand chunks is fine, and vector similarity on those is also fine at our scale (we can get answers in milliseconds to a second range).

Backup and Reliability:

- We should have a backup strategy for the file content (maybe the Git approach inherently has history; if not, periodically copy the repository or have a sync to a cloud storage).
- The metadata DB should be backed up or be reconstructable by re-parsing all files (we can always re-ingest all markdown to rebuild the index if needed).
- The vector index can be rebuilt from content if lost, though that’s more time-consuming. We’ll likely checkpoint it (save vectors to disk) after any major ingestion.
- Using simple local storage means easier control but we must be mindful of not losing the single server. Offsite backups or cloud storage for critical data might be used in a

lightweight way (for instance, pushing the markdown repo to GitHub or an S3 snapshot daily).

In summary, the storage architecture will use **files for content** (human-friendly and simple), **a database for metadata** (fast lookup by structured queries), and **vector indices for semantics** (intelligent retrieval). This layered approach ensures we can handle different query types and scale pieces independently if needed. Next, we define how content moves through various **states and lifecycles** as it is created and utilized.

4.3 Context Lifecycle & State Management

Content at each level goes through a lifecycle from creation to archival. We will define these states and transitions, as well as triggers that move content upward or mark changes.

Lifecycle States (for content pieces):

- **Draft:** An initial state for new information that hasn't been confirmed or finalized. E.g., a first version of a document uploaded, or notes from a meeting that haven't been processed. Draft content might be incomplete or unverified.
- **Working:** Content currently in use or being actively developed. For example, a design doc that is being iterated on through a subproject's life, or a task that is ongoing. In this state, content is subject to change.
- **Final:** Content that has been finalized or approved. A final task artifact after a task concludes, a final version of requirements, etc. "Final" means it's considered a source of truth and can be relied on for upward summaries (basis info). (Note: final doesn't mean it can never change, just that at that moment it's the settled version.)
- **Archived:** Content that is no longer active or current, but kept for record. This could be a past version of a doc, or content from a completed project that is stored for historical reference. Archived items are generally excluded from active context unless specifically requested.
- **Deprecated:** Similar to archived, but with the nuance that it's not just old, it's actually superseded or invalid. For example, an old requirement that was removed would be marked deprecated. This flag tells bots not to use it at all in new contexts (whereas archived might still be relevant if looking up history).
- (We might not need both Archived and Deprecated, but we include for clarity: Archived = time-based inactive, Deprecated = validity-based inactive.)

Lifecycle within a Task:

- **Open Task (Working):** When a new task thread is started, it's effectively in a working state. The context in that thread is fluid – ideas are being thrown around, partial results made.
- **Closure/Completion (Finalizing):** The task reaches a point of conclusion (decided by the team, indicated maybe by a specific phrase or a command like /task complete or simply an agreement in chat).

- **Final Artifact Creation:** At this point, the task's key outcome is captured. This could be done by:
    - A human writing a summary message or attaching the final output.
    - The Task Bot generating a draft summary and the human editing/approving it.
    - Or even multiple artifacts: e.g., a code file plus a summary. The final artifact(s) get proper metadata (e.g., status: final, nuance_level: basis for the summary).
- **Promotion/Archival:** The final artifact is then moved to the Subproject context (the bot can do this by adding it to the store and linking it). The task thread itself can be archived in Discord (closed or moved out of sight). If needed, the chat log could be saved as an archived transcript with nuance (for future reference, but not actively used).
- **State update:** Mark the task as completed (maybe via metadata field or a task tracker). If using Discord thread status, we simply stop using it.

Lifecycle within a Subproject:

- A subproject might be considered "active" when it's the current focus. The context (subproject vault) is in a working state – tasks are adding content to it.
- When the subproject achieves its goal or is put on hold, we do an **Upward Synthesis** for the project:
    - The Subproject Bot (or project lead) will produce a subproject summary (final artifact of subproject) which abstracts all tasks outcomes.
    - That summary (final, basis-heavy) goes into the Project vault (perhaps under the appropriate segment).
    - The subproject context can then be marked archived (or completed). This might involve marking all tasks closed, and maybe not actively retrieving from them unless asked historically.
- Only one subproject is active per project (per the current assumption), so when one finishes and is archived, another can start. If multiple subprojects run parallel, the system can handle it, but we'll assume sequential for simplicity and to encourage focusing context.

Lifecycle within a Project:

- Projects are broader; they may span multiple subprojects sequentially. The project vault is continuously updated.
- After each subproject, the project's context (especially the segments like Timeline, Research) should be updated. We could accumulate changes or produce a running Project Summary document that is updated incrementally.
- When a project is completed (all subprojects done, or the objective reached):
    - A final **Project Report** or summary is generated, capturing the overall outcome, key metrics, learnings (some of which might actually feed into global insights).
    - The project vault can then be considered closed/archived for active use, but remains for reference.

    - The product context is then updated with this project’s outcomes (especially if it changes the product’s features or state).
- If a project is long-running (no clear “end” but an ongoing effort), we might do periodic syntheses (e.g., monthly summaries) that roll up to product.

Triggers for Upward Synthesis:

- **Manual Trigger:** Most reliably, a human can trigger it by command or by convention. E.g., after finishing a task, the intern might type “/finalize task” which prompts the bot to compile context and ask for confirmation of the summary to store. Or a project manager might say “@ProjectBot summarize subproject X now that it’s done.”
- **Automated Suggestion:** The system can detect conditions:
    - If a Discord thread is marked resolved or goes inactive for a certain number of days, Task Bot can ping: “It looks like Task 123 is no longer active. Would you like me to generate a summary and close it out?”
    - When a subproject channel hasn’t seen new tasks in a while or someone marked a subproject as finished, the bot could prompt to summarize for the project.
    - We likely want human sign-off, so the bot would produce a draft summary and let a user approve/edit it before it’s officially stored and propagated.
- **Completion Events:** If integrated with project management (like if we mark a task done in a task list, that event could trigger the summarization automatically too, but since everything is in Discord, we’ll rely on chat/channel usage patterns).

Transformation Rules for Abstraction:

- We need consistent rules on *what* gets kept when summarizing up:
    - Remove or compress details that are not needed for higher-level understanding. For example, exact numbers from a test can be replaced with just stating the result (“improved by 5%” might become “improved significantly” at a very high level, or might remain if relevant).
    - Preserve key decisions, outcomes, and any context that higher-ups would need to know (e.g., “We chose technology X over Y for reason Z” is crucial to carry upward, maybe in brief).
    - Tag or label anything that was left out so that it can be retrieved if needed. E.g., if a subproject summary omits some nuance, one might footnote that “details in Task 7, 8, 9”. In a digital sense, we have the links.
    - Use metadata to mark in the summary what level of nuance was removed (“nuance_level: basis” indicates it’s already stripped down).
    - Possibly maintain two versions of some content: one detailed, one abstract. But that could be redundant. Instead, rely on drill-down on demand: the summary can be annotated with references to deeper content which stays in lower vault.
- The system might utilize AI summarization for this, but guided by templates to ensure certain info is kept. For instance, a subproject summary template:

    - Goals of subproject,
    - Tasks completed and their outcomes,
    - Decisions made,
    - Outstanding issues (if any moved out of scope),
    - Impact on project metrics. The bot can fill that using the task artifacts as input.

State Machine Concept: We can envision each content piece as a state machine:

- Initially, state is determined by how it enters (e.g., a doc uploaded might start as draft if not explicitly final).
- It can move to working if people actively edit it (maybe not tracked automatically unless we integrate an editor, so might skip).
- It moves to final when some condition is met (maybe an explicit action or an update metadata).
- It moves to archived either by a separate action or when a parent element closes.
- At each state transition, certain actions can fire:
    - Draft -> Final: alert relevant bots that a new final piece is available for context.
    - Task Working -> Task Final (completed): trigger upward inclusion.
    - Subproject Final -> archived: trigger project summary update.
    - Project Final -> archived: trigger product update.
- We should define these triggers clearly so they can be automated or at least checklist for the PM to do.

Example Workflow (tying it together):

- An intern completes Task A in Subproject Alpha. They or the Task Bot marks it done and posts a summary.
- The Subproject Bot sees this (maybe via metadata or because it monitors the thread or is pinged) and adds Task A's artifact to its vault.
- Suppose Task A was the last task needed for Subproject Alpha; the project lead signals that Subproject Alpha is now complete.
- The Subproject Bot (or Project Bot) compiles a Subproject Alpha Summary (using Task A, B, C artifacts) and posts it in the Project channel (and stores it in Project vault).
- The Project Bot updates the project's "Timeline" segment maybe (like marking a milestone done).
- Now the Project might start Subproject Beta. Subproject Alpha's channel is archived (read-only), and a new subproject channel opens with a fresh context.
- At a much later point, once all subprojects done, the Project is completed and an overall Project summary is created, which is then used by the Product Bot to update the product's context, and so on.

This clear lifecycle management ensures context is neither static nor chaotic; it evolves systematically. Next, we will look at how the system decides **what context to retrieve at**

each level for answering questions or generating prompts, which heavily relies on the hierarchy and state we established.

4.4 Retrieval & Context Selection by Level

Different levels have different information needs, so our retrieval strategy must adapt to each. We outline what context each bot (level) should gather by default (active context) and what remains available on demand (passive). The guiding principle is **relevance and abstraction**: higher levels should automatically get only distilled info, whereas lower levels can handle more detailed context.

Let's break down each level's context retrieval:

- **Task Level (Task Bot):**
    - **Context Needs:** The task needs *immediate, detailed information* relevant to the specific work. This includes any requirements or constraints relevant to that task, recent outcomes from other tasks in the same subproject (so as not to duplicate work or to build on them), and any relevant higher-level direction (like the subproject goal, project objective, or product standard that applies).
    - **Active Context (auto provided):**
        - The subproject's summary or description (what are we trying to achieve in this subproject?).
        - Key points from the project context that apply (e.g., if the project has a requirement or a design principle that this task addresses).
        - The product's basis info that might constrain the task (like "the product's target user is X, so keep that in mind").
        - Any **recent sibling task outputs** within the same subproject, especially if they are logically connected (e.g., Task A's result is needed for Task B).
        - Local design insights from the project if relevant (e.g., "we learned last time that doing Y is problematic").
        - Global design frameworks that match the domain of the task (the system might check the task description against tags in the global insights to pull maybe one or two relevant principles).
        - The task's own thread history (to maintain continuity in conversation, though the Discord client will show some history, the bot might still have an internal memory or summary of what's been discussed so far).
    - **Passive Context (on-demand):**
        - Full transcripts or documents that are too large to inject but can be searched if needed (e.g., a user can ask, "Find the part of the requirements doc that mentions X" and the bot will search the project requirements doc).
        - Older task outputs from the project that are not directly related to this subproject (maybe something similar was done in another subproject months

    - ago; the bot might not surface it automatically but can retrieve if asked or if the user says “has this been done before?”).
    - Very fine-grained nuance from earlier in the subproject that isn’t immediately needed unless queried.
  - **Retrieval Approach:** Likely uses a combination of **metadata filtering + vector similarity**. For example, the Task Bot, on being invoked, will filter the vector index to project = X and (subproject = Y or level = project with important tag or level = product with basis) and then do a similarity search with a query that represents the task’s context (the user’s query or the task’s problem statement). It may also have a curated small set of context (like subproject summary, etc.) that it always includes from metadata DB.
- **Subproject Level (Subproject Bot):**
  - **Context Needs:** The subproject bot deals with a collection of tasks under a theme. It needs to keep track of what tasks have delivered, what is in progress, and outstanding questions or issues. It also needs to understand how this subproject aligns with the larger project and product.
  - **Active Context:**
    - **Task artifacts** within this subproject: The distilled outputs of any completed tasks (these essentially make up the subproject’s current knowledge).
    - If tasks are ongoing, perhaps a brief status of each (but that’s more for a PM view; for generating answers, likely just final artifacts).
    - The project-level direction relevant to this subproject (e.g., the project’s goal and any specific instructions for this subproject).
    - Applicable product constraints or standards for this kind of subproject.
    - Possibly any high-priority items from the project’s other segments (like if timeline or ops segment has something like “Subproject must finish by Q3” or “Budget limit X for this feature” – those should be considered).
    - Local design insights if they exist for this subproject (though usually insights come after, but maybe from similar past subprojects).
  - **Passive Context:**
    - The raw task discussions (if needed for detail, the subproject bot could go into a specific task log, but normally it wouldn’t unless asked to “show how we arrived at this decision”).
    - Other subprojects’ info (the subproject bot normally doesn’t look at siblings unless asked or if something is very related).
    - Detailed project documentation that isn’t directly about this subproject (like research done for another part of the project).
  - **Retrieval Approach:** The Subproject Bot likely has fewer things to sift through than a Task Bot (since it mainly looks at summary artifacts from tasks). It may rely more on structured queries: e.g., fetch all task outputs from its subproject vault (metadata query) and then maybe vector rank them if answering a question. For project info, it might directly retrieve certain fields (like project goal). So a mix of direct lookup (for

known context like subproject description, project constraints) and vector search (for unstructured content from tasks).

- **Project Level (Project Bot):**
  - **Context Needs:** The project bot focuses on the high-level status and knowledge of the project, across all its subprojects and segments. It cares about aggregated results, project-wide decisions, and how the project contributes to the product.
  - **Active Context:**
    - **Subproject summaries:** each subproject (especially the active or recently completed ones) should have a distilled summary or key results which the project bot will primarily use. This gives a snapshot of each component of the project.
    - **Project segments documents:** E.g., the project's "Timeline" (milestones, deadlines), "Operations" (maybe budget or resource info), etc. If the user asks a timeline question, it will specifically use that segment.
    - **Product basis:** The project bot needs to frame answers in terms of the product's goals. If the product context says "Product Alpha must remain compatible with legacy system Y", the project bot should always keep that in mind if relevant to the project. So product-level requirements and any interface with other projects under the same product might be included.
    - **Major decisions and changes:** If any subproject resulted in a decision that impacts the project overall (e.g., "We pivoted the target user group"), that should be surfaced.
    - **Recently updated global or client directives:** If the client (top level) or company gave a directive that affects all projects (like a new privacy rule), the project bot should include that context as active.
  - **Passive Context:**
    - Detailed task-level info (rarely needed at project level unless someone drills down).
    - Internals of subprojects beyond their summaries.
    - Minor insights or small details unless specifically asked (the project bot stays fairly high-level).
    - Old historical data of the project (like if this project spans a year, details from early phases might be archived – not shown unless asked for historical reasons).
  - **Retrieval Approach:** The project bot can probably lean heavily on **structured data and summaries**. It might have a small number of key docs (like one summary per subproject, one timeline doc, etc.) which can even be stored as short text fields that it can concatenate for an answer. For more detailed queries (like "What user research was done for this project?") it might do a vector search specifically in the Research segment files. So it will first filter by segment if the query implies one, or by subproject if the query is about a particular feature, then find relevant content.
- **Product Level (Product Bot):**

    - **Context Needs:** The product bot's scope is the entire product which likely consists of multiple projects (past and present) and the overall state of the product (requirements, roadmap, known issues, etc.).
    - **Active Context:**
        - **High-level product documentation:** This includes the product's current specification, feature list, design principles specific to the product, target metrics, etc. Essentially, what's the product and where it's going.
        - **Key outcomes from each project:** If Project A delivered Feature X and Project B delivered Feature Y, the product bot should know those features and any important results (like "Feature X increased engagement by 20%"). These would come from final project summaries.
        - **Cross-project issues or dependencies:** If there's an insight or issue that spans projects (for example, two projects discovered the same user complaint, meaning it's a product-level problem), that might be highlighted.
        - Possibly **global design frameworks** too, because at product level, you might discuss aligning with global standards (though probably the global frameworks apply everywhere).
        - **Client goals:** If the product is for a client, any specific client requirements or strategic goals should be active context at product level (since this is as high as the work goes before client).
    - **Passive Context:**
        - The details of how each feature was built (project specifics) unless needed. The product bot won't automatically talk about which subproject did what, it will talk about the resulting product capabilities.
        - Implementation details or internal notes from projects.
        - Only on request or if summarizing for internal use would it dip into project-level detail.
    - **Retrieval Approach:** The product bot likely has even more curated information. Possibly a single "Product Brief" or knowledge base that is kept updated by synthesizing project outputs. So it might store those in its vault directly (like each project's final outcomes). For queries, if it's an internal user asking, it might fetch from project vaults if needed, but if it's a client query scenario, it will stick to its curated data (to avoid revealing internal stuff). So retrieval may involve checking the product's vault first (which is likely small and mostly final data). If the question is specific and not answered there, it might then reach into relevant project summaries.
- **Client Level (Client Bot):**
    - **Context Needs:** The client bot deals with the entire client's portfolio (which could be multiple products). It should provide broad overviews, high-level statuses, and ensure nothing sensitive slips out.
    - **Active Context:**

        - **Product summaries:** each product likely has a one-pager or summary of current state, achievements, next steps.
        - **Client priorities:** any information on what the client cares about across products (e.g., key business goals, strategic themes).
        - **Aggregated metrics:** perhaps combined results from projects (like total hours, ROI, etc. if needed for reporting).
        - Possibly **global frameworks** if those are relevant to communicate (though a client might not care about our internal frameworks explicitly).
    - **Passive Context:**
        - Anything too detailed or internal (the client bot should not fetch a subproject detail unless specifically asked and even then might sanitize it).
        - If an internal user interacts with the client bot persona (like testing it), it might have capability to fetch more, but presumably client bot is mostly for delivering info upward/outward.
    - **Retrieval Approach:** Very curated. It might not even use vector search much, because the set of things it can say should be carefully controlled. Likely it will rely on prepared client reports or product summaries. It can assemble an answer from those. If a client asked a detailed question, the bot might either escalate ("I will get back to you on that") or, if it has access, retrieve from product data but then summarize it in a client-friendly manner.

Algorithmic Nuance Reduction Up the Chain: We need to formalize how we reduce nuance:

- We already built in a lot of this in the design by storing summarized forms at higher levels. Algorithmically, when generating those summaries, we can:
    - Filter out content marked nuance_level: high unless a summary specifically calls for an explanation (maybe include only if it was the reason for a decision).
    - Use summarization models that aim for certain target lengths (force abstraction).
    - Use **tags**: e.g., in metadata, if some info is "for internal use only", label it. The upward summary generator can exclude anything with that tag when producing a client or product-level summary.
    - Possibly assign a "confidence" or "stability" score to info: if something is still being debated (nuance), don't include it in a final summary except as a note of risk if needed.
    - At the code/algorithm level, we can implement rules like:
        - When summarizing tasks to subproject: include reasons if important, but mostly decisions and results.
        - When summarizing subproject to project: include what changed in project outcomes, not how team came to it.
        - And so on.

Active vs Passive Surfacing:

- We have to decide which things to auto-load vs keep discoverable:
  - Active surfacing likely is implemented by the bots themselves posting messages with context when a new task starts or a user asks a question.
  - E.g., on a new task thread, Task Bot could post: "**Context for Task**: Goal of Subproject Y is …, Last task outcome was …, Key product constraint: …".
  - Passive discoverable means a user has to ask for it, or manually use a command like @TaskBot search "keyword".
  - Also the UI (like the optional GUI or pinned messages) can hold some passive context (like pinned subproject summary in the channel).
- The division can be based on importance: perhaps mark some docs as "active_recommendation: yes" in metadata to always push them. For instance, a project's core requirement doc might always be suggested at task start.
- For transcripts or lengthy docs, leave them passive but let users know they exist ("There is a 30-page research doc available; ask if you need details on [topic]").

We will crystallize these guidelines further in section **5.4 Retrieval Strategy Matrix**, which tabulates the context sources and nuance levels per hierarchy level.

4.5 Retrieval Methods to Compare

The question arises: how should the bot retrieve information? There are a few paradigms: RAG (Retrieval-Augmented Generation using vector similarity), ReAct (an agent that uses tools and reasoning), or a hybrid. Let's evaluate each and see which fits where:

- **Retrieval-Augmented Generation (RAG) with Vector Similarity:**
  - *Description:* The system encodes all documents into vector embeddings. When a query or context need arises, it turns the query (or a description of current task) into an embedding and finds similar documents or passages. Those are then fed into the prompt for the LLM to generate an answer.
  - *Pros:*
    - Great for unstructured, large text retrieval.
    - Doesn't require the LLM to know about knowledge unless it's provided, so it can be updated easily (just update the index).
    - Scales to large knowledge bases as long as index can handle it.
  - *Cons:*
    - Might retrieve things that are lexically or semantically similar but not contextually appropriate (which is why metadata filtering is important to narrow scope).
    - Can miss relevant info if not similar in wording (though a good embedding helps).

        - Requires chunking which can break context (like splitting documents might lose some structure).
    - *Best Use:*
        - Task-level when you have lots of possibly relevant info scattered around (like many past chat logs or documents).
        - Searching within large transcripts or research documents for relevant sections.
        - Anytime a user query is detailed and we want exact reference text (the model can then quote or summarize it).
        - In our system, RAG is excellent for **Tasks and possibly Subprojects**, because at those levels one might need to pull any detail from the vault quickly. For example, an intern asks in a task thread, “What did we learn about user onboarding last time?” The Task Bot can vector search the local insight docs or relevant project doc for “user onboarding” concept and bring the answer.
        - Also good for global insights matching: we could vector search global insights with the task description to see if any principle is closely related.
- **ReAct-style Tool-Using Agents:**
    - *Description:* The ReAct framework has the LLM plan steps and call “tools” (like search, calculators, DB queries) in an interactive loop before giving a final answer. Here, tools could be: search in metadata DB, search in vector DB, read file content, etc.
    - *Pros:*
        - More **reasoning** power: the agent can decide, for example, “First, I need to find project X’s summary; then I need to check if any tasks mention Y; then combine.”
        - Can handle complex queries that require multiple steps or combining info from multiple sources logically. (E.g., “Compare the results of Project A and Project B” – an agent could retrieve two summaries and then synthesize.)
        - It’s dynamic: if a straightforward retrieval doesn’t answer, it can dig deeper or try a different approach.
    - *Cons:*
        - More complex to implement and potentially slower (multiple calls).
        - LLM errors in tool use can happen (hallucinating a tool result if not careful, or going off track).
        - Might be overkill for simple queries or when the needed info is directly available via a single search.
    - *Best Use:*
        - Higher-level bots (Project, Product) might benefit from ReAct because queries at those levels might require aggregating multiple pieces. For example, “What is the overall progress and what risks remain?” – an agent could search the timeline, then search risk logs, and then compile.

    - Also for cross-level queries: e.g., if asked “How did the findings from Subproject X influence the product design?” – an agent might retrieve the subproject’s findings and the product’s design document and then analyze them.
    - ReAct could also be used by a single bot to break down tasks. But one has to consider cost (each step may be an LLM call, unless we use a lightweight model for reasoning).
    - Possibly the **Project Bot and Product Bot** could internally use a ReAct chain for complex tasks because they often have to consider structured info (like scanning multiple segments).
    - At Task level, ReAct might be less needed because typically the question is narrow and a single vector search might suffice. But if a task requires reading several docs to answer, the Task Bot could do a mini agent loop too.
- **Hybrid Approach:**
    - Likely the winner for our case: use **metadata filtering + vector retrieval as one tool** among others. For example, a bot first uses metadata to narrow down relevant vault (e.g., choose which sub-index to query), then uses vector search to get content, then possibly uses a reasoning step if it has multiple pieces to combine.
    - Another hybrid idea: Use vector search to get candidate passages, then use the LLM to reason which are actually relevant or how to use them. (This is common in RAG where the LLM itself discards or uses what is retrieved).
    - We could implement a simple chain:
        1. Identify intent and scope of query (maybe by rules or a classifier: e.g., “user is asking about timeline, so use timeline segment”).
        2. Retrieve relevant pieces (vector search within that scope).
        3. Feed into LLM to get answer.

        That’s a limited form of ReAct (just one step retrieval).
    - Or full ReAct for multi-step queries.

Where Each Method Fits Best:

- **Task Bots:** Should rely heavily on **RAG with semantic search**, because tasks will have lots of contextual text around. Also, tasks often ask for specifics (like “what does document X say about Y?”) – vector search is great for that. The Task Bot doesn’t need complex multi-hop reasoning usually; it just needs to fetch relevant info and maybe do minor reasoning (which the final LLM answer can handle).
    - If a task is extremely complex (like an analysis task that might need to gather data from various places), a ReAct approach could be triggered, but that might be an advanced scenario.
    - Possibly each Task Bot could have a built-in set of “tools”: search subproject docs, search global insights, search codebase (if applicable) etc. It could use a ReAct

    approach to decide which to query first. But given the domain, the number of sources is small enough that parallel or sequential fixed retrieval might be fine.

- **Project Bots:** Might need a bit more **planning** because they have to consider multiple segments or subprojects.
    - E.g., if asked “Are we on track?” the Project Bot might need data from timeline + recent subproject statuses. It could in one prompt try to gather all, but ReAct could let it fetch timeline info, then fetch any delays/issues noted in ops or tasks, then answer.
    - If the query is straightforward (“Give me a summary of project X”), no need for complex reasoning, just retrieve the summary doc. So the bot should be smart enough: if an answer likely exists in one piece, just retrieve that (metadata lookup). If not, consider multiple retrievals with reasoning.
    - We can implement a **decision tree** or simple rules for the Project Bot: known question types map to certain retrieval strategies.
- **Product and Client Bots:** Likely keep it simple and safe. They probably have canned answers or straightforward compilations. They might have a curated knowledge base small enough to not require heavy retrieval logic. However, if asked something like “Which project faced the biggest delay and why?” (client might ask that), then an agent approach could find which project had a delay (maybe search timeline across projects, find the one with largest delay, then find cause from its notes).
    - This is a cross-project query that a ReAct agent could handle nicely. It’s an edge case but shows value.
    - So for advanced analysis queries, a ReAct chain could be invoked by the product or client bot.

Scaling Considerations (chunking, recency, multi-level embedding spaces):

- **Chunking:** As mentioned, break long docs. Use overlapping windows if context needed. Possibly store chunk embeddings plus whole-doc embeddings for coarse search then fine search (HNSW or Hierarchical search).
- **Recency weighting:** In tasks and subprojects, recent info is often more relevant. We can incorporate that by:
    - For example, boosting embeddings of recent content (some vector DBs allow custom distance metrics or re-ranking by recency).
    - Or simply by filtering: maybe by default only search content from the last N days for a task unless user asks historically.
    - We can store a timestamp in metadata and after vector retrieval, filter out or downrank older stuff (or the bots can mention “there is an older item from 1 year ago, not sure if relevant”).
- **Per-level embedding spaces or combined:**
    - We could maintain separate embedding models or spaces for different levels. But likely one embedding model is fine. However, the nature of text at each level might

    differ (task chats vs polished reports), but a general model should handle both.
  - More importantly, separate indexes by context scope (as mentioned, e.g., per project vs global) to reduce search noise. We definitely want to avoid a task accidentally pulling something from an unrelated project just because of a similar word – metadata filter is crucial.
  - If a global index is used, it must strictly filter by relevant context (like project_id in query).
  - Possibly maintain one index for global insights that is separate (so when looking for design principles, it searches that index).
  - People's personal notes could also be another index if we allow semantic search through personal stuff.

In summary, we plan a **hybrid retrieval approach**:

- **Primary method:** Metadata-driven filtering + Vector similarity (RAG).
- **Supplemented by**: Deterministic queries (for specific known items) and possibly multi-step reasoning for complex queries at higher levels.
- We will use whichever method is most efficient for the given level/query type, balancing complexity and performance.

Next, we focus on the **Task-level context management** more deeply, since tasks are the front lines where interns and bots interact heavily.

4.6 Task-Level Context Management

Tasks are dynamic and can produce a lot of intermediate information. Managing this properly is vital to avoid confusion and to capture useful outputs. Let's detail how tasks will be handled:

Task Thread as Workspace: Each task is a Discord thread under a subproject channel. This thread serves as a mini workspace where:

- Team members and the Task Bot discuss, share ideas, and perhaps drop small files or code snippets.
- The Task Bot might create ephemeral summaries to keep context fresh (for instance, if the conversation exceeds a certain length, the bot can offer a TL;DR so far).
- There might be checklists or sub-tasks enumerated by the bot or user to organize the effort.

Ephemeral vs Persistent Content:

- During the task, a lot of ephemeral content (messages) is generated. Not all of it will be stored permanently. The goal is to let people explore freely (divergent thinking) without worrying that every half-baked idea goes into the long-term knowledge base.
- Mechanisms to handle the volume:

    - **Summarization Buffer:** The Task Bot can maintain a running summary of the discussion in the background, updating it as new significant points emerge. This could be stored as a hidden or pinned message. This ensures that if the thread becomes long or if someone joins mid-way, the summary can catch them up. It also is easier to use than raw chat logs for context injection.
    - **Highlights/Tags:** If someone says something particularly important, they might react with an emoji or a command like "!important". The bot can take that as a cue to log that sentence in a separate "key points" list. This list then contributes to the final summary.
    - **Limiting context length:** The bot will likely only keep the last N messages in direct LLM context to avoid token overload (for LLM calls), relying on the summary to bring older context in.
- **Decision Log:** The Task Bot should keep track of any decisions or conclusions made during the task. For instance, if halfway through the task, they decide "We will adopt approach B", the bot can note that. This decision log can be used to ensure the final summary captures it.

Producing the Final Task Artifact:

- When the task is wrapping up, there should be a clear action to produce the final artifact. Options:
    - The intern or lead writes a concluding message that is effectively the final artifact (the bot can then format it as needed).
    - The intern asks the Task Bot "Summarize the results of this task." The bot then generates a summary highlighting the problem, what was done, and the outcome/answer.
    - If there's a tangible deliverable (like a design file link or code snippet), that is shared, and then the bot or person writes a short description (e.g., "Created design mockups for the new login screen, see link. Key decisions: …").
- The final artifact should be relatively short and to the point (maybe a few paragraphs or a list of outcomes). It should reference any critical details or rationale (could include "we considered X but chose Y" if that's important for future readers).
- The Task Bot can help by providing a template or checklist to ensure nothing is missed ("Please confirm the task outcome:\n1. What was the solution or decision?\n2. Any assumptions or next steps? …").

Storing Final Artifact:

- As soon as it's approved, the artifact is added to the **Subproject Vault**. This could be done by having the user or bot post it in the subproject channel or directly injecting into storage:
    - Perhaps the artifact is written in a Markdown code block in the thread, and the bot picks it up, wraps it with YAML frontmatter (level: task, etc., status: final,

    nuance_level: basis) and posts it to the dump channel itself for recording, or even directly calls an API to store it.
  - Alternatively, the team member could copy it into a prepared Markdown file with frontmatter and dump it (but that's more friction; better if the bot automates it).
- The artifact's metadata should link back to the task (e.g., include the task ID or thread ID) so we can trace it to the original discussion if needed.
- If multiple artifacts come out (say, a piece of code and a summary), we might store both separately but link them (the code could be attached or put in a repo; the summary references it).
- The Task Bot could also suggest updating the subproject summary if applicable ("Should I update the subproject summary with this result?" sometimes might not be needed until subproject end, but for critical info maybe yes).

Using Finalized Artifacts in Future Tasks:

- Once an artifact is in the subproject vault, any future Task Bot in the same subproject can access it as part of context (likely as active context if recent).
- For instance, if Task 5 happens after Task 3 was done, the result from Task 3's artifact can be proactively surfaced by the bot if relevant ("Recall: In Task 3, we decided X.").
- This is the whole reason for capturing these artifacts, to avoid repeating work and ensure continuity.

If Task Yields No Concrete Output:

- Sometimes tasks are exploratory and end with "no, this path doesn't work" or "we need to revisit later". In such cases, it's still important to document that outcome (negative results are results too).
- The final artifact could be a brief note like "Investigated A, B, C; none are viable due to D. Recommend not pursuing further until condition E changes." This is basis info (the conclusion: not feasible).
- That way, if later someone has the same idea, they find that it was tried and why it failed (saving time).
- So every task, success or failure, should leave some trace in the vault.

Integration with Task Management (if any):

- We might not have an explicit task management beyond Discord threads. But if we did, marking a task done in a project management tool should trigger the same process. Currently, we rely on Discord usage and bot commands.

Communication to Team:

- When a task artifact is posted to subproject, the Subproject Bot can announce in the subproject channel: "Task X completed. Summary: … [maybe truncated]. See vault for details." This keeps everyone in the subproject loop without reading the full thread.

Cleaning Up Task Context:

- After finalization, the thread can be closed or at least no longer actively monitored by the bot.
- We might keep the thread for a short period for any follow-up questions then archive (Discord allows archiving threads after inactivity).
- The bot should free any resources (like forget the short-term memory of that thread's details beyond what's in the summary).
- If someone tries to continue in an archived task thread, maybe the bot politely nudges them to start a new task or re-open properly.

Schema for Final Artifact in Storage:

- It will be stored similarly to any doc but marked as level: task or level: subproject depending on how we categorize it (we could say since it's stored in subproject vault, mark it subproject level, but it's really output of a task).
- Perhaps use level: task and have a task: ID in metadata, so it's clear it's a task output not just any subproject doc.
- That way, if we ever list all tasks outcomes for a subproject, we filter by level=task and subproject=XYZ.
- This helps track tasks as discrete knowledge items.

Example (for clarity):

- Task: "Design login page UI".
- Divergent context: Discussion of 3 design options in the thread, linking to Figma mocks, etc.
- Final artifact: "**Outcome:** Chose design option B for login page. It met the ease-of-use criteria and aligns with our design system. Option A was rejected due to accessibility issues; Option C was too time-consuming. Next step is to implement this design in code. (Figma link attached)."
- Stored with metadata referencing Project=WebsiteRedesign, Subproject=LoginFeature, maybe segment=Design, status=final, nuance_level=basis.
- Future tasks in LoginFeature subproject, if about implementing, will get context "Design option B was chosen (with rationale)" included by bot.

Having defined task management, we proceed to specify the roles and behaviors of each type of bot in the system, as they orchestrate these processes.

4.7 Bot Roles & Behaviors

We deploy bots at multiple levels, each with a defined role, scope of knowledge, and behaviors. Here we describe each bot type, their responsibilities, interactions, and how they function:

General Bot Characteristics:

- All bots are instances of essentially the same backend AI agent, but configured with different context scopes and “personalities” (perhaps via system prompts that tell them their role).
- They should all follow organizational guidelines (e.g., style, not revealing info beyond scope).
- They communicate in the Discord channels/threads of their scope.

Now, bot by bot:

- **Client Bot:**
  - **Scope/Purpose:** Represents the top-level client context. It is used to provide high-level summaries and answer questions about the entire client’s portfolio of work. It might be used by senior leadership or for generating client update reports.
  - **Input Sources:** It has access to *product vaults* for all products under that client, but primarily uses the curated product summaries and any client-specific documents (like contract, client briefs, overarching goals).
  - **Outputs/Actions:** It answers queries in the client channel (if one exists, possibly a private channel for internal team to simulate what to tell client). It can generate client-ready summary reports on demand. If we integrate with output to actual clients, it must be careful to only output authorized info. It might also push downward any client-level directives (e.g., if client changes a requirement, the client bot could inject that info down to product or project bots).
  - **Special Behaviors:** Likely heavily moderated; might escalate complex questions to a human (or ask a Project Bot for details then sanitize answer).
  - **Example:** If asked “What’s the overall status of our project with ACME Corp?”, the Client Bot collects the product snapshot and key project updates and answers with a concise summary.
  - **Memory Boundaries:** Very small memory of its own conversation (just high-level chat). It largely relies on retrieval from product contexts. It should not retain or reveal any subproject-level detail in its model memory, to avoid blurting out something not meant for client level.
- **Product Bot:**
  - **Scope/Purpose:** Focuses on a single product. It knows about all projects under that product and the state of the product itself. It is the go-to for questions like “How is Product X doing? What features does it have? When are releases?” It ensures consistency across projects in that product.
  - **Input Sources:** Has access to *project vaults* (final outputs and key docs from each project under the product). Also the product’s own docs (roadmap, vision, technical standards, etc.). It might also reference global frameworks if the product has adopted some (like design language guidelines).

    - **Outputs/Actions:** Answers in the product channel, providing cross-project info. It can summarize the product status for management, or detail how different projects contribute to product goals. It may also coordinate between projects (e.g., if two projects overlap, the product bot can highlight that).
    - **Escalation/Elevation:** If a question is very specific to a project, the product bot might defer to the project bot or incorporate project-level retrieval. Conversely, if the product bot sees a pattern or conflict between projects, it might raise that to a higher level or notify those projects.
    - **Memory Boundaries:** Keeps a short history of product channel conversations. It primarily fetches from static product info and updated project results. Should avoid diving into raw task data. If needed, it pulls from a project summary rather than tasks.
    - **Example Behavior:** If one project finishes a feature, the product bot might broadcast in the product channel “Feature X completed in Project Y, integrating into Product roadmap…”.
- **Project Bot:**
    - **Scope/Purpose:** Oversees a specific project. It has the most holistic view of that project including all subprojects and segments (ops, research, etc.). It helps project managers or team leads to get information, and can update product bot with changes.
    - **Input Sources:** *Subproject vaults* (especially their summaries and any key docs from tasks), all *project segment docs* (plans, research findings, etc.), and relevant *product context* (requirements or constraints that guide the project).
    - **Outputs/Actions:**
        - Answers questions in the project channel (e.g., “What did we find out in our user research?” or “Show me the timeline status.”).
        - Provides periodic updates or can compile a project status report.
        - When a subproject summary is finalized, the project bot might incorporate that into a project-level doc or announcement.
        - It could suggest starting a new subproject when one finishes (like “All tasks for phase 1 done, should we initiate phase 2?”).
        - It may also detect if some discussion in the project channel should actually be a task and suggest, “This is detailed, perhaps start a task thread for it.”
    - **Escalation/Coordination:** If someone asks a product-level question in the project channel (“How does this affect the overall product?”), the Project Bot might either answer from known product info or ping the Product Bot’s data.
        - Also, if the project bot notices an issue that goes beyond project scope (like a risk that affects the whole product or client), it should alert higher levels (e.g., notify product bot).
    - **Memory Boundaries:** It keeps memory of the recent project channel discussion and recent subproject completions. For older things, it fetches from vault. It might maintain a small state (like current active subproject, current milestone).

    - **Example Behavior:** If asked "What's our progress so far?", the Project Bot might gather all subproject completions (like 2 out of 5 subprojects done) and any delays from timeline, and give a summary. It might say "We are 40% through development, slightly behind schedule on design, key risk is X."
- **Subproject Bot:**
    - **Scope/Purpose:** Manages the context of a specific subproject (which often corresponds to a phase or major feature in a project). It's like the lead for that subproject's knowledge, ensuring all tasks align and sharing info between tasks.
    - **Input Sources:** *Task artifacts* from tasks within the subproject, any *subproject-level notes* (maybe an overview of plan for subproject, open questions list), and the relevant slice of project/product context (the subproject bot might be configured with the particular objectives relevant to that subproject).
    - **Outputs/Actions:**
        - Answers queries in the subproject channel about status or details ("Which tasks are done, what's left?" or "What decisions have we made so far in this subproject?").
        - Provides a subproject summary (possibly pinned in channel or on request).
        - When tasks finish, it records their outputs and might update a running subproject summary.
        - It may identify if a new task is needed (like if a question arises in subproject channel, it might suggest spawning a task thread to handle it).
        - It might enforce focus: e.g., if someone tries to discuss two different topics in one subproject channel, maybe they should be separate tasks; the bot can help organize that.
    - **Coordination:** It reports upwards to the Project Bot with subproject completion or major findings. It might occasionally request info from project (like if something is unclear about requirements, it can query project vault).
    - **Lifetime:** A subproject bot might be created when a subproject is started (maybe by making a Discord channel for that subproject) and archived when done. If only one active subproject allowed, then project bot might reuse the subproject bot slot for the next one with new context (but more likely we treat them as separate instances for clarity, maybe multiple channels but highlight active).
    - **Memory:** It will store an aggregated view of all tasks (which is much smaller than raw tasks themselves). It uses retrieval to recall details of any specific task as needed.
    - **Example Behavior:** After each task, the subproject bot updates the channel: "Task X done: (short result). Outstanding questions: Y. Next: plan Task Z for that." Keeping everyone aligned within subproject.
- **Task Bot:**
    - **Scope/Purpose:** Lives in a task thread to assist with that specific task's execution. This is the most interactive bot that collaborates with interns and team on immediate problems.

    - **Input Sources:** Primarily *subproject vault (nuanced data)* for context and any *immediately relevant higher-level info*. Also, it actively watches the conversation, using previous messages as context.
    - **Outputs/Actions:**
        - Provides relevant info on request ("@TaskBot, what did we decide about X in a previous task?").
        - Proactively offers help: if a user seems stuck or discussing something that has known context, the bot might say "FYI, there's a doc that covers this."
        - Summarizes as needed: "Here's a summary of the discussion so far" or synthesizes options discussed.
        - It can fetch answers or data (acting like an intelligent search interface).
        - It helps generate content: e.g., "@TaskBot, draft an email to the client about the feature we just designed" – it should use context to draft appropriately.
        - At task conclusion, it assists in producing the final artifact (as discussed).
    - **Behavior:** The Task Bot should be fairly conversational and helpful. It can ask clarification questions if user request is ambiguous.
    - It should avoid providing irrelevant info to keep focus (so as not to overwhelm with context). Only share what's needed or specifically asked.
    - **Memory:** It has short-term memory of the thread (the last several messages, plus any summary it has). It will rely on retrieval for anything beyond that. So if the conversation goes on for hours, it might not "remember" older parts except through its dynamic summary.
    - **Example Behavior:** If in the task thread someone mentions a design principle, the bot might chip in "According to our global design guidelines, we also should ensure the design is mobile-friendly (just a reminder)." Or if asked "Can you list the key user requirements relevant to this task?" it will pull those from the project docs and list them.
- **Dump Bot (Ingestion bot):**
    - **Scope/Purpose:** Not tied to a single context; it's more of a utility bot that processes incoming files. It might not have a conversational interface beyond acknowledging uploads.
    - **Input Sources:** Discord channels for dumping info, possibly an API feed from a transcription service.
    - **Outputs/Actions:** We described in ingestion: parse files, classify, store them, and notify relevant channels.
    - It might have a command interface too, like "/ingest status" to see if any files are in queue, etc.
    - Also, if someone just pastes a quick note without metadata, the dump bot might allow inline commands like "/classify as subproject=Alpha, segment=research".
    - **Memory:** Not much memory needed; stateless per file, just uses the metadata schema and any cached list of known projects/products to validate against.

- **Person Bot:**
  - **Scope/Purpose:** Private to each person, handling their personal data and queries. It acts more like an assistant than a project bot.
  - **Input Sources:** That person's *personal vault* (notes, journal, tasks assigned to them, etc.), plus a view into the shared context limited by what projects they're in or roles they have.
  - **Outputs/Actions:**
    - Answers personal queries ("What's on my plate today?", "Summarize my notes about Project X from last week.").
    - Helps with personal tasks like drafting messages, brainstorming in private, scheduling (if integrated with calendar).
    - Facilitates sharing: e.g., the user says "Share my note about design ideas with Project Y", the person bot will format that note with frontmatter and send to Dump Bot or directly to the project channel.
    - It might keep a personal to-do or log (the user can tell it "remember that I need to talk to Alice tomorrow about feedback" and it logs it).
  - **Interactions:** Only the user and this bot see the conversation (it's likely a DM or private channel). Possibly the person bot could DM other bots or coordinate if asked (like if user says "ask the Project Bot if there's an update", the person bot might fetch it).
  - **Memory:** It will have ongoing memory of the user's interactions with it (their private conversation), because that forms a personal knowledge base. But it should not leak that anywhere.
  - **Example Behavior:** If an intern forgot what tasks they did last month, they ask their bot, which then finds all tasks that had that intern as author in metadata and lists them with outcomes. Or the intern drafts a prompt in private and asks the bot "Does this cover all context?" and person bot might pull relevant points they missed from shared context (acting like a second check before they post it publicly).

Channel and Instance Spawning:

- When a new subproject starts, likely a new channel is created by the project lead (maybe via a bot command). The system then initiates a Subproject Bot in that channel (with knowledge of parent project and initial scope).
- When a task thread is created under a subproject channel (could be by simply starting a thread in Discord UI), the Subproject Bot or a central controller can instantiate a Task Bot for that thread. Possibly one bot process could handle multiple threads if it can differentiate context by thread ID, but conceptually we treat it as a bot per thread for isolation.
- Implementation might be a single bot user that joins threads and looks up context based on thread, or separate bot accounts (less likely, one bot user is easier).

- So context mapping: the bot process receives a message event with channel or thread ID, and it maps that to a context scope by looking up in metadata DB or configuration (like threadID -> Task under Project X Subproject Y).
- For person bots, maybe each person's bot could just be the main bot user via DM with a mode, or we create individual bot users (e.g., "AliceBot") for fun. The latter might be heavy; simpler is one user like "AssistantBot" that can serve multiple DM contexts distinguished by user.

Escalation and Referral Logic:

- If a bot at one level gets a query outside its scope, it should either fetch from the appropriate source or gently redirect. For example:
  - If in a task thread someone asks a high-level question: "What's the overall project budget?" The Task Bot knows this is project-level, not something it stores. It can either:
    - Use its ability to query the project vault and answer ("Project Bot says: the budget is $X, per ops docs."), or
    - Reply: "That's a project-level question. Let's ask @ProjectBot in the project channel." Possibly easier to just fetch and answer, but depending on security maybe it should encourage going to the right channel.
  - Conversely, if in the project channel someone starts discussing a very detailed design solution, the Project Bot might suggest: "This sounds like it should be a task in Subproject Y. Shall we create a task thread for it?" and maybe even create it on command.
- **Permissions**: Bots need to respect not giving info to channels where it doesn't belong. The above should only happen if the person asking has access, etc.

Memory vs Retrieval:

- None of the bots should rely on long-term memory of the LLM conversation history beyond a session – they should always reconstruct context from the knowledge base. This avoids drift and ensures updated info is always used.
- They might cache the last user query or their last answer to allow follow-up questions to be answered (some small context window).
- But for anything older, use retrieval rather than raw memory to avoid forgetting updates.

This multi-bot setup distributes responsibilities and keeps context management modular. Now, controlling who can see what leads us to **Access Control & Visibility**.

4.8 Access Control & Visibility

With sensitive information at lower levels and broad audiences at higher levels (like possibly clients at top), we need rules to ensure data is appropriately siloed or sanitized. We rely on both **Discord's channel permissions** and our own metadata-based filtering/redaction.

Discord Native Permissions:

- We will structure Discord roles and channels such that:
  - Interns and team members have access to internal project and subproject channels.
  - Clients (if they are ever invited to the server) might only see a high-level channel where the Client Bot posts updates (and maybe a Q&A channel for them). They would not see raw project channels or dump channels.
  - Certain sensitive subprojects could be in private channels only certain team roles can see.
  - Person bot DMs are private by nature.
- This provides a baseline security: even if a bot accidentally tried to post something in a channel, if the channel is locked to certain roles, it might not matter because the wrong audience isn't present. However, since bots can technically see everything they have access to (likely all internal data), we need them to intentionally filter.

Metadata-Driven Permissions:

- Each content item has a permissions field (with values like default, restricted, inherit, etc.). We define usage:
  - default might mean visible to anyone who has access to that level/channel normally.
  - restricted could mean it's sensitive; only a subset should see it. For example, maybe a certain financial doc is marked restricted and only project leads have access. The bot should know not to share it with interns or include it in a general summary.
  - inherit might mean it inherits permissions from its parent context. For example, a subproject doc might inherit the project's default unless overwritten.
  - We may define more finely: e.g., confidential-client meaning "do not share outside internal team".
- When retrieving or summarizing, bots will check these flags:
  - If producing a client-level summary, they will exclude any content marked internal-only. Perhaps even within internal team, some info might not be for interns, etc., depending on roles.
- We can manage this by tagging content and perhaps tagging user roles. The bot can be aware of who asked (Discord provides user ID and we can map to a role) and then filter accordingly.
  - Example: a piece of context says permissions: restricted-finance and only the PM and finance lead roles are allowed; if an intern's bot query tries to access it, the bot should either skip it or heavily abstract it (like "budget info exists but cannot be shown").

Bot-Mediated Redaction & Summarization:

- Upward synthesis inherently does some redaction (nuance removal). But beyond that:

    - When the Project Bot summarizes for a product-level update, it should remove team names, internal comments, or minor issues that were resolved (unless it impacts product).
    - If a client is to see a summary, it should remove any mention of internal delays or blame, etc., presenting a cleaned narrative as appropriate.
    - We can create templates for external vs internal reports to enforce that style.
- If a user explicitly asks the client bot something that requires internal info to answer truthfully, the client bot has a choice:
    - Possibly provide a high-level answer and say "I can follow up later on details" (giving time for a human to decide what to share).
    - Or if it's a safe detail, answer but still phrased appropriately.

Data Access Inheritance:

- Generally, someone working at a subproject level can see all info in that subproject, and by extension see summarized info above. But not necessarily all nuance from parallel subprojects unless given access.
- We likely treat all internal team as having access to all internal data (depending on how open the org is). But if needed:
    - We could restrict an intern to only their project's channels. Then they naturally only get that context.
    - If an intern's person bot queries something about another project they are not in, the bot should either not have access or at least warn "You don't have access to that project."
    - This implies linking Discord roles to context permissions: e.g., each project might have a role for members.
    - The metadata could list authors or intended audience (maybe an optional visibility: [roles]).
- Upward aggregation is meant to be safe to share at broader levels, so typically the product and client vaults only contain sanitized info. Thus if someone at product level reads everything in the product vault, it should be fine for them (assuming they're high-level internal or client).
- But downward, a person at product level (like an executive) presumably can see anything under (if they have permission, or at least ask for details).
    - If we had an executive who wants more detail, they could dive, but more likely they ask someone to get it. Our system might not allow a client to directly query a task vault obviously.
- So effectively:
    - **Upward info**: by nature curated for wider consumption.
    - **Downward info**: likely restricted to those doing the work.

Audit Logging:

- Each time content is promoted upward or summarized, we should log who initiated it. E.g., if an intern finalizes a task summary, mark that in metadata (authors field, etc.). If a bot auto-summarized something, maybe list the bot or the reviewer.
- If later there's a discrepancy or leak, you can trace "this client summary included X, which came from Project doc Y, promoted by person Z."
- This could be as simple as adding a line in metadata "promoted_by: @alice on 2025-07-19".
- We can also keep an internal change log: "Task 12 artifact added to Subproject A by BobBot (approved by @alice)".
- For security, bot messages themselves are logged by Discord. But having our own log (even just text file or DB) for knowledge flow is useful.

Visibility of People's Personal Notes:

- Those remain private until shared. A Person Bot should never share someone's note to the team unless instructed by that person. If another user's bot tries to access someone else's personal vault, it should be forbidden (unless maybe an admin override, but likely unnecessary).
- So personal vault has strict per-user access.

Redaction Mechanism:

- If summarizing a document with restricted bits, the bot could replace specifics with "[REDACTED]" or generalize ("certain internal issues were encountered" instead of "developer X was sick, causing a delay").
- Possibly maintain two versions of some summary: internal and external.
- To enforce this systematically, we could mark sections of docs as sensitive via annotations or splitting docs (like have separate doc for "client safe summary" vs "full summary").

Training the Bots on these rules:

- In their system prompt or initial instructions, specify: "If you are a Project Bot summarizing for a higher level, exclude any details marked restricted or unnecessary for that level. If uncertain, ask a human."
- Person bots: "Never reveal personal notes unless user explicitly says to share."

Inter-Project Visibility:

- Usually, all internal team can see all projects. If that's not true, we could have contexts separate (like one product team shouldn't see another product's data). If needed, we can enforce that with roles and ensure bots do not cross that either.
- But our assumption is one org, one team, multiple clients or products but same team, so likely fairly open internally.

In essence, access control is about **not exposing nuance beyond its intended audience**. Our design uses a combination of metadata tags and channel segregation to achieve this, with the bots programmed to obey those constraints.

Now, on how we incorporate the special **Design Insights** knowledge into our retrieval, let's examine that.

4.9 Integrating Design Insights

Design insights are a valuable kind of context that's not tied to just one task or project timeline, but rather serves as wisdom to guide decisions. We have:

- **Local Insights:** specific to a project or subproject.
- **Global Frameworks/Principles:** apply generally.

Storage of Insights:

- **Local Insights:** Could be stored as part of the project vault, perhaps under a segment called "insights" or appended to a project retrospective document. Alternatively, each insight can be its own markdown note with metadata like type: insight and maybe tags for topic.
    - For example, after finishing a project, the team might do a retro and record "Insight: Users in domain X respond better to Y". The Project Bot could prompt them to store that formally.
    - These would have level: project (or subproject if very specific), and maybe a field linking them to context (like origin: Subproject Beta, Task 3).
- **Global Frameworks:** Maintained by leadership or a central team (maybe the design lead). They could live in a separate vault (like a top-level "Design Insights" vault). Or possibly under Client or Product if these frameworks are company-specific vs client-specific. Since they said "system-wide", likely one global library for the whole org.
    - Each global principle might be one note, or grouped by category (usability, visual design, etc.).
    - Include tags/keywords for each (like "usability, onboarding, accessibility" for specific principles).
    - Possibly formatted as concise rules or guidelines.

Retrieval Heuristics for Insights:

- We don't want to flood every task with generic advice (that can get annoying), so we need triggers:
    - **Pattern Match:** If the content of the task (like its description or current conversation) strongly matches the keywords or embedding of an insight, then include it.
        - e.g., If a task is about "improving sign-up flow" and we have a global insight tagged "onboarding", the similarity will be high. The Task Bot can then say

        "According to our UX principles, onboarding flows should be under 3 steps [Global Insight: Onboarding Simplicity]."
    - Use the vector search: represent the question or the task's goal as an embedding and search in the global insights index. If something comes up with high similarity above a threshold, it's likely relevant.
    - Alternatively, maintain a mapping of certain trigger words to insights (e.g., "login, sign-up, authentication" -> suggests "security & simplicity principle").
    - **Local vs Global priority:** Check local insights first (they are directly relevant if any). If none or if the global ones are just as relevant, include global.
    - Possibly limit to at most 1-2 insights surfaced at a time to avoid spamming.
    - The insights injection should feel like helpful reminders, not directives, unless it's a known rule.

When to inject:

- At **task start**, the Task Bot might add any immediately obvious relevant insights. Example: In a design task thread creation, the bot might post, "Remember our design principle: Maintain consistent branding (Global Principle #3)." This could be based on the task name or initial prompt.
- During discussion, if the conversation hits a topic that an insight covers (like someone suggests something contradictory to a known principle), the bot should gently remind: "Note: Past experience (Project Z insight) showed that doing X can cause Y, so consider that."
- Possibly when finalizing: ensure outcomes are evaluated against global principles (the bot could checklist: "Does this solution follow all applicable guidelines? It seems to conflict with principle A a bit; is that acceptable?").

Tagging and Mapping:

- We should tag insights with categories (UX, Eng, Ops, etc.) and also maybe with hierarchy:
    - A local insight might be tagged with the product or domain, so if another project in the same product or client comes up with a similar scenario, we can choose to surface that too. E.g., If two projects are similar domain, an insight from one might help another.
    - Could incorporate that: if retrieving for a project and no local insight, maybe search other projects' insights that share tags or domain.
- Global frameworks likely have broad tags, so we rely on semantic match mostly.

Conflicts between Local and Global:

- It could happen that a project has a local insight that contradicts a global rule (this might spark a revision of the global rule or an exception).

    - For example, global rule says “minimalist interface”, local insight says “our specific users preferred more instructions”.
    - In such case, both are relevant. The Task Bot could present both perspectives: “Note: Generally we do X, but last project Y found that for this audience, doing Z was better.”
    - This is actually valuable to highlight, as it might lead to refining the global principle or at least documenting an exception.
- The system can’t fully decide which to follow, but it ensures the team is aware of the tension. Then humans decide how to proceed (maybe update the global library or mark that insight as context-specific).

Updating Insights:

- If during a task, a new insight is learned (like “hey, this design trick really improved usability, we should remember this”), the team or bot should capture it:
    - The bot could detect a user message like “This is a great learning” or user explicitly says “/record insight: …”.
    - Then save it to the project’s insights and maybe bubble it to an appropriate global category if it’s general.
    - Possibly with a review step by a lead before adding to global library.

Using Insights in Prompts:

- The prompt generation for the LLM can include relevant insight text as part of the context. For instance, provide the actual excerpt of the insight note in the context window for the LLM to consider when generating the solution or response.
- That’s straightforward if we retrieve them like any doc with RAG.

In summary, integrating design insights ensures that **the system leverages prior knowledge and best practices** not just facts from documents. This should improve the quality of solutions by preventing repeat mistakes and aligning with proven strategies.

Now we consider the **person-level bots and personal workspace integration**, bridging private and shared contexts.

4.10 Person-Level Bots & Personal Workspace Integration

Each team member will have a personal AI assistant (Person Bot) and an associated personal knowledge vault. The goal is to assist individuals without cluttering the shared project space until they decide to share something.

Personal Vault Structure:

- A person’s vault may simply be a folder in the system labeled with their name, or a separate Obsidian vault on their machine that syncs to the system. For integration:

    - We could use a plugin or script to watch their Obsidian notes for certain tags or commands (like a tag “#share(ProjectX)” that triggers sending that note to the Dump Bot via an API).
    - Alternatively, if they prefer Discord for notes, they could have a private channel that the Person Bot monitors (like a journal channel). That might be simpler: the intern can DM or post to their bot channel things like “Note: tried X approach, will finalize tomorrow”.
    - Person Bot can also store some data structurally (like tasks assigned, maybe from the metadata DB if tasks have an owner field).
- Metadata for personal notes would include level: person and person: @username. If a personal note is about a particular project, the user might indicate linked_items: [project: XYZ] or just mention the project in text and rely on search.

Person Bot Capabilities:

- **Personal Deliverables Tracking:** The bot knows what tasks or subprojects the person is involved in (could query the metadata DB for any task where authors include that person, or any subproject where that person is assigned).
    - The bot can answer “What are my tasks for this week?” by looking those up, possibly filtering by status.
    - It might even remind: “Task X is due tomorrow” if deadlines are recorded somewhere (like timeline segment).
- **Daily Notes & Journal:** The user can converse with their bot to record daily stand-up info (“Today I worked on Task 5, encountered an issue with API.”). The bot might structure that or at least store it in a dated note.
    - These notes are only for the user unless they choose to share them (for example, for a weekly team update, an intern might have the bot compile their notes into a report).
- **Parked Prompts and Ideas:** If the user has an idea or a half-formed prompt they want to explore, they can do so in private first. The Person Bot can help refine the idea.
    - E.g., an intern tries to come up with an approach to a design, they talk it through with Person Bot, perhaps even have it role-play as a rubber duck or a reviewer to test the idea.
    - Once ready, they can take that refined idea and present it to the team with more confidence (or decide to scrap if it wasn’t good).
- **Linking to Shared Work:** If the user writes something in personal notes and wants to contribute it:
    - They might instruct Person Bot: “Share this note with Project Y’s research segment.”
    - The Person Bot would then either directly post it to the dump channel with appropriate metadata (with the user as author and properly tagged as project Y research), or even directly call an internal function to ingest it.

    - Possibly ask for confirmation: “Are you sure? It will be visible to the team/project.”
- **Privacy Controls:** By default, Person Bot will not share anything unless asked. It might sometimes ask: “This insight you noted seems very relevant to your project. Would you like to share it with the team?” But it should abide if user says no.
    - We could implement a system where personal vault content can be explicitly flagged for sharing or set to auto-share certain things, but likely manual control is best to avoid accidents.
- **Accessing Shared Context:** The Person Bot can retrieve info from projects the person is involved in, just like going through the official bots but with less formality:
    - For example, the user could ask in DM, “What was the conclusion of Task 7 from Project Z?” The Person Bot can fetch that from the project’s data.
    - This is essentially a convenience; it’s the same as them asking the Project Bot, but privately they might feel more comfortable or might want to combine it with their personal notes.
    - However, the Person Bot should enforce permissions: if the user asks about a project they are not assigned to and if that’s not allowed, it should refuse or at least warn. Depending on company openness, maybe all internal projects are visible anyway, but we should respect if certain things are off-limits.
- **Personal Planning:** Possibly integrate with calendar or to-do list if we had, but even without that:
    - The bot can maintain a simple list of the user’s current tasks/goals and daily progress. It can provide a quick recap: “This week you completed 2 tasks (Task A, Task B) and have 1 ongoing (Task C). No meetings today.”
    - If integrated, it could schedule or set reminders (but that might be an integration with Discord reminders or external).

Personal to Shared vs Task to Subproject Promotion Differences:

- For tasks, promotion is expected and routine as part of the project work.
- For personal notes, promotion is optional and initiated by the person.
- When a task promotes content, it’s structured and goes up levels.
- When a person promotes a note, they have to decide where it goes (which project, segment, etc. – the bot can ask or infer if note mentions a project).
- Also, personal notes might need some clean-up or context added when shared:
    - The person bot might help by ensuring necessary metadata or even rewriting first-person perspective notes into a neutral form. For example, personal note: “I think we should do X” – when sharing, maybe it should be “Recommendation: Consider doing X (from [Person])”.
    - Or the bot can just attach the author name.
- The person might also keep things longer-term that never get shared, which is fine. For instance, maybe they keep a personal learning diary not relevant to others.

Security and Boundaries:

- The person should trust that their bot is not leaking info. So the person bot will not talk about the person's private notes in a public setting or to another user.
- If a team lead queries something like "What has intern Alice been working on privately?" we do not allow that through the system. (Unless the organization had that culture, but that's invasive – likely no.)
- Each person's vault could be encrypted or at least separately stored so that other bots can't accidentally access it. But given the AI likely has access to all data logically, we enforce by policy/coding not to cross those lines.

Obsidian Integration (if used):

- If interns prefer writing notes in Obsidian, an approach:
  - They use a template in Obsidian that matches our metadata format for any note they might share. The Person Bot can be linked via an API or plugin to their vault (the user might have to login or allow it).
  - When ready, they tag a note or run a command (like move to a certain folder that syncs to the ingestion channel).
  - The Person Bot or an integration script sees that and sends the file to Discord dump channel or directly to ingestion pipeline.
  - Conversely, Person Bot could push updates from shared space to the personal if needed (like maybe they want a copy of relevant project updates in their vault to annotate personally).
  - But that might not be necessary; they can always query through the bot instead of copying.

Summary: Person bots empower individuals with a personal lens on the project data, and a safe space to think. They improve productivity and integration of personal efforts into the team's workflow. Next, we will touch on a possible **GUI/Visualization layer** that can give an overview of all this structured info.

4.11 GUI / Visualization Layer (Optional)

While Discord is the main interface, a simple external GUI could greatly help in visualizing the hierarchy and monitoring the project. We propose a minimal web-based dashboard with the following features:

Hierarchy Tree View:

- A collapsible tree showing:
  - Clients at top, expand to see Products, expand to see Projects, expand to see Subprojects, and tasks (active tasks might be listed under their subproject).
  - Each node labeled with its name (maybe ID or title) and possibly an icon or status indicator (active, completed).

    - For example:

```
Client: ACME Corp
  - Product: AlphaApp (active)
     - Project: WebsiteRedesign (phase 2 active)
        - Subproject: LoginFeature (active, 3 tasks ongoing)
           - Task 1: Design login UI (done)
           - Task 2: Implement backend (in progress)
           - Task 3: ...
        - Subproject: ProfilePage (archived)
     - Project: MobileAppUpdate (planning)
  - Product: BetaService (on hold)
```

-
- Clicking on any node could open its details (see below).

Active vs Inactive Indication:

- Use color coding or badges. For instance, active subproject highlighted, completed ones greyed out.
- Only one subproject per project might be marked active at a time if we follow that rule.
- Tasks could have status tags (in progress, done, archived).

People and Roles Mapping:

- Possibly a side panel listing team members and which projects/subprojects they are associated with.
- Could show each intern and their active tasks, to help leads see load distribution.
- If clicking a person, see their person bot’s context like their notes or tasks (if one has permission to view or if we design it for leads to see deliverables).
- Might integrate with HR or time tracking, but that’s beyond scope right now.

Content Viewer:

- If a user clicks on a Project or subproject in the GUI, it could show:
    - Summary or description of that node (maybe the latest summary artifact).
    - The list of documents/entries in its vault segmented by category (like list of operations docs, research notes etc., each clickable to view content).
    - Key dates (like creation date, last update).
    - Perhaps metrics like number of tasks done, tasks remaining.
- For a task, clicking might show the final artifact and possibly a link to the Discord thread (if someone wants to read the whole discussion).

- For personal bots, maybe not included in main GUI unless it's the person's own view or an admin view (private data concerns).

Visualization of Relationships:

- Could include a simple Gantt or timeline if data available (like if timeline segment has dates, show them).
- Possibly network graph if showing which tasks connect to which requirements, but that might be too complex for now.

Data Freshness / Last Update:

- Each node in tree might show last updated timestamp. E.g., Project updated 2 days ago, subproject last task completed 1 hour ago, etc.
- If something hasn't been updated in a long time (e.g., a project idle for a month), highlight that (maybe we need to archive it or prompt a summary).
- Also helpful to ensure the upward synthesis happened: if subproject finished but project not updated in a week, the project node might show a warning or stale indicator.

Access Control in GUI:

- The GUI should respect permissions as well; e.g., if a client user was to see it (less likely we'd give them this, but if so), they only see their product and above, not all internal details. More realistically, this GUI is internal.
- Possibly an admin or PM could use it to quickly click through and audit content.

Write-Back or Read-Only:

- Initially, likely **read-only aggregator**: it's a dashboard to view progress and find info quickly.
- Write-back (editing content or triggering bots) could be useful, but might complicate things. However:
  - We could allow some actions like marking a task done or starting a new subproject via the GUI (which would then call the bot backend to do it in Discord).
  - Or editing a project description field that then posts an update in Discord or updates metadata.
  - These can come later; for now, the user can just go to Discord to do actions.

Technology for GUI:

- Could be a simple React app or even an Obsidian published workspace, but likely a custom small web app hooking into the database and Discord API for current statuses.
- If budget is a concern, we can delay GUI; it's nice-to-have, not essential since Discord is primary.

Value of GUI:

- The reason to include a GUI at all is some things are easier seen visually:
  - Understand the whole hierarchy at a glance (Discord UI by itself can be messy if many channels).
  - See progress indicators (like how many tasks done in a subproject vs total).
  - Identify inactive pieces that might need attention.
  - Possibly easier to retrieve a document from a vault via GUI than trying to query the bot for it, especially if you're not sure what to ask ("browsing" the vault).
- It's optional, meaning we can operate fully via Discord and the backend, but this can be a separate utility the team can open in a browser.

This covers the design considerations for an external visualization. Now that we have covered all domain topics, we will proceed to compile the key outputs and artifacts: diagrams, schemas, workflows, etc., that summarize our design in more concrete terms.

---

5. Key Design Artifacts and Specifications

In this section, we present structured outputs summarizing the proposed system. These include diagrams, schemas, matrices, and examples as requested.

5.1 Context Model Diagram (Hierarchy and Vaults)

Below is a representation of the system's context hierarchy and how information flows between levels. Each node corresponds to a context vault and typically has an associated bot. Arrows indicate parent-child relationships in the hierarchy, and dashed/dotted lines represent other relationships (such as personal contributions and global insights usage):

```
Client Context (Client Bot)
 └─ Product Context (Product Bot)
      └─ Project Context (Project Bot)
           ├─ [Ops Segment]
           ├─ [Research Segment]
           ├─ [Timeline Segment]
           ├─ [Design Segment]
           └─ Subproject Context (Subproject Bot)
                └─ Task Context (Task Bot)
```

- Each **Project Context** contains multiple **Segment** buckets (Operations, Research, Timeline, Design, etc.) that organize content by type.
- Each **Subproject Context** exists within a Project and contains detailed task outputs and any subproject-specific notes.

- Each **Task Context** is a transient thread for active work, feeding results upward to Subproject.

Other elements:

- **Global Design Insights Library** (with its own store and possibly a dedicated “Insights Bot” or accessible by all bots) is separate, but bots at all levels (mostly task/subproject) can pull from it (depicted as a dotted line feeding into Task level).
- **Personal Context Vaults** for each person (handled by Person Bots) stand to the side. A person’s bot can access the shared contexts *for the projects that person is involved in*. Personal vaults can contribute to subprojects when the person shares notes (depicted as a dashed line from Person vault to a Subproject or Project vault, labeled “contributes to”).

(In a static text medium, the above tree illustrates the containment. Dashed arrows (not shown in text) would connect Person Vaults to the places they have access, and a dotted arrow from Global Insights into all levels of context retrieval.)

Explanation: This diagram shows the hierarchical **information architecture**. Each context vault (Client, Product, Project, etc.) collects information relevant to that scope and abstracts it for upward use. The Person and Insights nodes are non-hierarchical: Person vaults are individual-specific and can feed into any level when that person shares knowledge; the Global Insights library is a universal resource that can be tapped by any context as needed.

5.2 Metadata Schema (v1)

Here is the proposed YAML frontmatter schema for content ingestion, including fields, their types, and usage:

- **id** (string, required): Unique identifier for the content item. Can be a UUID or a composite like projectX_research_2025-07-19 for easier human reference. Uniqueness allows linking and update tracking.
- **level** (enum, required): The hierarchy level of the content’s primary relevance. One of: client, product, project, subproject, task, person, global-framework. This determines which vault it goes to.
- **client** (string, if applicable): ID or name of the client. Required if level is client or below (except person/global). It ties the content to the top-level client container.
- **product** (string, if applicable): ID or name of the product. Required if level is product or below (except person/global).
- **project** (string, if applicable): ID or name of the project. Required if level is project or below (except person/global).
- **subproject** (string, if applicable): ID or name of the subproject. Required if level is subproject or task.
- **task** (string or number, if applicable): ID of the task. Only used if level is task (or if linking a document specifically to a task).

- **person** (string, if applicable): The username or ID of the person this content is for/from. Only for level: person items (personal notes).
- **segment** (enum or string, optional): The category of content within a project. E.g., operations, research, timeline, design, etc. Only meaningful for project or subproject level items (tasks could optionally tag a segment if their output pertains to one).
- **status** (enum, required): Lifecycle status of the content. One of: draft, working, final, archived, deprecated.
  - draft: Newly created, not finalized.
  - working: Actively being edited or added to (for multi-edit docs or evolving content).
  - final: Completed or approved content.
  - archived: No longer active but kept for record.
  - deprecated: Outdated or replaced content (should generally not be used going forward).
- **nuance_level** (enum, optional): Indicator of detail level. Options: high (very detailed/nuanced), medium, basis (distilled core info). This helps in filtering what to include in higher-level summaries.
- **active_importance** (enum, optional): active or passive. active means this content should be automatically surfaced in relevant contexts (high importance), passive means only fetch if specifically relevant or queried. Default could be passive unless marked.
- **authors** (list of strings, optional): The Discord usernames or IDs who authored or contributed. Useful for attribution and for personal bot linking.
- **meeting_ref** (string, optional): If the document is a meeting transcript, a reference ID or name of that meeting (could be a date-time or title). Helps link transcripts to calendar events or to multiple parts if a meeting is split.
- **linked_items** (list of strings, optional): A list of IDs of other content items related to this one. Could represent:
  - For a task artifact: link to other tasks it relates to.
  - For a summary: link to source docs or tasks it summarized.
  - For a document update: link to the previous version's id.
  - For a personal note: link to a project or task it references.
- **permissions** (enum or list, optional): Access level. Could be a simple enum like:
  - default (visible to team members of that project by default),
  - restricted (sensitive, maybe only leads or specific roles),
  - inherit (inherits from parent context; e.g., if project is confidential then all sub-items inherit that).
  - We could extend this to lists of roles or usernames for fine-grain control if needed, but likely not in v1.
- **created** (datetime, required): ISO8601 timestamp of creation/upload time.
- **updated** (datetime, optional): ISO8601 timestamp of last update/edit.

This is our starting schema. It might be refined as we implement (for instance, adding a title field could be helpful for easy identification, though often the first line of the content or file name serves as title).

Example Metadata Frontmatter:

```
---
id: "projX_research_user-interviews_v1"
level: project
client: ACME_Corp
product: AlphaApp
project: WebsiteRedesign
segment: research
status: draft
nuance_level: high
active_importance: passive
authors: ["@alice", "@bob"]
created: "2025-07-10T14:30:00Z"
---
```

(The body would then follow, e.g., the content of a research findings document.)

In the above example, it’s a draft research doc for project WebsiteRedesign under AlphaApp for client ACME_Corp. It’s high nuance (likely raw interview notes), and by default will not be surfaced unless asked (passive). Authors Alice and Bob wrote it.

Another quick example for a task summary:

```
---
id: "task123_summary"
level: task
client: ACME_Corp
product: AlphaApp
project: WebsiteRedesign
subproject: LoginFeature
task: "123"
segment: design
status: final
nuance_level: basis
active_importance: active
authors: ["@charlie"]
linked_items: ["task122_summary"]
```

```
created: "2025-07-19T10:00:00Z"
---
```

This indicates a final output of task 123 in the LoginFeature subproject, design segment. It's basis (distilled decision), very important for context (active), perhaps linked to task122_summary if that was a related prior task.

5.3 Content Lifecycle Specification

We define the state transitions and triggers for content at various levels. Below is a state diagram in textual form, followed by rules/algorithms for upward synthesis and abstraction:

States and Transitions:

- **Draft → Working:**
  - Trigger: Content is actively being edited or discussed.
  - Example: A doc uploaded as draft becomes "working" once people start collaborating on it or once it's confirmed as the base to iterate on.
  - Not all content will explicitly hit a "working" state; some might go from draft straight to final (if it's approved as is).
- **Working → Final:**
  - Trigger: Content is completed/approved.
  - Example: A task discussion has concluded and a final summary is produced (task context goes to final artifact), or a design doc reaches sign-off.
  - Action: Mark status as final, record who approved (if applicable), and initiate any upward propagation (detailed below).
- **Final → Archived:**
  - Trigger: The content is no longer current or the project/subproject is done.
  - Example: A subproject summary when project finishes might be archived (still stored but not considered active). A final doc might be archived when a new version is created.
  - Action: Possibly move the file to an archive folder or mark metadata archived. Bots generally exclude these from active context unless explicitly asked for historical info.
- **Any → Deprecated:**
  - Trigger: Content is found to be incorrect or superseded by something else.
  - Example: An initial requirement doc is deprecated after requirements change entirely; an insight proven wrong.
  - Action: Mark status deprecated; bots should not use this content unless for historical analysis. We may still keep it for record (with a note perhaps pointing to the replacing info).
- **Task-specific states:**

    - Task thread opened (implicitly a Working state for the task).
    - Task completed (Final artifact created, thread closed).
    - Task thread archived (once done and not needed for active conv).
    - We can treat a task as an entity that goes: Open -> Completed -> Archived.
- **Subproject states:**
    - Active (Working): while tasks are ongoing.
    - Completed (Final): when its goal achieved.
    - Archived: after it's integrated into project and no longer active.
    - Possibly On-Hold could be a state if we pause a subproject mid-way; that would be akin to working but inactive tasks.
- **Project/Product states:**
    - Active (ongoing) vs Completed vs Archived similar to above. Many projects might remain "active" until formally closed.

Triggers for Upward Synthesis (Promotion):

- **Task Completion (Working → Final for a task):**
    - Trigger: User signals task done (via command, or marking thread resolved, or simply stating conclusion).
    - Action: Task Bot generates or collects final artifact. The artifact (content) itself goes into final state.
    - Then:
        - Save artifact in Subproject vault (level: task, status: final).
        - Notify Subproject Bot (so it can update any subproject summary if needed).
        - Possibly auto-summarize multiple task artifacts if that was last task of subproject (see subproject triggers below).
    - Example algorithm: On task_done event -> fetch last summary of conversation + key decisions -> format into markdown -> call Dump Bot to ingest to subproject context.
- **Subproject Completion (Working → Final for subproject):**
    - Trigger: All planned tasks for subproject are done, or user explicitly closes subproject.
    - Action:
        - Subproject Bot (or Project Bot) compiles a **Subproject Summary** document. This involves summarizing all task outputs and highlighting changes/decisions.
        - That summary is given level: subproject or project (we could store it at project level since it's an input to project context).
        - Status of subproject summary = final.
        - It is then **promoted** to Project vault (likely in the segment relevant; e.g., if subproject was about design, maybe into project's design segment or operations).

        - Mark subproject context as archived (so tasks and subproject-specific channel become read-only).
        - Set project's state or progress to reflect subproject completion.
    - Possibly trigger Product update if this subproject completion significantly changes something at product level (like a major feature done).
- **Project Milestone/Completion:**
    - Trigger: Either a milestone is reached (which might correspond to a subproject completing if each subproject = milestone), or whole project done.
    - Action:
        - Generate a **Project Update** or Final Report:
            - If milestone: an interim project update summary (could be after a subproject or monthly, etc.).
            - If project complete: a final project report capturing objectives, outcomes, metrics, insights.
        - Store that in Product vault (since it's relevant for product-level knowledge).
        - If final, mark project as completed in metadata (status final for project context).
        - Possibly glean any global insights from it (if something broad was learned, extract and add to global library).
        - Notify Product Bot to integrate the results (like update the product's feature list or performance metrics).
        - Archive project vault if fully done (or leave it read-only).
- **Product Update:**
    - Trigger: A project under the product finishes or some major change occurs (or periodic update).
    - Action:
        - Update the **Product Context**:
            - For example, add "Feature X implemented" to product documentation, or update product metrics.
        - If product reaches a new version or goal, maybe create a snapshot doc.
        - If the product itself is done (e.g., end of engagement or product retired), mark it archived.
        - Possibly create a high-level client report if needed at certain junctions.
- **Periodic Synthesis:**
    - Not all synthesis is event-driven. We might schedule periodic refresh:
        - e.g., weekly project summary even if not complete, to keep everyone updated and check if the abstraction is staying current.
        - This can be triggered by a cron or by a command ("@ProjectBot summarize weekly").

Transformation/Abstraction Rules:

As content moves up:

- Include only **Basis information** from lower level:
    - Identify key decisions, results, metrics. Omit step-by-step details or minor alternative ideas unless they clarify the decision.
- Preserve context needed to understand the why of decisions:
    - Could be a short note on reasoning if crucial (e.g., "…because user feedback indicated X").
- Omit personal attributions or colloquial text:
    - For external summaries, don't say "Alice discovered that …", but maybe internally it's okay. Decide style per audience.
- Compress multiple related points:
    - If Task A found X, Task B found Y, at project level just say "Subproject achieved X and Y".
- Mark uncertain info:
    - If some subproject left an open question, note it so higher level knows there's a risk or follow-up needed.
- **Tool support:** likely the upward summary generation will use templates or LLM prompt specifically fine-tuned to produce concise summaries (e.g., "Summarize the following tasks results in 5 bullet points focusing on outcomes and decisions").

Verification and Human Oversight:

- Ideally, a human reviews any major summary or client-facing output. The system can prepare drafts but should allow editing.
- E.g., the Project Bot posts a draft of the project update in the project channel for the lead to approve before it's considered final and moved to product vault.

Versioning Rules (revisited briefly):

- If an upward summary is updated (say a subproject summary after each task), we may overwrite the old or keep a log:
    - Could have subproject_summary_v1, v2, etc., or simply update the same file.
    - Leaning towards updating same to represent current truth, but archive older inside an "archive" folder if needed.
- The updated timestamp in metadata helps know that it's been refreshed.

This content lifecycle ensures that knowledge flows upward systematically and remains current at each level, avoiding stale info lingering un-updated.

5.4 Retrieval Strategy Matrix (Context by Hierarchy Level)

The following table summarizes what context each level's bot will retrieve, how much nuance is included, and how it contributes upwards:

Level	Context Automatically Pulled (Active)	Nuance Allowed	Primary Sources (via Retrieval)	Outputs Upwar
Task (granular work)	- Task thread history (recent messages or summary of it)- Parent Subproject summary/overview (goal, scope of subproject)- Key relevant outputs from sibling tasks in the same subproject (final artifacts of recent tasks)- Project-level requirements or constraints related to this task (basis info, e.g., “must follow design system” from project or product)- Relevant Design Insights (Global principles or local lessons matching task topic)	**High Nuance.** Task can handle detailed info and edge cases from subproject. It will include raw data if needed (e.g., excerpt from a research doc) because the people working on the task may need the nitty-gritty.	- **Subproject Vault**: all documents and task outputs in this subproject (with vector search to find specifics)- **Project Vault (filtered)**: only basis docs like requirements, style guides (maybe by metadata tag basis/active)- **Product Vault (filtered)**: maybe one or two key product constraints (like platform guidelines)- **Global Insights**: search for any principle matching task context- **Personal notes** (if the task assignee has relevant personal notes and asks for them)	- **Final Task Artifact** (distillec result of task) → added to Subproject vault- Possibly an update in subproject channel if something significant decided (so subproject summary can be updated)
Subproject (feature/phase)	- Aggregated results from all completed tasks in subproject (the “mini knowledge base” for this subproject)-	**Medium-High Nuance.** Subproject context carries some	- **Subproject Vault**: primarily task final artifacts and any subproject notes (maybe	- **Subproject Summary** (covering all tasks, outcomes decisions) → delivered to Project vault anc

Level	Context Automatically Pulled (Active)	Nuance Allowed	Primary Sources (via Retrieval)	Outputs Upwar
	Outstanding issues or questions within subproject (if any tasks failed or left open points)- Project-level direction specific to this subproject (e.g., “this feature must achieve X metric” from project requirements)- Product basis if relevant (e.g., ensure consistency with overall product)	nuance (because tasks results are relatively detailed), but less than raw task chats. It might include important context from tasks like why a decision was made, but mostly focuses on outcomes.	a subproject plan)- **Project Vault**: relevant project docs that apply (maybe research or ops docs that set subproject constraints)- **Tasks’ ephemeral info**: rarely needed, but if a question requires it, could fetch a specific task log- **Product Vault**: only if something at product-level directly affects subproject (not common, maybe a technical standard or API the product provides)	Project Bot- Possibly local design insight if something learned (which could also go to project’s insights
Project (overall project)	- Distilled summaries of each subproject (especially the active or recently completed ones). This gives the project’s current status across segments.- Key project documents by segment (project plan from Ops, key findings from Research,	**Medium Nuance.** The project level is a mix: mostly basis info and medium-level detail. It avoids task-level chatter, but may include some	- **Project Vault**: subproject summaries, project segment docs (Ops, Research, etc.) using metadata filter (e.g., if asked about timeline, fetch timeline segment doc)- **Product**	- **Project Updat** (e.g., milestone report or final project report) — to Product vault (and possibly to client via produc or client bot)- **Insights/Lessor Learned** (local insights doc) → Project vault (an flagged for globa if generalizable)

Level	Context Automatically Pulled (Active)	Nuance Allowed	Primary Sources (via Retrieval)	Outputs Upwar
	timeline status, design prototypes if needed for context).- Basis information from these segments (milestones, objectives, success criteria).- Product-level context: original product requirements or KPIs that this project is aiming to impact. (Keeps project aligned with product goals.)	rationale from subprojects if needed to explain project decisions. It should focus on cross-task synthesized knowledge.	**Vault**: product requirements or design guidelines relevant to project (likely directly known, not huge in number)- **Subproject Vaults**: if needed for more detail on a specific point, can dip into a subproject's detail (vector search a subproject if a question is specific and not answered by summary)	
Product (product/program level)	- High-level product documentation (vision, roadmap, core features).- Summaries of each project under the product (especially recent ones) to know what has been delivered or is in progress.- Overall status metrics for the product (e.g., user numbers, performance metrics if available from projects or operations).- Any global considerations like	**Low Nuance.** Product context is mostly basis. It does not dive into how each project achieved things, just what was achieved. It should be concise – e.g., "Project X delivered feature Y which resulted in	- **Product Vault**: product spec, design system, global goals (straight retrieval of known docs)- **Project Vaults**: final outputs of projects or their final reports (to pull in any specific info needed from a particular project if asked)- **Global Frameworks**: if the product	- **Product Snapshot/Repo** (covering overall product state an summary of recent project outcomes) → to Client context (fo client reporting o internal execs)- Possibly update to global knowledge if product yields ar strategic insight (e.g., "Product Alpha shows trend X among users" could be useful beyond one product)

Level	Context Automatically Pulled (Active)	Nuance Allowed	Primary Sources (via Retrieval)	Outputs Upwar
	regulatory requirements or design frameworks that apply to the product (if the product has to comply with something across projects).	Z outcome". Only critical issues or exceptional stories from projects bubble up here.	has adopted any specific framework (like a certain UX style), that might be referenced (though global frameworks are broad, product might have its own customizations of them)- **Client Vault** (maybe client preferences if any, since product often ties into client strategy)	
Client (client/org level)	- Consolidated summaries of each product for that client (so the client's entire portfolio status).- Key outcomes and value delivered (ROI, high-level metrics) for the client.- Any critical issues or high-level risks that need client attention (e.g., a delay in one product launch might be mentioned but framed appropriately).- Global insights or principles might be reflected indirectly (like "all	**Very Low Nuance.** The client level should contain almost no granular detail. It's essentially executive summary material only. Everything here is basis or aggregate data. Internal nuances or team discussions are completely filtered out,	- **Client Vault**: client briefs, contract, overall objectives (for context of what's expected)- **Product Vaults**: uses each product's snapshot info. Likely the client vault actually *contains* those snapshots for easy access (copied or referenced). So retrieval is straightforward from an aggregated client report	- **Client Report/Update**: document or set of talking points for the client covering all products and projects (often prepared for a meeting or quarterly review) This is generate for human consumption by client or execs.- **Organizational Knowledge**: If applicable, the lessons from working with this client (like doma insights) could feed into a globa knowledge base for future similar

Level	Context Automatically Pulled (Active)	Nuance Allowed	Primary Sources (via Retrieval)	Outputs Upwar
	solutions follow our UX philosophy” but not the detailed principles themselves).	except possibly a high-level reference to “some challenges were overcome” without specifics.	that the team maintains.- Possibly market research or global context if relevant to client (if client cares about competitor or industry trends, might include in client-level data, but that’s external info beyond our system scope).	clients, but that’s outside the immediate hierarchy.

Key Points from the Matrix:

- Task level pulls from everywhere relevant but focuses on local details, allowing high nuance and using powerful search to grab specifics. It outputs a refined result to subproject.
- Subproject level pulls mostly from its tasks and some project context, with medium-high nuance (because it may include why decisions were made across tasks). It outputs a coherent summary to project.
- Project level pulls distilled results from subprojects and project docs, with medium nuance, focusing on decisions and overall direction. It outputs an update to product.
- Product level pulls only distilled results from projects and core product info, with low nuance, focusing on features delivered and goals. It outputs a product summary to client.
- Client level pulls those product summaries and focuses only on outcomes and value, effectively no nuance, for client consumption.

The retrieval methods (e.g., vector search vs direct lookup) each bot uses in these contexts were described in section 4.5 and align with the sources listed above.

5.5 Bot Interaction Contracts (Per Level)

Here we outline each bot’s “contract”: what it takes in, what it provides, and any special permissions. This is almost like an API spec for each bot’s behavior in the Discord context.

Client Bot:

- **Inputs (from user):** High-level questions about overall progress, product statuses, outcomes, or requests for reports. These might come from leadership or someone preparing client communication.
- **Inputs (from system):** Notifications of major updates (e.g., Product Bot could feed it when a project finishes).
- **Processes:** On a query, it will:
    1. Identify which products under the client are relevant (if question is generic, might gather all; if specific, just that product).
    2. Retrieve latest product snapshots and any client objectives relevant.
    3. Formulate a concise answer, possibly doing minor calculations (like total number of features delivered across products if asked).
    4. Ensure no restricted internal details are included (it sanitizes output).
- **Outputs (to user):** Answers in natural language focusing on results/value. It might also output documents like a “quarterly update” if asked (embedding charts/tables if integration possible, but likely just text summary).
- **Permissions:**
    - It can see product and project summaries but not raw tasks.
    - It has a ‘client-safe’ view meaning certain fields or tags (like cost breakdowns if sensitive) are either not accessible or auto-redacted.
    - It only responds in designated channels (client updates channel or DM to authorized person).
- **Example:** User asks, “@ClientBot, how is AlphaApp performing overall?” – It responds with a summary: “AlphaApp has completed 2 major projects this quarter, delivering Feature A and B, resulting in a 15% increase in user engagement. Project C is in progress and on track for next quarter.”

Product Bot:

- **Inputs (from user):** Questions about a specific product’s development, list of features, overall health, cross-project coordination. E.g., “What features were delivered for AlphaApp in the last release?”
- **Inputs (from system):** Project bots might feed it update info when projects complete or hit milestones.
- **Processes:**
    1. If the query is about a specific project, it might fetch that project’s data.
    2. If it’s about the product in general, it collates data from all projects: for example, aggregate feature lists or statuses.
    3. It references the product’s own documentation as needed (for context like product vision).
    4. It might use a small reasoning step to combine project info (like merging outcomes from multiple projects into a coherent answer).
- **Outputs:**

    - Answers that may include bullet lists of features or projects, or timeline info for the product.
    - Could also create a “Product roadmap update” if asked, detailing each project’s contribution.
    - Not as fine-polished for client consumption as ClientBot, but still fairly high-level.
- **Permissions:**
    - Visible internally to product team members. If the client is not supposed to see everything, the client wouldn’t have access to product channel anyway.
    - It has access to all project-level info (including some nuance if needed to answer, but it should abstract it).
    - Should not reveal internal issues outside of context – e.g., wouldn’t volunteer “Project X is delayed because Bob is on leave” unless asked internally.
- **Example:** Team lead asks, “@ProductBot, list all completed features for AlphaApp this year.” The Product Bot replies with a list by project: “Project X: delivered Feature A and B. Project Y: delivered Feature C,” etc., possibly noting impact or status.

Project Bot:

- **Inputs (from user):** Questions about project status, deadlines, what decisions have been made, what the research says, etc. Also requests for summaries (“summarize the project for me”).
- **Inputs (from system):** Subproject bots or tasks ping it when something finishes or if they need clarity on a requirement.
- **Processes:**
    1. If asked a direct question, it will filter context by segment (if question mentions timeline, go to timeline data; if general, gather from all segments).
    2. It can list tasks or subprojects and their statuses.
    3. It often simply retrieves the needed doc (like the latest timeline) and summarizes or presents it.
    4. If asked to escalate (e.g., produce a product-level update), it compiles info appropriate for that.
    5. If subprojects are active, it keeps track of which one is active (state).
- **Outputs:**
    - Informative messages, sometimes with structured data (like “Milestone X: achieved on Jan 1; Next Milestone: Feb 1”).
    - It might also produce documents: e.g., if told “create project summary”, it might output a formatted summary in the chat or as a file.
    - Could use bullet points for clarity if listing items.
- **Permissions:**
    - Accessible to all team members on that project.
    - It holds all project internal info so it must restrict usage to project channel.

    - If someone not on the project somehow interacts, it should verify or politely refuse (though if Discord roles are correct, that scenario won't occur often).
- **Escalation Hooks:**
    - If user asks something that should be answered by Product Bot, Project Bot might fetch product info or instruct user to ask product bot. But likely project bot can answer product-related questions only in broad strokes since it knows product constraints.
- **Example:** PM asks, "@ProjectBot, what were the key findings from our research segment?" – Project Bot might answer, "We conducted 10 user interviews. Key findings: 1) Users struggle with onboarding, 2) Feature X is highly requested, 3) … (from Research doc dated 2025-07-15)." It pulled these from the research segment doc summary or highlights.

Subproject Bot:

- **Inputs (from user):** In subproject channel, someone might ask, "What tasks are remaining?" or "Summarize what we've accomplished so far on this feature." They might also discuss approach, and the bot may chime in if needed.
- **Inputs (from system):** Task bots notify it on completion of tasks. Project bot could feed it high-level goal info at creation.
- **Processes:**
    1. Maintains a list of tasks in this subproject and their status (this could be gleaned from metadata or a simple list updated when tasks finish).
    2. If asked for status, it will report tasks done vs open, any key outcomes (basically a mini progress report).
    3. If asked a detailed question, it might retrieve the relevant task's summary to answer.
    4. It can also proactively note if progress is stalled or if new tasks need definition based on conversation.
- **Outputs:**
    - Answers about subproject scope or progress ("3 out of 5 tasks completed, currently working on Task Z, results so far: …").
    - When a task completes, it might output a brief summary of that task's result in the channel.
    - Possibly a compiled subproject summary if asked or at subproject end.
- **Permissions:**
    - Channel likely visible to project team members. It contains some nuance (task outputs are detailed, but if someone outside project was here, they'd see it – though ideally only relevant people are in subproject channel).
    - It doesn't need to hide much internally; it's more about not exposing outside.
- **Example:** Developer asks in subproject channel, "@SubprojectBot, where do we stand?" Bot replies, "LoginFeature Subproject: Goal is to improve sign-in UX. Completed: Task1 (design done), Task2 (API done). In progress: Task3 (frontend integration). So far, design

tests show 20% faster login times. We still need to address social login (planned in Task4).”

Task Bot:

- **Inputs (from user):** In the task thread, could be anything from “Help me recall X info” to “Draft an email for Y” to general questions like “What are some approaches to this problem?” The user might directly instruct it or just mention something that triggers it to assist.
- **Inputs (from system):**
    - On thread start, the subproject bot or some orchestrator might send it a brief of the task (if available).
    - It might also get a copy of subproject context or at least pointers.
    - It monitors every user message in the thread to decide if it should act (like a background listener).
- **Processes:**
    1. For any query or when it senses it should help, it identifies key terms (e.g., if user says “I recall a doc about this from last month”, it will search relevant vault).
    2. Uses retrieval (likely vector search plus metadata filtering as discussed) to gather relevant context.
    3. Constructs a helpful response or performs a requested action (like summarizing or writing something).
    4. It also updates its internal short-term memory (like updating the conversation summary).
    5. When conversation hits certain points (like decision made), it may log that.
- **Outputs:**
    - Answers with referenced info (“According to the requirements doc: …”).
    - It may present lists, summaries, or direct quotes from content as needed.
    - Intermediate summaries or suggestions.
    - On finalization, it outputs the final summary artifact (maybe tagging Subproject Bot or sending to Dump Bot).
- **Permissions:**
    - It has broad read access to needed context (project, subproject, etc.), but it should not spontaneously give context that wasn’t asked for unless it’s sure it’s relevant (to avoid info overload or revealing things out of scope).
    - In a private task thread with only team members, likely fine to use anything from that project. If a task had a special restriction (like maybe a confidential investigation), that could be an exception.
    - It will not reveal info from other projects or tasks unrelated (unless user specifically asks and presumably they have that access).
- **Example:** Intern asks, “@TaskBot, what did we decide about password requirements in the last project?” The Task Bot finds an insight or requirement in Project vault and

answers, “In the previous project, we set the password requirements to min 8 chars, with at least one number (Project Security Guidelines, 2024). It was decided to keep it simple to improve signup rates.”

Dump Bot:

- **Inputs (from user/system):** File uploads or posts in the dump channel; possibly commands like “/ingest this as project X”.
- **Processes:** As detailed in ingestion, it parses metadata, or if none, tries to classify. It then calls the storage APIs to save the data and update indices.
- **Outputs:** Confirmation messages (“Document X categorized under Y.”) or clarification requests (“Please provide project name for this document.”). Could DM the uploader if needed for clarification.
- **Permissions:**
  - Likely an internal bot only, everyone can drop files, but only it posts there.
  - It can see all contexts to know where to put things, but it should not reveal content in doing so beyond saying where it went.
- **Example:** User uploads design-spec.md with frontmatter specifying project and segment. Dump Bot posts: “✅ design-spec.md stored in Project WebsiteRedesign > Design segment.”

Person Bot:

- **Inputs (from user):** Private messages such as “What did I do last week?”, “Summarize my notes tagged #ProjectX”, “Remind me to finish task Y”, “Share note Z with Project channel”, etc. Could also be general like “brainstorm ideas for feature”.
- **Inputs (from system):** Possibly calendar events or assigned tasks from project metadata where author = that person (if we push such info to them).
- **Processes:**
  1. For a question like “what did I do”, it queries the metadata DB for tasks completed by that user or notes in their vault from that timeframe.
  2. For summarizing notes, it retrieves those note contents (maybe via vector search if topic-based).
  3. For brainstorming, it’s essentially just an LLM with context of user’s notes and general knowledge.
  4. For sharing, it will format and send the content appropriately.
  5. It may maintain a profile of the user’s interests or style to personalize responses somewhat.
- **Outputs:**
  - Direct answers to user in DM.
  - Confirmation when it does something (“I’ve shared your note in Project X.” or “Reminder set for tomorrow 10am.” if we implement reminders).

    - Possibly it could push summary of person's weekly activity to them on Monday mornings proactively.
- **Permissions:**
    - It can access the person's notes fully.
    - It can access shared project info limited to what the person can see (basically projects they're in, or everything if no restrictions internally).
    - It should not send the person restricted info from projects they aren't in.
    - It acts as a guardian of their privacy to the outside, only sharing when told.
- **Example:** Person says, "@MyBot, generate a brief of all tasks I completed in Project WebsiteRedesign." Person Bot finds tasks where authors included that person, and replies with something like: "In Project WebsiteRedesign, you completed: Task 5 (design login UI) – delivered login mockups; Task 7 (user testing) – summarized findings; Task 9 (fix bugs) – resolved 5 issues. Great job!"

This contract outline ensures each bot's functionality and limitations are clear. Next, we detail the ingestion pipeline flow to see how raw data moves through the system.

5.6 Ingestion Pipeline Flow

The ingestion process from Discord dump to storage can be outlined in steps:

1. **User Drops Content:** A team member drags-and-drops or pastes content into the designated `#dump` Discord channel (or a specific channel like `#ingest` or `#transcripts` if separated by type). This could be a file attachment (doc, PDF, markdown) or a long text (like pasted notes or a transcript).
2. **Dump Bot Capture:** The Dump Bot (listening to that channel) detects the new message/file. If it's an attachment, it downloads the file. If it's text, it reads it directly.
3. **File Type Handling:**
    - If the file is Markdown or text: read as text.
    - If it's a PDF/DOCX: use an extractor (could be a library like Apache Tika or a PDF parser) to get plaintext and any metadata.
    - If it's an image (like a photo of whiteboard, or screenshot): optionally run OCR to get text (perhaps not in v1 unless needed).
    - If it's a JSON or other structured data (less likely, but maybe output from a tool): parse accordingly.
4. **Metadata Extraction:**
    - Attempt to parse YAML frontmatter at the top if present. If the content has lines beginning --- and ending ---, treat that as metadata block.
    - If frontmatter found: parse key-values into a metadata dictionary.
    - Validate required fields (level, id, etc.). If something essential is missing (like no level), mark as incomplete classification.
    - If no frontmatter: fallback classification. For example:

        - Check the Discord message context: did the user mention some key? (E.g., user might type “Project: WebsiteRedesign” in message).
        - Check file name: maybe WebsiteRedesign_DesignDoc.pdf contains hints (project name and segment).
        - Perhaps open the file text and scan for known project names or keywords like “Client: ACME Corp” in text.
        - Use default values if possible: maybe assume draft status, assume nuance high if it’s raw text, etc.
    - If still ambiguous: The Dump Bot can reply asking: “Could not determine target project for this file. Please provide a hint or include metadata.” This is an interactive step where a user might respond with clarification or edit the message with frontmatter.
    - There could also be a simple classification model or rule-based mapping: e.g., if it finds words like “Meeting Transcript” and a date, maybe map to operations segment of a known ongoing project.
5. **Routing Decision:**
    - Based on metadata level and IDs, determine where to store:
        - If level=project or below: identify the exact project vault (via client->product->project keys).
        - If level=global-framework: route to global insights library.
        - If level=person: route to that person’s vault (which might not be in the same structure; maybe a separate store).
    - Also note segment to categorize within vault.
    - If the hierarchy references (client, product, etc.) don’t exactly match an existing entity (typo or new entity):
        - If new client/product/project is allowed (maybe the system admin has to set these up first, or maybe the bot can create one on the fly if it’s a new client):
            - Possibly the bot could create a new vault folder if new. Better to have known list though to avoid typos creating duplicates.
        - If it seems like a typo (off by one letter from existing project), bot might ask confirmation or auto-correct if confident.
6. **Storage Actions:**
    - **Save File Content:** Save the content into appropriate location. If using a file system:
        - Construct path like /Client/AlphaApp/WebsiteRedesign/Research/docname_v1.md.
        - If the content is originally PDF, maybe convert to markdown or txt for storage plus keep original PDF in an attachments folder.
    - If using database for content, create a record and store text (though large text maybe better stored as file or in a text column with full-text index).
    - Save the metadata in the metadata index (DB). For example, insert a row in documents table with columns (id, client, product, project, subproject, task, segment,

status, etc, plus file path if needed).

- If it’s an update (metadata might have same id as existing or some “supersedes” link):
  - Mark previous record as archived/deprecated.
  - Or update it. We likely preserve old separately though, and maybe just mark newest as active.
- If the content is a meeting transcript or something that could be chunked, might immediately do chunking and embed those:
  - For large docs, split into segments (keeping metadata tags).
  - Compute embeddings for each chunk (likely offline or asynchronous if heavy, but for now could do on the fly).
  - Store those in vector index with references back to doc id and chunk number.
- If it’s small (like a one-page note), embedding it as one chunk is fine.

7. **Post-Storage Confirmation:**
   - The Dump Bot then posts a message in the dump channel confirming, e.g.,
     - “Stored **design-spec.md** -> Project WebsiteRedesign (Design). Status: draft.”
   - Or if any info had to be assumed, mention it:
     - “Assumed segment ‘operations’ due to no segment provided.”
   - If the item is especially important (active_importance), it might also notify the project or relevant channel:
     - Perhaps the Project Bot could be triggered to post, “New Research document added: User Interview Notes (draft)” in the project channel or subproject channel. This could be direct by Dump Bot or via Project Bot after insertion.
   - These notifications keep the team aware and also double as a log of new info.
8. **Error Handling:**
   - If classification failed or permission issues (maybe someone tries to upload to a project they shouldn’t):
     - Bot warns or asks for intervention.
     - Possibly put content in a “holding area” (maybe mark it unfiled) until resolved.
   - If storage fails (disk full, etc.), bot alerts an admin or in a log channel with error.
9. **Versioning & Duplicate Handling:**
   - If a file with same name or id is already stored, the bot might ask “Update existing [doc id]? (yes/no)” or version it automatically.
   - Based on metadata ‘updated’ timestamp, maybe automatically assume it’s an update if timestamp is newer and id matches.
   - If duplicate content with different id is uploaded, treat as separate.
   - The bot could also detect if content is very similar to an existing doc (like 90% same text) and warn of duplicates.
10. **Index Update:**

    - After adding to DB, update any full-text search indices (if we have for metadata or content).
    - Update vector index as mentioned.
    - If we have tag/keyword index for quick filter, update that too.

Meeting Transcript Special Flow:

- Possibly, meeting recordings are transcribed outside and then the text file is auto-dropped in a transcripts channel. Dump Bot sees it:
    - It might have naming like ProjectX_meeting_Jul19.txt.
    - Could parse that name or look inside: often transcripts start with participants or meeting title which could hint project.
    - Or the integration that posts it might include a comment or metadata.
- Then follow same steps to store under operations segment maybe, with meeting_ref if known.

Personal Notes Ingestion:

- If a person's Obsidian note is to be shared:
    - Possibly the Person Bot or user themselves will copy the note content into the dump channel (maybe via command).
    - Then Dump Bot handles it like any markdown.
    - Alternatively, the Person Bot might bypass the dump channel and directly call the internal ingestion function with the note (since it knows metadata).
    - In either case, once in the shared space, it's treated like a normal doc belonging to some project.

This pipeline ensures that every piece of information gets systematically slotted into the knowledge base with proper labeling, making subsequent retrieval effective. The next workflow describes how information flows upward after tasks and subprojects finish.

5.7 Upward Synthesis Workflow

This describes how final information gets propagated up the hierarchy:

Task to Subproject Upward Synthesis:

1. **Task Finalization:** Task thread concludes with a final artifact (e.g., a summary, a decision, a design output). Let's say the intern confirms completion.
2. **Task Bot Prepares Artifact:** It may combine the final summary text and any attachments (like if there's an image of design, it references it).
3. **Promotion to Subproject:** The Task Bot or Subproject Bot does the following:
    - Formats the final artifact as a Markdown file with frontmatter (level: task, linked_items maybe other tasks, etc.).

- Either posts it to the subproject channel for visibility or directly to Dump Bot (to store it). Possibly it can do both via an API call to ingestion (to avoid a loop of Dump Bot reading it from same server, maybe directly store since it's internal).
- The subproject vault now has this new item.

4. **Subproject Context Update:** Subproject Bot gets notified (if it wasn't the one doing it already). It then:
   - Updates a running subproject summary document (if we maintain one continuously). For example, add a line in "Subproject Summary.md" about the task result.
   - Or simply knows to include that task's result in future answers.
   - If this task was addressing an open question, mark that closed.
   - If all tasks done (check if this was last planned task), trigger subproject completion sequence (below).

Subproject to Project Upward Synthesis:

5. **Subproject Completion Trigger:** Project lead or the Subproject Bot notices that subproject is done (either all tasks finished or goal met).
6. **Subproject Bot Compiles Summary:** It gathers:

- The goals of the subproject (from its initial definition).
- All task final outputs.
- Any metrics or results achieved.
- Challenges/insights encountered.
- Formats a subproject summary (could be a short report or bullet list).
- E.g., "Subproject LoginFeature Summary: Implemented new login UI, reducing average login time by 30%. Key decisions: Used OAuth for social login, simplified password rules. Remaining issue: need to monitor security impact. All tasks completed."

7. **Promotion to Project Vault:**
   - That summary is given appropriate metadata (level: project or subproject? We might set level: project for summary documents because it's meant to be used at project level now).
   - Dump Bot or a direct function stores it in the Project's context (perhaps in the "Operations" or "Design" segment, or maybe we have a dedicated segment for subproject summaries).
   - Alternatively, the project bot maintains a "Project Master Summary" doc and appends the subproject info to it (like a running record of changes).
8. **Project Bot Notification:** Project Bot is alerted of new subproject output.
   - It might post in project channel: "✅ Subproject LoginFeature completed. Summary: …" to inform the team and stakeholders.
   - It also uses this info for any project-level queries now.
9. **Subproject Archival:**

  - Mark subproject’s tasks and channel as archived (if using Discord archive).
  - Subproject Bot might be deactivated or put in standby.
  - If one-at-a-time subprojects, now project is free to start a new subproject channel. If parallel, this one just stays closed while others continue.

Project to Product Upward Synthesis:

10. **Project Completion Trigger:** Once all subprojects or major deliverables are done for the project, or at a scheduled major milestone.
11. **Project Bot Compiles Project Report:**

- Summarizes overall project objective and to what extent it was achieved.
- Collates subproject results: e.g., “Project WebsiteRedesign delivered: New login, revamped profile page, improved performance by X%. User satisfaction increased from A to B (from survey). Completed under budget by 5%.”
- Includes key insights that might be useful broadly or for future (some of these might be added to design insights library too).
- Possibly includes metrics or before/after comparisons (if any data from timeline or research segments).
- This report should be fairly concise and high-level.

12. **Promotion to Product Vault:**

- The report is stored in the Product’s context. Possibly in a “Project Summaries” folder or appended to a running “Product development log”.
- Mark project as completed in metadata (so queries know it’s done).

13. **Product Bot Notification:**

- It gets the info that Project X is completed.
- It could announce in product channel: “Project WebsiteRedesign finished, delivering features A, B, C. See project report for details.” (which maybe is attached or accessible via command).
- The Product Bot also now knows to include those features in product-level documentation (it might update a list of features or a product manual, if maintained).

Product to Client Upward Synthesis:

14. **Product/Portfolio Update Trigger:** Perhaps at end of a product release or periodically (quarterly).
15. **Product Bot Compiles Product Snapshot:**

- If not already existing, create/update a doc that lists all active and completed projects and their outcomes for that product.

- Summarize current status (e.g., “AlphaApp Version 2.0 released with features from Projects X and Y; next, focusing on user growth.”).
- If the product is ongoing (most are), this might not be a final report but a periodic one.
- If product is finished (like contract ended), then a final summary: all goals achieved and any final metrics.

16. **Promotion to Client Vault:**

- The summary relevant for client consumption is stored in the Client’s context.
- For internal use, it might also remain in product vault, but key is now it’s available for client reporting.

17. **Client Bot Notification:**

- It picks up that there’s a new update on Product for client.
- Prepares to include that in the next client briefing or answers queries accordingly.
- Possibly, if it’s a significant milestone, it could proactively draft a message like “AlphaApp achieved its Q3 goals, ready for client review”.

18. **Client Report Synthesis (if needed):**
    - If multiple products, a final step may be combining all product updates into one client report.
    - The Client Bot or a human can do: basically copy-paste or summarize across products.
    - This might be a manual process with AI assistance because cross-product summarization might need nuance (like prioritizing what to tell).
    - But for completeness, the system could support it: the Client Bot could retrieve each product snapshot and then create an overall summary or slide deck outline.
19. **Archiving at Client/Product Level:**
    - If an engagement ends or a product is retired, mark them archived, move channels to an archive category in Discord maybe.
    - The data stays but is read-only.

Throughout these steps, **human confirmation** is important at junctures:

- After task summary generation, ideally the intern or lead quickly reviews the summary.
- After subproject summary draft, the project lead might review or tweak phrasing, especially if it goes to a broad audience.
- Same for project report if it goes to client.

Our system can streamline creation of these artifacts but should allow edits (which could be done by just editing the message or doc and re-ingesting if needed).

This upward synthesis ensures **continuous integration of knowledge** from detailed to abstract levels, so no insight stays buried at the bottom. Next, we look at the data model for design insights and how they are managed.

5.8 Design Insights Data Model

We structure design insights (both local and global) in a way that they can be easily stored, tagged, and retrieved.

Global Design Insights Library:

- Could be implemented as a special Project or vault, e.g., treat it like a “Global” project under a pseudo client (like an internal knowledge base client).
- Each insight principle is a Markdown note with:
    - **id:** e.g., UXPrinciple001
    - **level:** global-framework
    - **title:** short name of the principle (maybe use the id for title or add a title field in metadata).
    - **tags:** topics like usability, visual, performance, etc. (We can use YAML list or just include as part of content/metadata).
    - **content:** description of the principle, perhaps with examples.
- Alternatively, global insights could be a single structured document, but separate notes allow easier embedding and tagging.
- Possibly maintain an index or TOC listing all principles for reference.

Local Insights:

- For each Project (or sometimes subproject), we could maintain an “Insights & Reflections” document.
    - This might accumulate bullet points of lessons learned throughout the project.
    - Or split by categories if many (design insights, process insights, etc.).
    - It would have metadata like:

```
id: "projX_insights"
level: project
segment: operations (or maybe we define a new segment "insights")
status: final (if recorded after project done, or could be working
throughout)
```

- -
    - Inside, it might list insights with dates or sub-headings by subproject.
- Alternatively, record each insight as an individual item:
    - e.g., after a usability test, the team adds:

```
id: "projX_insight_2025-07-19"
level: project
segment: insights
authors: ["@uxresearcher"]
linked_items: ["task45_summary"]
---
Users strongly preferred the one-click login option, indicating
convenience trumps minor security concerns in this context.
```

- -
    - This is stored like any doc. Multiple such notes accumulate.
    - At project end, they could be compiled or left as is for retrieval when needed.
- We should tag local insights with keywords too (similar to global) because that will help retrieval beyond the project if needed.
    - Perhaps add a tags: field in metadata or just in content "#onboarding", "#mobileUX", etc., to facilitate cross-project searching.

Linking Global and Local:

- Sometimes a local insight might basically be an example of a global principle. We could link them:
    - e.g., local insight "Using fewer form fields improved sign-up rate" is related to global principle "Minimize user input for onboarding".
    - Could add a related_principle: UXPrinciple034 in metadata or link text.
    - Then if someone queries the principle, system might mention the real-case example (though that might be more advanced usage).
- Conversely, if a global principle was contradicted, the local insight might note "Exception to [Principle X]" which is valuable to highlight.

Retrieving Insights:

- We maintain a vector index for insights:
    - Possibly combined global and local insights (with metadata so we can filter if needed).
    - Or separate but can search both sets and present relevant ones.
- When a task or project query is processed, if we want to include insights:
    - We do a semantic search among insights with the query or context.
    - Also possibly filter by relevant domain: e.g., if the task's project is about mobile app, maybe favor insights tagged mobile or from that product's history.
- Could also incorporate a rule base:
    - e.g., if task in domain "UX design", automatically include top 3 UX principles global (though that might be too generic).

    - Better to trust targeted search.

Maintenance of Global Frameworks:

- Over time, might want to update or add new principles.
- Perhaps the design team can directly edit the global MD files or have a UI for it.
- Those changes would just be ingested as any doc (or manually updated out-of-band).
- If there’s a major update (like they decide a certain principle is outdated and remove it), ensure references in system are updated (maybe by id—if a global principle is deprecated, mark it so in metadata).

Data Example:

Global insight note example:

```
---
id: "principle_consistent_icons"
level: global-framework
tags: ["UI", "Consistency", "VisualDesign"]
---
**Principle: Consistent Icons Across Platform**

Ensure that icons for common actions (save, delete, refresh, etc.) are
consistent in design and meaning across all products and screens. This
builds user familiarity and trust.
```

Local insight note example:

```
---
id: "projectX_insight_login_tutorial"
level: project
project: WebsiteRedesign
segment: insights
linked_items: ["task12_summary"]
tags: ["onboarding", "UX"]
---
**Insight:** In Project X, we found that first-time users were confused by
the lack of guidance during sign-up. A short tutorial improved activation
by 25%. This suggests future onboarding should include an interactive
guide.
```

Retrieval would treat these similarly to other docs, but because of small size, they will rank highly if relevant.

Injection conditions:

- For global: if similarity score > threshold or if certain tag exactly matches query tag.
- For local: if query is within same project, lower threshold because it's likely more directly relevant.
- Possibly always present some global design checklist when starting a new design task (even if not asked, just as active context – maybe a curated set like "our 5 design commandments" or so).
- But to not overwhelm, we might only do that if user hasn't interacted much or if it's known the team uses those.

In essence, the data model for insights is not drastically different from other content, just categorized differently and heavily tagged for semantic match. The system then leverages those tags and semantics to bring the right insight at the right time.

5.9 Person Bot Integration Details

This section details how personal vault data interacts with the system and how the Person Bot operates with respect to data flow:

Personal Vault Data Structure:

- Each person's vault might be stored in a folder named after them, or in a database table keyed by user.
- Content types:
  - Journal entries (daily log, maybe dated).
  - Personal notes (could be free-form or categorized by topic).
  - Drafts (things they plan to share but haven't yet).
  - A copy of tasks assigned to them (the person bot can store pointers or text of their tasks for convenience, updated via metadata DB).
- Metadata for personal notes can be simpler:
  - level: person
  - person: @username
  - maybe linked_items if they mention a project.
  - They probably won't have segments; they might use tags in content for themselves.

Synchronization with External (Obsidian):

- If the intern writes in Obsidian locally:
  - Perhaps they use a plugin that commits to a Git repo or some sync which the system can pick up. If they tag a note with something like share: false by default, and set to true when ready.
  - Alternatively, provide an option to manually import a note: like copy paste into the DM with person bot or attach the file.

    - For initial design, probably easier if they write in DM to the bot directly (thus it's immediately captured).
    - Or they can send a file to the person bot which triggers it to store in personal vault.

Personal to Shared Promotion Workflow:

1. User decides to share a personal note with the team.
2. They can either:
    - Use a command: @PersonBot share note "Idea for feature X" to Project Y (segment: research).
    - Or just copy the content into the project's dump channel themselves but likely they'd use the bot to retain metadata properly.
3. Person Bot on "share" command will:
    - Retrieve the note content from personal vault (if it's identified by name or ID).
    - Append or add YAML metadata:
        - Possibly keep original author info (the user) but now add project references.
        - If the note already had frontmatter, merge or override level to project, etc.
        - E.g., from level: person to level: project, with authors preserved.
    - Send this to the Dump Bot or directly to the ingest pipeline.
    - Confirm to user "Shared successfully."
4. The note now appears in the Project vault as a normal doc (with attribution to that user).
    - Maybe the Project Bot announces it: "@alice contributed a note: 'Idea for Feature X' in Research."
5. The Person's personal copy could remain as is, or the person bot could mark it as shared (to avoid editing it further or to note that it's now redundant with the official copy).
    - Could add a link to the shared version or an update in personal vault that "This note was shared on date to Project Y".
    - Possibly tag it as shared or move it to an "Archive/Shared" subfolder in their vault so they know it's taken care of.

Shared to Personal (less obvious but maybe):

- If something happens in a project that a person wants to keep in their personal notes:
    - They can just query person bot for the info anytime.
    - Or have the person bot automatically log things that involve them:
        - For example, if a task they are assigned is completed, maybe person bot can note "Completed Task X on date, outcome Y".
        - Or they could instruct "log this outcome to my diary" and the person bot adds an entry summarizing it.
    - This way they have a personal record of their contributions.

Personal Bot's Use of Shared Context:

- When asked something like “Summarize my notes about Project X from last month”:
    - It will search within the personal vault (notes tagged Project X or mentions of it).
    - When asked “What’s the latest on Project X?”:
    - It might either politely say “Let me check the project data…” and then actually ask the Project Bot or search the project vault.
    - Because the person bot does have access to project vault if the user does. It could perform similar retrieval to project bot but in a simplified way.
    - Possibly we integrate it so that person bot can call other bots or at least query the central index with filters (e.g., project= X and get summary).
    - Then it will answer privately. This is basically letting the user get project info without leaving their DM.

Daily Stand-up / Status generation:

- The person bot can compile what the person did by:
    - Checking tasks completed, tasks in progress (from metadata DB, filter by author and status).
    - Checking personal log entries from that period.
    - Combining them to a short report. Possibly output in a format ready to post in a daily standup channel if they have one, or just for user’s own reflection.
- If integrated with calendar:
    - It can list meetings of the day etc., but we haven’t included calendar integration in scope explicitly.

Edge Cases:

- Person leaves the team: their personal vault might be archived or deleted according to policy (maybe keep for a while or transfer to manager if anything useful).
- Multiple people trying to share similarly named notes:
    - Person bot deals individually per user, so no conflict in IDs because personal content is separate namespace effectively.

Person Bot Implementation Notes:

- Could reuse the same code as other bots but with a context filter to that user.
- Or could be a distinct service focusing on per-user data (maybe simpler just logically partition by user).
- Possibly run in same process just with condition if DM vs channel and user mapping.

Privacy Recheck:

- Person vault likely only accessible by that person’s bot instance.
- Not indexed in global search except if user shares something (then it’s in project index).

- So, e.g., an admin should not be able to ask a bot to search someone's personal notes. Unless obviously there's some override for admin (we likely avoid that to encourage trust).
- If the system is internal and no malicious use by admin, we trust it but may log if admin does search personal vault for transparency.

Integration with Access Control:

- The person bot knows what projects the user is in (maybe from user's roles or from tasks assigned).
- It should gracefully handle if user asks about a project they aren't in. Possibly it can answer if the data is not sensitive? But better to say "You don't have access to that project's details" if it's strictly separated.
- This implies we maintain mapping of user to project access rights. Could derive from who is listed as author or subscriber in project data, or could just use Discord role membership if we assign roles per project.

This integration ensures individuals can leverage the system fully for their own productivity, creating a virtuous cycle: personal improvements lead to contributions back to the shared knowledge, which in turn help others.

5.10 Prototype Data Examples (Metadata & Content)

Below are a few example data pieces showing how they might look with metadata and content, illustrating different scenarios:

1. Example Markdown Document Dump (Project Research Note):

User has an Obsidian template for research notes. They fill in metadata and content, then drop it in the dump channel:

```
---
id: "webredesign_user_research_may2025"
level: project
client: ACME_Corp
product: AlphaApp
project: WebsiteRedesign
segment: research
status: final
nuance_level: medium
authors: ["@alice", "@bob"]
created: "2025-05-20T09:00:00Z"
updated: "2025-05-22T16:30:00Z"
---
```

```
# User Research Summary (May 2025)

**Participants:** 5 existing AlphaApp users (3 mobile, 2 web).

**Goals:** Understand pain points in account creation and profile setup.

**Findings:**
- Onboarding is **confusing**: 3/5 users got stuck on the profile setup
step.
- Users highly requested a **social login** option to skip password
creation.
- The term "AlphaApp ID" was unclear to users; they weren't sure if it's
different from email.

**Implications:** Simplify onboarding flow, possibly add social login and
clarify terminology.

*(Detailed interview notes are in appendix or separate transcripts.)*
```

- This would get stored under ACME_Corp/AlphaApp/WebsiteRedesign/Research/.
- The content shows a summary of research findings. It's marked final and medium nuance (it's a summary, not raw transcripts).
- Active importance might default to active if it's important (could have included that field if we want to ensure it's surfaced).
- Now, when a subproject related to onboarding runs, the Task Bot can pull this summary to inform decisions.

2. Example Meeting Transcript (Subproject Operations):

Suppose a meeting transcript file AlphaApp_WebsiteRedesign_Kickoff.txt is dumped (with a known meeting ID):

```
---
id: "mtg_websiteRedesign_kickoff_2025-04-01"
source_channel: meeting-transcript
level: project
client: ACME_Corp
product: AlphaApp
project: WebsiteRedesign
segment: operations
status: final
nuance_level: high
meeting_ref: "Zoom-2025-04-01-XYZ"
```

```
authors: ["@transcript_bot"]
linked_items: []
created: "2025-04-01T13:00:00Z"
---

**Meeting: Website Redesign Kickoff (April 1, 2025)**

**Participants:** Alice, Bob, Carol, Client_POC

**Summary:** (The bot or user might add a short summary here manually for
quick reference)

*Transcript:*
- Alice: "Welcome everyone, this project aims to refresh the website
UX..."
- Bob: "We should set a timeline, maybe 3 months..."
- Carol: "I will handle research phase in the first month..."
- Client_POC: "Our priority is the signup flow improvement..."
- ...
```

- The Dump Bot might not have all fields if absent but could infer from the file name "WebsiteRedesign".
- It's stored under Project's operations segment (since it's a kickoff meeting).
- nuance_level high because it's raw conversation.
- This would typically be passive context; not auto-included in every query, but available.

3. Example Task Final Artifact (Subproject output):

After Task 5 "Design Login UI" is done, the Task Bot (or user) produces a summary:

```
---
id: "task5_loginUI_design_summary"
level: task
client: ACME_Corp
product: AlphaApp
project: WebsiteRedesign
subproject: LoginFeature
task: "5"
segment: design
status: final
nuance_level: basis
authors: ["@david"]  # intern who did it
linked_items: ["task3_research_summary", "principle_consistent_icons"]
```

```
created: "2025-07-18T17:45:00Z"
---

**Task 5 (Design Login UI) - Summary:**

- Created a new login page design with streamlined input fields.
- **Decisions:** Chose to use only email + password (no username) to
simplify onboarding. Incorporated "Sign in with Google" as an alternative
option.
- **Rationale:** Based on user research, fewer fields and offering social
login address key pain points. (See Task 3 research)
- **Design details:** Used consistent iconography and styling per design
system (applied global principle of consistent icons).
- The design prototype link: [Figma prototype](http://figma.com/...).

*Next Steps:* This design will be reviewed with stakeholders and then move
to implementation in Task 6.
```

- We see it references a prior task's research summary and a global design principle by ID in linked_items (for traceability).
- This is highly basis (decisions and rationale, no extraneous discussion).
- It gets stored in the subproject vault. The subproject bot might incorporate it into the subproject summary.

4. Example Subproject Summary promoted to Project:

After LoginFeature subproject is done, a summary might be created:

```
---
id: "subproj_loginFeature_summary"
level: project   # storing at project level
client: ACME_Corp
product: AlphaApp
project: WebsiteRedesign
segment: operations   # could use operations or a generic summary category
status: final
nuance_level: basis
authors: ["@project_bot", "@alice"]  # maybe bot drafted, Alice approved
linked_items: ["task5_loginUI_design_summary",
"task6_loginBackend_summary", "task7_loginTesting_summary"]
created: "2025-07-20T10:00:00Z"
---
```

```
**Subproject "LoginFeature" Summary (WebsiteRedesign Project):**

*Goal:* Improve the user login and account creation experience.

*Key Outcomes:*
- **New Login UI** implemented (Task 5) – Simplified to one-step email
login, added Google OAuth. Result: expected to reduce drop-off during
signup (pending metrics).
- **Backend Support** added (Task 6) – Created APIs for OAuth and
streamlined account creation logic.
- **User Testing** completed (Task 7) – 10 users tested; 8/10 found the
new process easy, average signup time dropped from 2min to 45s.

*Decisions & Learnings:*
- Dropped username requirement (email alone suffices) – based on research
and to align with modern standards.
- Social login can significantly speed up onboarding for 50% of users
(from test feedback).
- A guided setup after login is still needed (users want to be led through
profile creation – consider for next phase).

*Impact on Project:* This completes the account onboarding improvements.
We expect a 20% increase in new user activation (to be measured in coming
weeks). This subproject addressed the client’s primary concern for Q3.
```

- Stored in project vault, segment “operations” or perhaps we should have a custom “summary” segment. Using operations here just for placement.
- Linked items trace back to each task summary used. If someone wants detail, they can find those tasks.
- Project Bot will use this content for answering overall project queries and it will be part of the project’s basis.

5. Example Global Principle (again, for context linking):

Already gave one in 5.8, but just to see how it stands in data:

```
---
id: "UXPrinciple_Consistency"
level: global-framework
tags: ["UX", "Design", "Consistency"]
---
**Principle: Consistency in Design**
```

```
Ensure consistent layout, terminology, and icon usage across all screens.
Users should not be surprised by changes in style or placement as they
navigate the product. Consistency improves usability and perceived
stability of the app.
```

- When Task 5 was ongoing, the Task Bot pulled this or a part of it when design choices came up, since it matched “icon usage” etc.

These prototypes illustrate how information is formatted and how the pieces link. It helps to see concrete examples when implementing the parsing and retrieval logic.

5.11 GUI Wireframe Description (Optional)

(Since providing an actual image is not feasible here, we describe how a minimal GUI might look.)

Imagine a web dashboard with a clean, two-pane layout:

- **Left Sidebar:** Hierarchy Navigation
    - A tree control listing Clients, expandable to show Products, which expand to Projects, etc.
    - For example:

```
ACME_Corp [client]
  > AlphaApp [product]
     > WebsiteRedesign [project] (active)
        > [Ops]
        > [Research]
        > [Timeline]
        > [Design]
        > LoginFeature [subproject] (active)
            - Task 5: Design Login UI (done)
            - Task 6: Implement Backend (done)
            - Task 7: User Testing (done)
     > MobileAppUpdate [project] (planning)
        > [Ops] ...
        > OnboardingFlow [subproject] (paused)
  > BetaService [product]
     (etc.)
```

-
    -
    - Possibly icons: folder icons for each level, or special icons (e.g., briefcase for Client, box for Product, project icon, etc.).

    - Completed or archived items might be greyed or in an “Archived” subtree.
- **Main Panel:** Details View
    - When a node is selected, this panel shows relevant information:
        - For a Project: Show project description, status, perhaps progress bar (if we can compute, e.g., tasks done/total), and segments.
            - Could have tabs or sections in this view: “Overview”, “Segments”, “Subprojects”, “Team”.
            - Overview might list the project goal, key deadlines, and recent summary.
            - Segments: clicking a segment like Research lists documents in that segment (titles, dates). Clicking a document could open its content in a modal or new window.
            - Subprojects: list each subproject with status (active/completed) and perhaps hyperlink to them.
            - Team: list members on this project (maybe gleaned from authors or assigned roles if any).
        - For a Subproject: Show subproject goal, start date, tasks list.
            - List tasks with their status (checkbox or green/red icon).
            - Perhaps a timeline if dates are involved.
            - A button to “View Summary” which opens the subproject summary content.
        - For a Task: If clicking a specific task, maybe pop up a detail with task description (if any), assignees, and the final artifact content if done (or current status if not done).
            - Could also have a link “Open Discord Thread” to jump to it for discussion (since execution happens in Discord).
        - For a Person: If the UI had a section for team members, selecting a person could show what projects they’re in and their recent notes (if user has rights).
        - Global Insights: We might have a section or a special “Knowledge Base” node where one can browse all global insights or search them.
    - **Search Bar:** Possibly at the top of main panel, allowing quick search for a keyword across the selected scope or entire hierarchy. Results could be filtered by context.
        - E.g., if user searches “login”, it might show results grouped by where found (Task 5 summary, Global principle, User research doc).
        - Clicking a result navigates to that item’s detail.
- **Indicators:**
    - Each level might show an indicator for “active vs archived”. For example, the active subproject might have a green dot.
    - Projects could have a percent complete indicator if we define one (maybe based on tasks or simply phases done).
    - Could highlight if any tasks are overdue (if due dates known).

    - A small icon or color to indicate if there’s unread/new content at a node (if someone added a doc recently, to draw attention).
- **User Interaction from GUI:**
    - Initially read-only:
        - Clicking around should not require confirmation because it’s just viewing.
        - If we allowed actions: maybe a button “Mark task complete” next to a task if it’s not done (which would call some API to finalize it, but we would have to then ask for summary; so maybe not straightforward).
        - Possibly a “Create new Task” button when a subproject is active, which could prompt for a task name and then create a Discord thread via integration, spinning up a task bot.
        - Or “Start new Subproject” in a project view if none active, which could create channel.
        - These actions require connecting the GUI to Discord and our backend - possible but beyond a simple read-only panel, and for v1 we might skip.
- **Access Control in GUI:**
    - The GUI could require login; likely integrate with Discord OAuth for identity, then show only what that user should see.
    - If an intern logs in, they see only their projects, or all internal if no restriction. If a client logs in (if that was allowed), they’d see only their client node and under that only product summaries maybe (we might hide the sub-levels and show a simplified view).
    - We can tailor the content: maybe a client user sees a summary view with no raw documents list.
- **Technology assumption:**
    - This could be a simple React app calling some API that queries our data store (or calls a summarization endpoint for real-time info).
    - Possibly an internal tool not exposed publicly, just for team.

Even without the actual GUI, this description outlines how one could quickly visualize the complex structure and get information without typing commands. It complements the conversational Discord interface by providing a bird’s-eye view.

5.12 Risk Log (Potential Failure Modes & Mitigations)

Finally, we list anticipated risks or failure scenarios and how we plan to mitigate them:

- **R1: Missing or Incorrect Metadata Leading to Misclassification** *Description:* If users forget to include metadata or provide wrong info (e.g., wrong project name), documents could be filed in the wrong place or not at all. This can cause relevant info not to be found when needed. *Mitigation:*
    - Provide clear templates and training to users for using metadata (perhaps even macros or forms to fill it easily).

    - Dump Bot’s interactive query on ambiguous input ensures it rarely files blindly; it will ask for confirmation if uncertain.
    - Implement a periodic audit (maybe a command to list “unclassified items” or allow an admin to see if any docs are sitting without classification).
    - We could also create a default “Unsorted” vault for anything truly unknown and have a process to manually sort those.
- **R2: Context Overload / Irrelevant Context in Prompts** *Description:* The system might pull in too much context, causing prompt stuffing or confusion for the LLM (or the user). For instance, Task Bot might include a large irrelevant document because a keyword matched. *Mitigation:*
    - Tune the retrieval ranking and similarity thresholds carefully. Perhaps limit the number of documents or tokens inserted into any prompt.
    - Use the active/passive flags to include only high-importance context automatically.
    - Continuously refine based on feedback: if users say “that info isn’t relevant”, adjust weighting (e.g., boost more recent documents, or those tagged active).
    - Possibly implement a feedback mechanism: e.g., user can react with an emoji to a bot message that had context, indicating “this was not needed”, and system learns from that if patterns emerge.
- **R3: Information Silos or Out-of-Date Upstream Data** *Description:* If upward synthesis is not done timely, higher-level contexts (project, product) can become outdated, leading to decisions made on stale info. Or if people bypass the process (e.g., don’t finalize tasks properly), knowledge might remain siloed in a closed thread or personal note. *Mitigation:*
    - Encourage a discipline: maybe the Project Bot posts a reminder: “Subproject X finished 3 days ago, no summary created. Please summarize to update project context.”
    - Possibly automate summarization drafts so even if humans forget, the bot can produce something (marked as draft) for review.
    - For tasks, if a thread goes inactive without closure, Task Bot pings assignee after a period: “Do you want me to wrap up a summary for this task?”
    - Also, incorporate review in normal meetings: e.g., a weekly sync where Project Bot prints out what’s new or what’s pending summary, so the team addresses it.
- **R4: Sensitive Info Leakage** *Description:* A bot might accidentally include sensitive details (like an internal cost or a personal note) in a summary meant for a broader audience. For example, client bot might reveal something said in an internal meeting. *Mitigation:*
    - Strict permission checks in code: before outputting any piece of content, cross-reference its permission level vs audience. E.g., client bot filters out anything not explicitly marked client-safe.
    - Possibly maintain a separate sanitized store for client-facing info (so client bot never even sees raw internal docs).
    - Use redaction tags: maybe mark segments of text or entire docs with “confidential” and program bots to remove or summarize vaguely those parts.

    - Have humans review major client-facing outputs (the system could flag anything with potential issues, e.g., if it mentions an internal person's name or an internal-only term, maybe highlight for review).
    - Additionally, test the system with various queries to ensure it doesn't cross boundaries (like try asking a Task Bot about something in another project and see if it refuses correctly).
- **R5: AI Hallucination or Inaccuracy** *Description:* Bots (especially if using LLM for answers) might hallucinate facts or misattribute information if context is missing or ambiguous. For instance, the Task Bot might fabricate a rationale if it doesn't find one. *Mitigation:*
    - Always provide the actual retrieved context to the LLM and encourage it to quote or refer to it rather than invent.
    - Use smaller, more controllable generations for tasks like summarization (maybe fine-tuned models or few-shot prompts that stick to given text).
    - The bots should indicate uncertainty and possibly ask for clarification rather than guess. E.g., if user asks a question and bot isn't sure, better to say "I couldn't find that, do you have more details?" than to make something up.
    - Allow users to trace source: maybe an advanced feature, but the bot could provide reference IDs (like [doc XYZ]) in its answer so the user can verify. This is in an internal context, so it's acceptable to refer to internal docs that way.
    - Continuous testing with known queries to see if answers align with the stored truth.
- **R6: Performance and Scaling Issues** *Description:* As the knowledge base grows, searches might slow down or context assembly might become too large. Also, vector DB might hog memory or CPU if not optimized. *Mitigation:*
    - Monitor performance and optimize indexes (e.g., if some projects are huge, maybe break down their index).
    - Archive older data out of the main index if it's not likely needed (e.g., projects completed a year ago could be in a cold storage that requires explicit search, not part of every similarity query).
    - Use pagination or limits: like only consider the top 100 relevant chunks from vector search instead of scanning thousands.
    - Possibly upgrade infrastructure as needed (because we did choose cost-effective, might need to allocate more memory or a dedicated machine for the DB if hitting limits).
    - If needed, use a cloud service for vector search for large scale (though more cost, so try to avoid until necessary).
    - Implement caching of embeddings and results for frequent queries (e.g., if "summarize project X" is asked often, keep that ready or partially computed).
- **R7: Concurrency and Consistency** *Description:* Two people might update related contexts at the same time (race conditions), or a bot might try to read while writing is happening, causing inconsistencies. E.g., two tasks finishing simultaneously might both trigger upward summary updates. *Mitigation:*

    - Because our team is small and events relatively infrequent, we can handle sequentially in most cases. But in design:
        - Use transactions or locks around database updates (especially for versioning).
        - Make bots aware of events: e.g., if subproject summary is being generated, maybe temporarily lock further tasks from triggering another generation.
        - Or queue such syntheses tasks (like one at a time per project).
    - Also, ensure id uniqueness to avoid collisions if two different dumps choose same id (maybe prefix ids by some context to reduce chance or enforce UUID usage).
    - Worst case, if a conflict occurs (like two separate summaries created), a human might have to reconcile. But that's rare.
- **R8: User Adoption and Manual Effort** *Description:* If the system is too cumbersome (e.g., requiring too much manual metadata or corrections), users might bypass it (not dump docs, not finalize tasks properly) leading to gaps. *Mitigation:*
    - Streamline the UX: maybe provide a Discord slash command to create a new task which automatically sets context rather than requiring them to manually start thread and remember context.
    - Possibly have templates for common actions (like a button or command to output daily log).
    - Education: show the benefit (like demonstrate the bot answering complex questions thanks to stored context, to motivate team to feed it info).
    - Gradual rollout: start by using it on a smaller project to fine-tune processes, incorporate feedback, then expand.
    - Make bots polite and non-intrusive: if they spam or annoy, users might ignore them. So fine-tune how often they intervene.
- **R9: Data Security and Privacy** *Description:* Although internal, storing a lot of data (some possibly sensitive like client info) in one system poses a security risk if not protected. Also personal notes may contain personal data. *Mitigation:*
    - Ensure proper access controls (as discussed) and also secure the storage (encrypt data at rest if needed, secure connections).
    - The server hosting this should be secure, with limited access. Use Discord's existing security for comms plus treat our DB with care (maybe behind VPN or local network).
    - Regular backups but also secure those backups (encrypted).
    - If any personal data (like user info) is stored, comply with any policies (maybe not relevant at small scale, but if this grew, consider GDPR-type compliance if containing user feedback logs etc).
    - Possibly implement a purge for personal data if someone requests (like ability to delete their personal vault easily).
- **R10: Bot Error or Downtime** *Description:* If a bot crashes or goes offline, certain processes halt (e.g., ingestion or assistance). *Mitigation:*

    - Have a watchdog to restart bots or a fallback (someone can do things manually until it’s up).
    - Data is not lost because it’s stored; only real-time help is impacted. So ensure important things (like the knowledge base) can be accessed even if bots are down (e.g., the GUI or even direct file access).
    - Possibly allow manual search in a pinch via a simple search UI or command-line if AI part fails.
    - Rate-limit or throttle bot tasks if needed to avoid overload causing crashes.

This risk log can be expanded as we implement and discover more edge cases, but it covers the most apparent concerns at design time.

---

6. Implementation Roadmap

Finally, to implement this system, we propose a phased approach with milestones aligning with the complexity:

- **Phase 0: Validation and Setup** *Goal:* Confirm key assumptions and prepare the environment. *Tasks:*
    - Discuss with stakeholders to confirm hierarchy rules (e.g., only one active subproject at once? Are clients ever external users on Discord?).
    - Finalize the metadata schema fields needed (maybe do a quick test run with a sample doc).
    - Set up basic infrastructure: a database (or decide file storage structure), a vector DB (install or choose service), and ensure the Whisper server for transcription is ready.
    - Outcome: A clear spec of what will be implemented (perhaps refined by this conversation), and environment ready to code bots.
- **Phase 1: Ingestion & Metadata Parsing Prototype** *Goal:* Build the dump bot and underlying storage mechanism to intake files. *Tasks:*
    - Implement a simple Discord bot that listens to a channel and can parse YAML frontmatter from messages/files.
    - Implement storing to a file system (maybe just save file to disk in correct folder) and a SQLite DB for metadata.
    - Handle a couple of file types (start with Markdown and plain text for simplicity).
    - Test by dropping a few sample files with metadata and verifying they end up in the right place (e.g., check DB entries, file location).
    - If using vector DB, maybe skip embedding in this phase or do a trivial embedding (e.g., use a small model or just store text).
    - Outcome: We can ingest and retrieve a document by structured query (like find by project and title from DB).

- **Phase 2: Basic Context Retrieval & QA Sandbox** *Goal:* Develop the retrieval system and test with manual queries. *Tasks:*
  - Index some documents into a vector store (use a known library).
  - Write a function to retrieve by metadata filter + similarity.
  - Use a small LLM (maybe local model or OpenAI API if allowed) to test generating answers from retrieved context.
  - No complex chain yet, just straightforward Q&A: e.g., ask "What did the research find?" and see if it pulls from a research doc.
  - Tweak similarity thresholds, test multi-turn (maybe keep track of conversation).
  - Outcome: A backend that can answer basic questions using stored context for one level (project maybe), without full bot integration in Discord.
- **Phase 3: Task Thread & Lifecycle Demo** *Goal:* Implement a Task Bot in Discord and demonstrate task context management. *Tasks:*
  - Extend the bot code to allow spawning a Task Bot when a new thread is created. Perhaps in the subproject channel, the main bot monitors new threads and joins them with a TaskBot persona.
  - Implement the Task Bot's ability to listen and respond to mentions/commands in the thread.
  - Include a limited context retrieval: feed it a preset subproject context (maybe manually provided for now, or from DB if Phase 1 done).
  - Let it produce a summary on command. For finalizing task: implement a simple command like !finalize that triggers it to spit out a summary of the thread (maybe using an LLM to summarize the conversation so far).
  - Then see that summary can be caught by Dump Bot (or directly saved).
  - Outcome: End-to-end flow from discussing in a task thread to generating a final note and storing it. This solidifies the interaction pattern.
- **Phase 4: Upward Abstraction & Summarization** *Goal:* Automate the summarization at subproject and project levels. *Tasks:*
  - When a task final artifact is saved, implement logic in Subproject Bot to update a cumulative summary or mark progress.
  - Develop a summarization function (could use LLM) that takes a set of task outputs and produces a subproject summary.
  - Test it with dummy data of multiple tasks to see if it captures key points.
  - Integrate: e.g., a command for project lead like !summarize subproject triggers it.
  - Also implement project-level summarization similarly: take subproject outputs and produce project summary.
  - Check that nuance is being filtered appropriately (maybe by controlling input lengths or instruct the model).
  - If possible, incorporate a few rules: e.g., do not include lines marked certain way, etc.

  - Outcome: The system can produce a multi-level summary with minimal human input (though humans will review).
- **Phase 5: Multi-Bot Orchestration & Permissions** *Goal:* Finalize architecture of multiple bots working together and implement permission gating. *Tasks:*
  - Ensure that each bot (client, product, project, subproject, task) can be instantiated with a context filter. This might be one process with conditional logic, or separate bot instances. Choose an approach (e.g., one Discord bot with many “modes” vs multiple bot accounts).
  - Implement the logic for bots calling each other or fetching cross-level data:
    - E.g., Project Bot retrieving from Product context if needed (which might just be another query to DB with product filter).
  - Implement role-based filtering: check Discord user roles when they ask something. If an intern asks ClientBot something, perhaps that’s not allowed; the bot should say no or give minimal answer.
  - Introduce a permission check in retrieval: e.g., the retrieval function filters out docs with restricted if the user’s role isn’t allowed.
  - Simulate a scenario: have an “Intern” role user ask a borderline question and verify the bot’s response appropriateness.
  - Outcome: Bots operating at all levels in Discord, with context separation and basic permission enforcement. We should be able to have a conversation with any bot and get answers relevant to that level.
- **Phase 6: Design Insights Integration** *Goal:* Ingest some global and local insights and enable their retrieval during tasks. *Tasks:*
  - Populate the global insights vault with a handful of example principles.
  - Tag them and embed them in vector index.
  - Implement in Task Bot’s logic: after initial retrieval, also query the insights index with the same query and see if something comes up to add.
  - Or implement a trigger: if Task topic (maybe derived from thread name or first message) matches certain keywords, fetch relevant principles.
  - Also, allow a command like !insights to list relevant insights if user asks.
  - Test: Start a design task thread about “icons”, see if it surfaces the consistency principle.
  - Record local insight: maybe manually add an insight doc for a project and see if the system can find it for a similar new project question.
  - Outcome: The bots augment answers with design insights when appropriate, and users can query the insights.
- **Phase 7: Person Bot & Personal Space** *Goal:* Implement personal bots for at least one user and test personal note sharing. *Tasks:*
  - Create a DM channel with the bot for a user (or some arrangement, possibly Slack in Discord is DM).

  - Person Bot can be essentially the same code but with logic to filter to that user's stuff. For now, maybe store personal notes in a JSON or a special DB table for simplicity.
  - Implement few commands in DM: like notes list, note add "text", share "note id" to project X.
  - Or make it natural: user says "Remember X" and it stores, user says "What about X?" and it searches.
  - Also feed the Person Bot with tasks from metadata: find tasks where authors include that user and allow them to query "my tasks".
  - Test sharing: have the person bot share a note to Dump Bot with metadata and see it appear in project.
  - Outcome: A basic functioning personal assistant that can store and retrieve notes and interface with the main system.
- **Phase 8: User Interface & Visualization** (Optional/Parallel) *Goal:* Provide a simple UI to visualize hierarchy and maybe do basic queries. *Tasks:*
  - If time permits, create a small web page showing the hierarchy (maybe static using data from DB).
  - Or generate an export (like a markdown or HTML report of the project statuses).
  - This isn't core to the AI but helps demonstration and usage.
  - Outcome: A prototype UI or even a well-organized set of markdown files in a published site (like an exported Obsidian vault) that people can use to navigate context.
- **Phase 9: Testing & Refinement** *Goal:* Rigorously test scenarios and refine system behavior. *Tasks:*
  - Create test cases for each bot: e.g., ask project bot something that should retrieve multiple segments and see if it does well.
  - Simulate simultaneous actions: finalize two tasks quickly and ensure the system handles both (no lost updates).
  - Security testing: try to access something restricted with a lower permission user to ensure it blocks.
  - Gather feedback from initial users (maybe have one or two team members try using it in daily routine).
  - Refine retrieval prompts, fix any incorrect citations or context mixing.
  - Possibly adjust LLM usage (maybe test with an open-source model vs a closed one to gauge quality).
  - Outcome: A polished system ready for broader use.
- **Phase 10: Documentation & Training** *Goal:* Document how to use the system and train the team. *Tasks:*
  - Write a user guide (how to format a note, how to ask bots questions, etc.).
  - Also technical docs for maintaining the system (how to add a new project, how to update global insights, etc.).

- Conduct a workshop or demo for the interns/team to get them on board.
- Outcome: Users comfortable and system maintainers have instructions for common tasks.

Each phase builds on previous ones. We expect Phase 1-3 to produce a usable skeleton that proves the concept. Phase 4-7 add the advanced features (summaries, insight injection, personal bots). The timeline might be spread over several weeks, adjusting based on how fast we iterate and issues uncovered. This roadmap ensures we tackle core functionality first and layer complexity gradually.

7. Open Decision Points & Assumptions

During the design, we identified some questions that require further clarification or might be decided during implementation:

- **D1: Single vs Multiple Active Subprojects:** The assumption was one active subproject per project at a time. If that's relaxed, we need to handle parallel subprojects (multiple channels). Our design can handle it (since context vaults are per subproject anyway). The main difference is project bot summarizing multiple concurrently. We should confirm if concurrency will happen or if the team prefers one focus at a time.
- **D2: Meeting Transcripts Tagging:** Will transcripts always come with some hint (like posted in a relevant channel or named appropriately)? If not, do we enforce that someone has to add metadata after transcription? Possibly we could integrate the transcription service to tag by calendar event which has project info. Clarification with team on how they schedule meetings can help optimize auto-classification.
- **D3: Human Approval for Upward Synthesis:** To what extent do we automate summarization vs require manual confirmation? E.g., should the project bot automatically publish a project summary to the product vault, or should it prepare a draft for a human to review? Current leaning: always allow human to edit, but maybe auto-publish internal ones and only require review for external/client.
- **D4: Storage Backend Choice:** The design allows either a heavy use of the file system (with maybe Git) or a proper database for content. We assumed file + SQLite for simplicity. If the organization is comfortable, that might be fine. If they want something like Notion or Confluence integration, or a cloud DB, that could shift approach. We'll proceed with local storage, but keep in mind how to migrate if needed.
- **D5: Knowledge Graph Use:** We opted not to explicitly build a graph DB. But if relationships become complex (like linking insights to principles, tasks to requirements), at what point does a graph DB make sense? Possibly not needed now, but it's an idea for future if queries like "show me all tasks that influenced a design principle" become important.
- **D6: LLM Autonomy Limits:** The bots currently don't do any action without user prompting except summarization triggers. Do we want bots to ever create tasks on their

own or re-organize content? Likely not without human input. We'll keep them reactive not proactive beyond reminders.

- **D7: Embedding/Vector DB vs Alternatives:** We assume RAG is needed. Could a simpler keyword search suffice for now (given moderate data)? Possibly at first, yes. But long term, vector search is beneficial for semantic matching. We might start with basic search and only enable vector when data grows or we have the bandwidth. It's a decision to weigh complexity vs immediate need.
- **D8: Tooling the Bots (ReAct usage):** Implementation might start with direct retrieval injection. If that proves insufficient (some answers need multi-step logic), we might incorporate a ReAct agent approach. We'll likely implement simpler pipeline first and only add complexity if clearly needed.
- **D9: GUI Scope:** Will the GUI be actually built or do we rely on Discord only initially? If time/resources are short, we might skip GUI until basic system is stable. Or we might do a read-only file-based report (like publish an Obsidian vault periodically as HTML, which could serve similar purpose).
- **D10: ID Strategy:** Right now, we assume unique IDs but format not set. Should it be human-readable (like "project_taskname_date") or just a UUID for every item and then relationships define location? Human-readable helps debugging and manual browsing, but might collide or become lengthy. We might use a composite scheme: e.g., prefix with project code, etc. We'll decide a format and be consistent (maybe use a short name for each entity plus a sequence).
- **D11: Integration with Project Management Tools:** Out of scope, but if the team uses Jira or Trello, do we need to tie in? Currently, the plan is to manage tasks within Discord (with threads). That should suffice. If they insist on using an external tool for tasks, we'd have to integrate those updates into the context, which is more complexity. Likely we stick to Discord-centric workflow.
- **D12: LLM Model Choices and Costs:** Which model to use for summarization and answering? If using OpenAI, cost might be an issue for heavy usage. We might try an open source (like GPT-J, Llama2, etc.) on our server for most tasks, using OpenAI only for complex summarizations if needed. We'll test smaller models to see if quality is acceptable. This decision affects infrastructure (if on-device, ensure we have GPU or enough CPU).
- **D13: Evolution of Knowledge Base:** Over months, the knowledge base can become large and possibly messy. We need a plan for maintenance: archiving old projects (maybe moving their vector entries out of main index), cleaning up outdated insights (if global principles change, do we remove the old or keep with deprecated tag?). This is not urgent now but should be flagged for future operational health.
- **D14: Multi-client separation:** If we eventually have multiple clients with possibly separate teams, we may need to partition data and ensure absolutely no cross-over. Right now it's one org handling multiple clients, presumably by same team, so it's okay if data is stored together with tags. If that changes, might need to spin separate instances or ensure stricter isolation in code/DB.

- **D15: Onboarding new members:** Person bots and such are great but require setup. We should decide how to easily initialize a person bot for a new intern (maybe auto when they join a certain Discord role).
- **D16: Error Recovery and Logging:** We should decide how bots log their actions (maybe a hidden channel for system logs or a file). This is minor but helps debugging when something goes wrong.

We'll need to address these decisions with the team or as implementation details. They don't hinder initial development but are noted for resolution.

8. Evaluation Criteria & Success Factors

We revisit what a successful solution should achieve, to ensure our design meets these criteria:

- **Comprehensive Context without Overload:** Our design explicitly filters and abstracts context at each level. We provide full nuance at task-level and basis at high levels. This should preserve detail where needed (you can always drill down via linked_items if necessary) but keep noise out of summaries. Success means users seldom complain of "too much irrelevant info" at higher levels, and also don't find "lack of info" at task level.
- **Fast, Accurate Retrieval:** With metadata indexing and vector search, any query to a bot should return relevant info within a second or two (depending on LLM response time). We'll measure: e.g., ask a factual question that is in the knowledge base and see if bot answers correctly and cites the right source. Also, anecdotal feedback: team members find they get answers more quickly than searching through docs manually.
- **Up-to-date Knowledge:** If a task finished yesterday, that outcome should already reflect in today's project answers. Our life-cycle triggers ensure immediate propagation. We'll verify by scenario: complete a subproject, then ask project bot for status, expecting it to mention the subproject results. If we see lag, that's an issue. The GUI's last-updated indicators can also highlight if something hasn't been rolled up (if timeline shows months old after tasks done, problem).
- **Traceability:** If an answer or summary is given, users can trace back (via references, or by asking follow-up for detail) to the source. For internal use, bots might even list related doc IDs. We'll consider it success if, for any piece of info a bot provides, the team can find its origin easily either by asking or by looking at linked_items. If not, trust might erode.
- **User Adoption & Efficiency:** Are team members actually using the bots and do they feel it improves their workflow? We'll gather feedback after a trial. If interns say the Task Bot gave them relevant docs that saved time, or PM says the Project Bot's summary saved them writing a status email, those are wins. If people bypass the system (like still searching manually or not dumping docs), then something's off either in UX or trust.
- **Access Control Compliance:** No incidents of info leaking where it shouldn't. e.g., if a client is added to a channel and they only see what they're meant to. We'll test by intentionally having some restricted content in a project and ensure the client bot

summary doesn't include it. Also maybe do a mock scenario where an intern not on a project tries to query that project via their person bot and ensure denial or empty result.

- **Design Insight Utilization:** A qualitative measure: do the design insights actually get used and help decisions? Perhaps after a few tasks, review transcripts to see if global principles were surfaced and if team found them useful (did they follow them or discuss them?). If those remain unused, maybe retrieval needs adjustment (or principles need refining).
- **System Fitting Discord Workflow:** Evaluate how well the integration feels: Are commands intuitive? Do bots speak at the right times? We want minimal friction. If users find bots spammy, we'll adjust. If they forget to use bots, maybe we need to remind or demonstrate more.
- **Maintainability:** Internally, can we maintain this easily? For instance, adding a new project or updating a global insight is straightforward (just add a file and it's picked up). If it requires dev intervention for every new thing, it's not ideal. We'll consider success if after initial dev, the PM or team can add content without needing a programmer (the metadata approach supports that).
- **Scalability for our needs:** While large scale isn't immediate, we should ensure after a few months, performance doesn't degrade. We will monitor DB size, index size. If things remain snappy with, say, hundreds of notes and transcripts, good. If we see slowdown, plan to optimize.
- **Explainability to Team:** The team should understand at least conceptually how info flows. If they get confused ("I updated a doc, why didn't product bot mention it?"), then either documentation or design might need tweaking (maybe they didn't finalize status etc.). We measure success by few user errors in using the system and if they do something wrong, the system clearly prompts them how to fix (like Dump Bot asking for metadata).
- **Flexibility:** If tomorrow they decide to add a new segment type or another level (like splitting subproject into "milestones"), can we adapt without redoing everything? Our design is generic with metadata, so adding a segment is trivial. Adding another level might need code changes. But hopefully the structure is flexible enough for foreseeable changes. We'll consider it a success if small changes in process (like wanting to treat "Milestone" as alias for subproject) can be handled with config or minor tweaks.

In conclusion, the design we provided is elaborate but addresses the needs described. It emphasizes structured knowledge management, context-appropriate AI assistance, and knowledge reuse through insights, all integrated in the team's familiar Discord environment. By following this plan, the team should be able to streamline their workflows, ensure important information is never lost, and leverage AI to reduce tedious work in managing context and generating outputs. The success will ultimately show in more efficient task execution, better continuity across work phases, and consistent quality of outputs thanks to the preservation of knowledge and application of best practices.