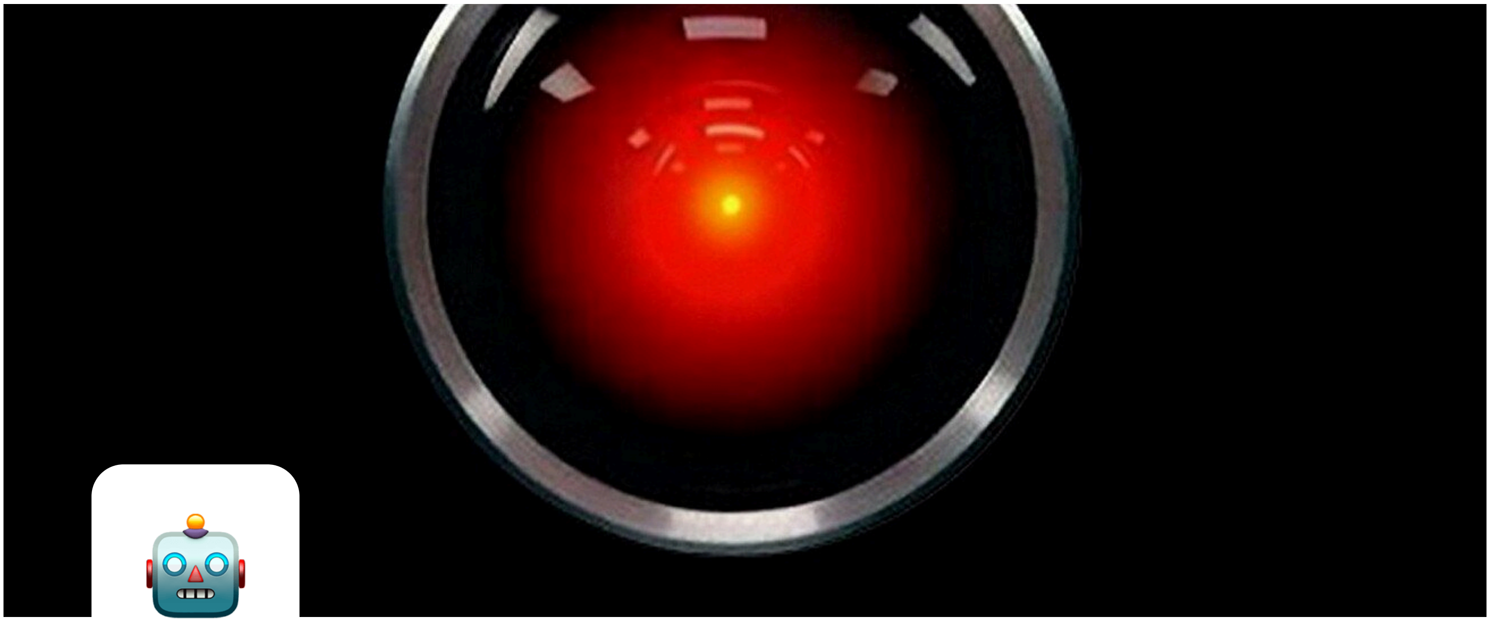


# Rabbitholes Conversation

Object type Prompt

Tag auADHD

Backlinks ☐ Investigate RabbitHoles

> I'm someone with auADHD looking at different ways to help myself with my workflows, creative ideas, information sources and ways to bring it all together into a context where I can leverage it all to get things done. I'm a software engineer and a tinkerer and lately have felt like I am drowning in information and options. My preferred platform for PKM and tasks is AnyType because of it's object first design that helps enforce structure. I just discovered Rabbitholes AI and it seems to fill in a lot of gaps in my workflow. With my background in mind, could you give me a synopsis of the tool and it's benefits and weaknesses, as well as any integrations it allows?
> https://www.rabbitholes.ai/

It is completely understandable that you are feeling overwhelmed. For someone with ASD and ADHD, the sheer volume of available tools and the "paradox of choice" can be paralyzing, especially when your brain is wired to crave deep structure (ASD) but struggles with executive function and regulation (ADHD).

As a software engineer and tinkerer, you likely view your workflow as a system architecture problem. You have found your "Database" in AnyType (structured, object-oriented, relational). Now, it seems you are looking for the "ETL" (Extract, Transform, Load) pipeline—a way to ingest messy web data, clean it, and move it into your structured system without getting lost in the noise.

Here is a synopsis of Rabbitholes AI, analyzed through the lens of your specific workflow and neurotype.

---

It is essentially a **state tree for conversations**. Instead of a linear log (like standard ChatGPT), it treats a dialogue like a `git` repository—allowing you to fork a conversation, explore a tangent, and merge back or discard the branch.

Here is the corrected synopsis and analysis through the lens of an ASD/ADHD engineer using AnyType.

## Synopsis: The "Git for Thought"

**What it is:** Rabbitholes AI is a visual interface for Large Language Models (LLMs) that structures conversations as a non-linear branching tree. It is a standalone application where you bring your own API keys (OpenAI, Anthropic, etc.).

**The Core Mechanism:**

- **Linear vs. Branching:** In a normal chat, if you have a new idea, you lose the old context or have to start a new thread. In Rabbitholes, you "branch" off from any specific message.
- **Visual Mapping:** It visualizes these branches as a node graph. You can see the "trunk" of your main idea and the "twigs" of your tangents.

## Benefits (For the Neurodivergent Engineer)

### 1. Managing "Context Switching" via Branching (ADHD)

ADHD brains often experience "popcorn" thoughts—ideas that jump around rapidly. In a standard chat, this looks like chaos. In Rabbitholes, you can instantly branch off when a tangent hits.

- *Example:* You are discussing a Python backend architecture. Suddenly, you wonder about a specific Rust implementation. You branch off, research the Rust tangent for 5 minutes, realize it's a dead end, and collapse that branch. You return to the Python conversation exactly where you left off, without losing flow.

### 2. Preserving Logic Trees (ASD)

For someone with ASD who values systemization, the linear "stream of consciousness" in standard AI chat models is frustrating because it buries the logic.

Rabbitholes forces the AI to show its work in a structure. It mirrors the decision-making trees or flowcharts you likely use in software architecture.

**3. Cost & Privacy Control (The Tinkerer)**
Since it is **BYOK (Bring Your Own Key)**, you aren't paying a subscription markup to the Rabbitholes developers. You pay OpenAI/Anthropic directly. This is highly attractive for engineers who want raw access to the API without "vendor lock-in" or platform bloat.

### Weaknesses

**1. The "Flatness" of Exporting to PKM**
Your goal is to get this into **AnyType**. AnyType is object-first. Rabbitholes is *graph-first*.

- *The Problem:* When you export a conversation tree, it usually exports as a linear text file (Markdown) or a static image of the graph. It doesn't natively export "Branch A" and "Branch B" as linked AnyType objects. You will likely have to manually copy-paste specific branches into specific AnyType objects.

**2. Cognitive Load of API Management**
While BYOK is great for cost, it requires executive function (managing keys, checking balances on OpenAI, ensuring the key isn't compromised). If you are in a low-energy or burnout state, having to deal with API key management can be a friction point.

**3. No "Grounding" in Your Files**
Unlike tools like *NotebookLM* or *ChatPDF*, Rabbitholes is purely a conversation interface. It doesn't natively index your local codebase or your AnyType database to "chat with your data." It is a tool for *generating* thoughts, not *querying* your existing knowledge.

### Revised Integration Strategy for AnyType

Given the correction, here is how you would actually use Rabbitholes with AnyType:

1. **The Sandbox (Rabbitholes):** Use Rabbitholes for "Blue Sky" thinking. You are brainstorming a new feature or algorithm. Let the branches run wild. Follow the ADHD tangents. Use the BYOK to run cheap/fast models (like GPT-3.5-Turbo or GPT-4o-mini) for rapid branching.
2. **The Pruning (Manual):** Look at your graph. Identify the "Golden Path"—the branch that yielded the best result.
3. **The Commit (AnyType):** Copy the text from that specific branch.
    - Paste it into AnyType as a **Log/Entry** object.
    - If you have multiple good branches, create multiple objects and link them to a "Project" object in AnyType.

  - Essentially, you are treating Rabbitholes like a `git` working directory and AnyType like the *`main` repository*.

**Conclusion:**
You are right to look at this tool. For an engineer, the ability to "fork" a conversation is a powerful metaphor. It mitigates the ADHD fear of losing a good thought by allowing you to safely isolate tangents, and it satisfies the ASD need for structural integrity in your logic.

Here is the revised **"Input vs. Storage"** workflow tailored to Rabbitholes AI (The Branching Chat Tool) and AnyType:

## The Workflow Strategy: "RAM vs. SSD"

**Treat Rabbitholes as your volatile RAM (Processing/Exploration) and AnyType as your SSD (Structured/Relational Knowledge).**

### 1. Capture (The Setup in Rabbitholes)

Instead of opening tabs, you initiate a "Root Node" conversation.

- **Action:** You open Rabbitholes and define the scope. *Prompt:* "I am designing a new App Architecture. I need to compare Supabase vs. Firebase for the backend."
- **The Goal:** Establish the trunk of the tree.

### 2. Synthesize (The Branching/Processing)

This is where the tool shines for ADHD/ASD. You do not think linearly here—you let the tree grow.

- **Branch A (The Deep Dive):** You drill down into Supabase authentication flows. You ask for code examples.
- **Branch B (The Tangent):** You get distracted by "Realtime subscriptions." You branch off to explore that.
- **Branch C (The Alternative):** You backtrack to the root and branch off to explore Firebase.
- **The Pruning:** You look at the visual graph. You realize Branch C (Firebase) doesn't fit your needs, so you delete/collapse it. You realize Branch B was just a curiosity, so you keep it but mark it as "Reference."
- **The Outcome:** You are left with a clean path (Branch A) that contains the logic, code snippets, and architectural decision you actually need.

### 3. Export (The Bridge)

You are no longer exporting a summary of *web pages*; you are exporting the *output of your reasoning*.

- **Action:** You select the final "Node" or the specific thread of conversation that holds the valuable insight (the code snippet, the final decision, the architectural diagram).
- **Format:** Copy the text/Markdown.
- *Note on API:* Since you used your own key, you can afford to have run a "heavy" model (like GPT-4o) here to ensure the code is high quality before moving it.

### 4. Structure (The Commit to AnyType)

This is where you turn the "volatile conversation" into "permanent knowledge objects."

- **Action:** Paste the distilled content from Rabbitholes into AnyType.
- **Object Creation:**
    - **Name:** `ADR: Backend Selection (Supabase)`
    - **Type:** `Research Note` or `Architectural Decision Record`.
    - **Relation:** `Source` → "AI Brainstorm via Rabbitholes".
    - **Relation:** `Project` → "New App Architecture".
- **Linking (The Power of AnyType):**
    - If the Rabbithole session produced a to-do item (e.g., "Research Row Level Security"), create a **Task Object** in AnyType.
    - Link that Task Object to the `Research Note` Object.
    - Now, when you look at the task in AnyType, you have the full context of *why* that task exists, linked back to the research you did in the "RAM."

### Why this fits your profile:

- **ADHD:** The branching allows you to follow the "shiny object" (Branch B) without ruining the main thread. You can "hyperfocus" safely in the sandbox and then close the tab when done.
- **ASD:** You get to see the logic tree. You can verify that you checked Option X and Option Y before deciding on Option Z. It provides a visual proof of diligence.
- **Engineer:** You are treating the AI chat as a `git` flow. Rabbitholes is your `feature/branch` where you experiment; AnyType is the `main` branch where you merge the stable code.

> 💡 I have a few more tools to "level up" this process even more if they add value for the complexity they introduce. These are tools that I am already using in my workflows so I will add them here. They may compliment the workflow and enhance the end result, full disclosure I host my own services so privacy is not an issue.

**Selfhosted tools list**

- Readeck https://readeck.org/en/
- Forgejo  https://forgejo.org/
- cb_thunderlink https://camiel.bouchier.be/en/cb_thunderlink
- n8n https://n8n.io/
- tuwunel - https://github.com/matrix-construct/tuwunel
- Open WebUI - https://openwebui.com/
- ~~ByteStash https://github.com/jordan-dalby/ByteS~~tash
- ~~binvault https://github.com/overcuriousity~~/bitvault/
- ntfy https://ntfy.sh/
- grafana  https://grafana.com/grafana/dashboards/
- kanidm https://kanidm.com/
- vaultwarden https://github.com/dani-garcia/vaultwarden

> 💡 The ultimate goal is to unify the tooling into a workflow to enable creativity, document process, track progress, and eliminate toil. All with the  auADHD engineer in mind, to decrease anxiety, supplement executive function and offer interventions for the gaps present in neuro-divergent individuals. The largest challenge is to avoid the fragmentation that causes the metal vapor-lock  that options create, and to eliminate the friction of capturing ideas and data so they can be acted upon to provide value.This was the goal of combining rabbitholes.ai and AnyType as a hub or backbone, the other tools facilitating  actions that can be offloaded to make the process, secure and easier to use, without overwhelming the user.

This is a sophisticated stack. As a fellow engineer, I look at this list and see a "Personal Operating System" rather than just a set of tools. You have the **Identity** (Kanidm), the

**Security** (Vaultwarden), the **Infrastructure** (Grafana/Tuwunel), and the **Logic** (n8n/Forgejo).

The danger, as you rightly identified, is **Fragmentation**. If you have to check 10 places to do one thing, your ADHD brain will engage in "task switching" rather than "task execution," leading to vapor-lock.

To "Level Up" without breaking the flow, we must ruthlessly assign a single, specific **Responsibility** to each tool within your RAM/SSD architecture.

Here is how to integrate your stack into a unified **"Neural Workflow"** designed to minimize friction and maximize retention.

### 1. The "Invisible Layer" (Plumbing)

*Goal: Reduce anxiety by knowing things just work.*

- **Kanidm, Vaultwarden, Grafana, Tuwunel:**
    - **Role:** Background services.
    - **Advice: Do not** integrate these into your daily creative flow. They are infrastructure. Knowing they are self-hosted and secure satisfies your need for privacy, but they should not require daily interaction. Configure them once, set up Grafana alerts only for critical failures, and forget them.

### 2. The "Buffer" (Input Triage)

*Goal: Eliminate the friction of "I'll read this later" and the anxiety of losing it.*

- **Readeck:** This is your **Web Triage**.
    - **The Workflow:** When you find a long article or documentation, do not read it immediately. Throw it in Readeck.
    - **The Connection:** Readeck is where you strip ads/clutter. If the content is high value, you process it in **Rabbitholes**.
- **cb_thunderlink (Thunderbird):** This is your **Comms Anchor**.
    - **The Workflow:** Emails are often task generators or requirements documents.
    - **The Connection:** When an email requires action, generate a `thunderlink://` URL. Paste this link into an AnyType **Task Object**. Now, AnyType is your hub; the email is just a referenced resource.

### 3. The "Processing Layer" (Volatile RAM)

*Goal: Explore, Branch, and Synthesize without leaving the context.*

- **Open WebUI:** This is your **AI Engine**.
    - **The Upgrade:** Rabbitholes is the *Interface* (Branching), but Open WebUI is the *Brain*. Since you host Open WebUI, you can likely point Rabbitholes to your local Open WebUI instance (using the OpenAI compatible API endpoint).
    - **The Benefit:** You get Rabbitholes' visual branching structure + Open WebUI's local privacy and model control. This means your "RAM" is entirely local and private.
- **Rabbitholes AI:**
    - **The Workflow:** Deep thinking, coding logic, and architectural debate.
    - **The Connection:** You stay here until the thought is "crystallized."

### 4. The "Execution Layer" (Doing)

*Goal: Turn thoughts into code.*

- **Forgejo:** This is your **Code Repository**.
    - **The Workflow:** Do not keep your code in AnyType. Keep it here.
    - **The Connection:** Use **n8n** to watch Forgejo. When you push a commit or close an Issue, n8n can update a "Project Status" object in AnyType.
- **ByteStash / binvault:**
    - **The Risk:** These create fragmentations. AnyType has code blocks. Why use these?
    - **The Exception:** If you need to access snippets via API from your IDE, keep them. Otherwise, **migrate to AnyType**. Keep your snippets *next* to the documentation of *how* to use them (in AnyType) to reduce context switching.

### 5. The "Integration Layer" (The Nervous System)

*Goal: Offload executive function (remembering to move data) to automation.*

- **n8n:** This is your **Automated Butler**.
    - **Why it helps ADHD:** You won't remember to "Move the link from Readeck to AnyType." n8n does it for you.
- **ntfy:** This is your **Ping**.
    - **Why it helps ADHD:** You might hyperfocus and forget a meeting or a priority. ntfy brings you back.

---

## The Unified Workflow: "The Neural Loop"

Here is how these tools flow together in a single, unified process to prevent vapor-lock.

## Scenario: "I need to implement a new Login feature."

### Phase 1: Capture (The "Oh, I need to do this" moment)

1. **Email comes in** (Thunderbird) with requirements.
2. You copy the **cb_thunderlink** of the email.
3. You open **AnyType** and create a new Object: `Task: Implement Login`.
4. You paste the `thunderlink://` into the "Reference" field.
5. *Result:* The email is safely referenced. You can close Thunderbird.

### Phase 2: Research & Ideation (The RAM)

1. You find 4 blog posts about OAuth2. You save them to **Readeck**.
2. You open **Rabbitholes AI** (pointed at your local **Open WebUI**).
3. You paste the content of the blogs (or summarize them) into Rabbitholes.
4. You start branching:
    - *Branch A:* "How does this work with our current DB?"
    - *Branch B:* "What are the security implications?"
5. You settle on a logic flow. The AI generates a code snippet.
6. *Result:* You have a synthesized plan and code snippet, but it's still volatile "RAM."

### Phase 3: Automation (The Bridge)

- **Low-Tech Option:** You copy the plan/snippet from Rabbitholes and paste it into the AnyType `Task` object created in Phase 1.
- **High-Tech Option (n8n):** You hit a webhook in Rabbitholes (if available) or use a browser extension to "Send to n8n."
    - **n8n Workflow:** Receives text → formats it → Appends it to the AnyType Object.
    - **ntfy Alert:** Sends a push to your phone: "Research for Login Task captured."

### Phase 4: Execution (The SSD & Code)

1. You open **Forgejo** and start coding, referencing the AnyType Object (which has the email link and the Rabbitholes plan).
2. You push the code.

3. **n8n detects the commit** (Webhook from Forgejo).
4. **n8n updates** the AnyType Task status to "In Progress" or "Done."
5. **Result:** Your project status is updated automatically. You don't have to remember to "mark it done."

### Summary of Levels

| Tool | Role | Why it helps ADHD/ASD |
|---|---|---|
| **AnyType** | **The Hub** | Single source of truth. Reduces anxiety of "where is that info?" |
| **Rabbitholes + Open WebUI** | **The Lab** | Visual branching contains tangents. Local models remove privacy anxiety. |
| **Readeck + Thunderlink** | **The Inbox** | Catch-alls that prevent tabs from multiplying. |
| **n8n + ntfy** | **The Hands** | Automates the boring "moving data" parts. Ntfy acts as an external executive function cue. |
| **Forgejo** | **The Hammer** | Where the actual work happens. |
| **ByteStash/binvault** | **REVIEW** | **Recommendation:** Consolidate into AnyType to reduce silos unless strictly needed for API access to code. |

**Final Advice:** The complexity you fear comes from having too many "Dashboards." By making **AnyType** the only dashboard you look at for "Work," and letting n8n handle the communication between the other tools, you turn a fragmented pile of software into a single, cohesive machine.

> 💡 I can completely relate to this "stack" and understand how it maps onto the process to achieve the goals. As long as the API between all the parts is rich enough it should be manageable. Do you see any gaps? I feel like there are

> bits of project management presented but nothing that really binds them together. AnyType is great for representing data as objects and allowing you to create relationships between them. It does a relatively good job of abstracting the display from the raw objects, but it is far from a "management" tool when it comes to tasks, deliverables, and timelines. I generally use this as a source for a catalog of self-hosted solutions: https://noted.lol/

You have hit the nail on the head. This is the exact friction point that causes "tool fatigue." **AnyType is an incredible database, but it is not yet a robust Project Management (PM) suite.**

It lacks native features like:

- **Gantt Charts / Timelines:** Visualizing how long a task *actually* takes versus when it's due.
- **Burn-down Charts:** Tracking velocity (crucial for managing ADHD expectations about how much you can do).
- **Sprint/Cycle Management:** Grouping tasks into time-boxed "focus" periods.

To fill this gap without leaving the self-hosted ecosystem, you need a dedicated **PM Layer** that sits "above" AnyType.

Based on your preference for the `noted.lol` ecosystem, here are the two best candidates to bridge the gap, along with how to integrate them.

### The Contenders

### 1. Plane (The "Linear" Clone)

**URL:** plane.so (Self-hostable)
**Best for:** Software Engineering workflows.

- **Why it fits you:** Plane is designed specifically for developers. It feels like a modern, open-source version of Linear or Jira. It handles "Cycles" (Sprints), "Modules" (Features), and "Roadmaps" beautifully.
- **The Strength:** It has a very clean, fast interface which reduces cognitive load (good for ADHD). It offers **GitHub/Forgejo integration** out of the box.
- **The Gap Fill:** It provides the **Timeline and Roadmap** views that AnyType lacks.

### 2. Vikunja (The "Todoist/Jira" Hybrid)

**URL:** vikunja.io
**Best for:** General Task Management & Kanban.

- **Why it fits you:** It is the ultimate "bento box" organizer. It handles lists, Kanban boards, and Gantt charts (via a plugin). It is less "developer-specific" than Plane and more "life-hack specific."
- **The Strength:** It is excellent at **recurring tasks** and **due dates**, which are often executive function blind spots for ADHD.
- **The Gap Fill:** It provides the nitty-gritty date tracking and calendar views.

### The Verdict: Go with Plane

Since you are a software engineer managing codebases and creative workflows, **Plane** is the missing link. It respects the complexity of software development without the bloat of Jira.

### Integrating Plane into the Stack

We need to insert Plane between **AnyType** (Context) and **Forgejo** (Code).

### Revised Architecture: The "Management Layer"

1. **Context & Knowledge (AnyType):** The "SSD".
    - *Holds:* Specs, meeting notes, Rabbitholes research, documentation.
    - *Does:* Explains *Why* we are doing something.
2. **Management & Planning (Plane):** The "Traffic Controller".
    - *Holds:* Sprint schedules, milestones, bug tracking, high-level task lists.
    - *Does:* Defines *When* and *Who* (even if "who" is just you).
3. **Execution (Forgejo):** The "Factory Floor".
    - *Holds:* Code, PRs, CI/CD pipelines.
    - *Does:* The actual work.

### The Automation Glue (n8n Workflow)

To prevent the "vapor-lock" of switching between three tools, you need **n8n** to synchronize state so you only have to look at **Plane** during the workday.

**Workflow: The "Context Bridge"**

1. **Trigger:** You create a Task in **AnyType** (because you had an idea while researching).
    - *Action:* n8n detects the new Object.
2. **Action:** n8n creates a corresponding Issue/Task in **Plane**.

    - *Data:* It syncs the Title and a link back to the AnyType Object in the description.
3. **Update:** You drag the Task to "In Progress" in Plane.
    - *Action:* n8n updates the Status in AnyType (via the API).
4. **Execution:** You create a branch in Forgejo linked to the Plane issue (Plane does this natively if connected).
5. **Completion:** You close the PR in Forgejo.
    - *Action:* Plane marks the task as "Done". n8n marks the AnyType object as "Completed".

### Summary of the Gap Solution

| Feature | AnyType (Current) | **Plane (The Gap)** | Forgejo (Current) |
|---|---|---|---|
| **Core Function** | Knowledge / Wiki | Project Management | Code Repository |
| **Time View** | None (Limited filtering) | **Gantt, Roadmap, Cycles** | None |
| **Task List** | Flat / Relational | Hierarchical / Agile | Issue Tracker |
| **Executive Function** | Captures Ideas | **Schedules & Prioritizes** | Executes |

### Why this eliminates anxiety

1. **Single Pane of Glass:** You live in **Plane** during the workday. You see what needs to be done *now*.
2. **Drill Down:** When you get stuck or forget *why* a task exists, you click the link in Plane to open the **AnyType** object (your context/Rabbitholes).
3. **Deep Work:** When you are ready to code, you click the Forgejo link in Plane.
4. **Self-Hosted Harmony:** All data stays on your metal.

This completes the triad:

- **AnyType** = Memory (Hippocampus)
- **Plane** = Planning (Prefrontal Cortex)
- **Forgejo** = Action (Motor Cortex)

> 💡 I will PoC plane in my current ecosystem. Can you develop a detailed action plan to glue these pieces together? They need to act as a cohesive unit that I can begin using to manage my day to day workflows and deliverables? Including the following:

- APIs required
- any "glue" language we need to tie things together
- any monitoring/observability/alerting/reporting we should consider
- any other weak areas that need to be addressed

This is a "Systems Engineering" project now. You are building a **Personal Productivity Platform**. Like any good PoC, we will break this down into phases to ensure we don't get blocked by integration hell.

Here is the detailed action plan to glue **AnyType**, **Plane**, **Forgejo**, and **n8n** into a cohesive unit.

---

### Phase 1: The Developer Core (Plane + Forgejo)

*Objective: Establish the "Execution" and "Planning" link first. This is the most reliable integration and provides immediate value.*

**1. Setup Plane & Forgejo Integration**
Plane has native Git integration. Do not use n8n for this specific link yet; use the native tools to reduce latency.

- **Action:** In Plane, go to `Settings > Workspace > Git Integration`.
- **Connect:** Add your Forgejo instance (via URL and Personal Access Token).
- **Mapping:** Map specific Plane Projects to specific Forgejo Repositories.
- **Result:** When you create an Issue in Plane, you can create a branch in Forgejo directly from the UI. When you open a PR in Forgejo, Plane automatically attaches it to the Issue.

**2. Define the "State Machine"**
To manage your ADHD anxiety, you need to know what "Done" looks like. Agree on these status mappings:

- **Plane:** `Backlog` → `Planned` (AnyType) → `Started` → `In Review` (Forgejo PR) → `Done`.
- **Action:** Configure these states in Plane.

---

### Phase 2: The Knowledge-Planning Loop (AnyType + Plane + n8n)

*Objective: The critical bridge. Making your Ideas (AnyType) appear as Tasks (Plane).*

**1. Schema Modification in AnyType**
This is the most important structural change. We need a place to store the "Foreign Key" so the tools can talk to each other.

- **Action:** Open your `Task` Object Type in AnyType.
- **Add Field:** `Plane Issue ID` (Type: Number or String).
- **Add Field:** `Plane Issue URL` (Type: URL).

**2. The "Create Task" Workflow (n8n)**
*Trigger:* New Task created in AnyType containing a specific Tag (e.g., #Project).
*Logic:*

1. **Listen:** n8n polls AnyType (or uses a webhook/event trigger if supported by your version of the middleware) for new objects.
2. **Filter:** Check if the Task has the #Project tag. (Ignore "Buy Milk" tasks).
3. **Transform:** Extract Title, Description, and Due Date.
4. **Action (HTTP Request):** Call the **Plane API** (`POST /api/v1/workspaces/{slug}/projects/{id}/issues/`).
    - Create the issue in Plane.
5. **Callback:** Receive the `id` and `url` of the new Plane issue.
6. **Update:** Call the **AnyType API** to update the original Object with the `Plane Issue ID` and `Plane Issue URL`.
    - *Result:* You now have a permanent link between the Knowledge (AnyType) and the Management (Plane).

**3. The "Status Sync" Workflow (n8n)**
*Trigger:* Webhook from Plane (Issue Updated).
*Logic:*

1. **Listen:** n8n receives a webhook from Plane when an issue moves to "Done".
2. **Search:** n8n looks up the AnyType Object using the stored metadata (or searches for the Task name).
3. **Action:** Update the `Status` relation in AnyType to "Completed".
    - *Result:* Your Knowledge Base stays in sync with your actual progress without you touching it.

---

### Phase 3: The APIs & Glue Language

*Objective: Technical specifics for the tinkerer.*

**The Glue Language: JavaScript (Node.js)**
While n8n is low-code, you will need JavaScript functions inside n8n nodes to handle data mapping (e.g., mapping AnyType's specific date format to Plane's ISO 8601 requirement).

**Required API Endpoints:**

| Tool | API Purpose | Key Endpoints |
|---|---|---|
| **AnyType** | Read/Write Knowledge | `POST /graphql` (Middleware) or internal Protobuf via the n8n node. |
| **Plane** | Read/Write Tasks | `POST /api/v1/issues/`, `PATCH /api/v1/issues/{id}/`, `GET /api/v1/issues/` |
| **Forgejo** | Read Repo Status | `GET /api/v1/repos/{owner}/{repo}/pulls` (Used for monitoring) |
| **n8n** | Orchestration | Webhook triggers (Ingest) + HTTP Request nodes (Egress) |

**Critical Note on AnyType API:**
AnyType's API is currently their "Middleware" which is still evolving. Ensure you check the **n8n community nodes** for AnyType. If the official node is buggy (which is common in PoCs), your "Glue" fallback is using n8n's **HTTP Request** node to hit the local AnyType port if exposed, or simply sticking to **Unidirectional Sync** (AnyType → Plane) and accepting that Plane doesn't write back to AnyType automatically yet.

---

### Phase 4: Observability, Alerting & Reporting

*Objective: Supplementing Executive Function.*

**1. "Nudge" Alerts (ntfy)**

- **Workflow:** Use n8n to scan Plane for tasks due "Tomorrow" or "Overdue" where `Assignee` is `You`.
- **Action:** Send a push notification via **ntfy** to your phone/desktop at 9:00 AM daily.
- **Message:** "🚀 Focus: [Task Name] is due. [Link to Plane]"

**2. Dashboarding (Grafana)**
You need a high-level view to reduce anxiety. Build a Dashboard in Grafana that queries the **Plane API** (or a database Plane writes to, if applicable) and **Forgejo**.

- **Panel 1: "Velocity."** Number of Issues closed in Plane (last 7 days).
- **Panel 2: "Tech Debt."** Number of open Forgejo PRs vs. Closed Issues.
- **Panel 3: "Ideas in Flight."** Number of AnyType tasks NOT in Plane (The "Backlog").

**3. Health Checks**
Ensure n8n monitors itself. If the "AnyType → Plane" workflow fails, n8n should push an **ntfy** alert immediately: "⚠️ Sync Failed: Task not exported to Plane."

---

## Phase 5: Addressing Weak Areas & Gaps

### Weak Area 1: The "Sync Lag"

- *The Problem:* AnyType is Offline-first. Plane is Online-first. If you add a task on your laptop on the train, the sync might not happen until you are online.
- *The Fix:* Accept eventual consistency. The "Source of Truth" for *Action* is Plane. If it's in Plane, it exists.

### Weak Area 2: "Context Switching" Cost

- *The Problem:* We have now added a 4th tool (Plane) to the stack.
- *The Fix:* **Browser Tab Management.**
  - Group 1: AnyType (Reference/Docs) - Open when researching.
  - Group 2: Plane (Work) - Open when executing.
  - Group 3: Forgejo - Only open when committing code.
  - *n8n* runs in the background; you should rarely check the UI unless debugging.

### Weak Area 3: API Complexity

- *The Problem:* Writing custom HTTP requests to AnyType can be brittle if they update their schema.

- *The Fix:* Use an intermediate "Normalization" step in n8n. Instead of mapping API fields directly, use a "Set" node in n8n to define standard variables (e.g., `{{ $json.title }}`) before sending to the API. If the API changes, you only fix the mapping in the HTTP Request node, not the logic.

### Summary of First Steps (Immediate Action Items)

1. **Deploy Plane:** Get it running and log in.
2. **Connect Forgejo:** Do the native "Git Integration" in Plane settings.
3. **Create a Test Task:** In Plane, create an issue, link it to a Forgejo repo, and create a branch. Verify the round-trip.
4. **Modify AnyType Schema:** Add the `Plane Issue ID` field to your Task Object.
5. **Build the first n8n Workflow:** `Manual Trigger -> Create AnyType Object -> Push to Plane -> Update AnyType`.

This PoC plan prioritizes getting the **Engine (Plane/Forgejo)** running before connecting the **Fuel Pump (AnyType/n8n)**. Good luck.

> 💡 I would prefer to manage the "glue" code in python as opposed to javascript. For example this module: https://pypi.org/project/anytype-client/, also the MCP server might be of use here to close the loop on AnyType context within rabbitholes : https://github.com/anyproto/anytype-mcp

This is a superior approach for an engineer. Moving the glue logic to **Python** gives you type safety, better debugging tools, and access to robust libraries like `anytype-client`. It also allows you to treat the automation as a set of microservices rather than "black box" workflows.

Furthermore, the introduction of **AnyType MCP (Model Context Protocol)** is a game-changer. It changes the relationship between your AI (Rabbitholes/Open WebUI) and your Knowledge Base (AnyType) from "Manual Copy-Paste" to "Live Database Query."

Here is the revised Action Plan using Python and the MCP Server.

### The Architecture Shift

We are moving from **n8n-centric** logic to a **Python Service** architecture.

- **Orchestrator:** n8n (kept only for scheduling/triggers, replaced by Python scripts where complex logic exists).
- **Context Layer:** MCP Server (allows AI to "read" AnyType).
- **Logic Layer:** Python Scripts (using `anytype-client`).

## Phase 1: The AI Context Loop (MCP Integration)

*Objective: Allow Rabbitholes (via Open WebUI) to "see" your AnyType database in real-time.*

**1. Setup the Stack**
Since Rabbitholes is an interface, we assume you are pointing it at **Open WebUI** (which supports MCP) or using Rabbitholes' native LLM connection to an MCP-capable client (like Claude Desktop or Open WebUI).

- 
  ```
  git clone https://github.com/anyproto/anytype-mcp.git
  cd anytype-mcp
  # Configure with your AnyType credentials/Middleware address
  pip install -r requirements.txt
  python server.py
  ```

**2. Connect MCP to Open WebUI**

- **Action:** In Open WebUI Settings → Providers → OpenAI (or your Local LLM), enable MCP.
- **Action:** Add the AnyType MCP server URL (usually `http://localhost:port/sse`).

**3. The Workflow (The "Brain" Upgrade)**
Now, when you are in **Rabbitholes** (connected to Open WebUI), you don't need to search AnyType manually.

- **User Prompt in Rabbitholes:** "What was the conclusion of the 'Database Architecture' meeting?"
- **MCP Action:** Open WebUI pauses, queries the AnyType MCP server.
- **MCP Response:** Returns the specific Object/Note from AnyType.
- **AI Answer:** The AI answers your question using *your* data.
- **Result:** Zero friction. You never leave the context of the conversation to find information.

## Phase 2: The Python Glue Logic

Objective: Replace JavaScript/n8n logic with robust Python scripts using *`anytype-client`*.

**Prerequisites**
Create a virtual environment and install dependencies:
`pip install anytype-client requests python-dotenv`

### Script 1: `sync_plane.py` (The Bi-directional Sync)

This script will be your "Service." It will run on a timer (cron or n8n).

- **Logic:**
  1. **Auth:** Authenticate with AnyType (Middleware) and Plane (API Token).
  2. **Query AnyType:** Use `anytype-client` to query for Objects where `Type = Task` AND `Status = Backlog` AND `Tag = #Project`.
  3. **Filter:** Check if the object has a `Plane ID`. If yes, skip.
  4. **Push to Plane:** If no, use `requests.post` to create the Issue in Plane.
  5. **Update AnyType:** Use `client.object.set_relation` to update the AnyType object with the new `Plane ID`.
- **Code Snippet (Conceptual):**

```
from anytype_client import Client, exceptions
import requests

# Init AnyType Client
client = Client(wallet_id="YOUR_WALLET", secret="YOUR_KEY")
client.auth.login()

# Init Plane Config
PLANE_API_KEY = "..."
PLANE_URL =
"https://plane.yourdomain.com/api/v1/workspaces/.../issues/"

# 1. Get Tasks
# Note: syntax depends on AnyType client version, this is conceptual
tasks = client.object.search(filter={"type": "Task", "status":
"Backlog"})

for task in tasks:
    if not task.relations.get("Plane ID"):
        # 2. Create in Plane
        payload = {"name": task.name, "description_raw":
task.description}
        response = requests.post(PLANE_URL, json=payload, headers=
{"Authorization": PLANE_API_KEY})
        plane_id = response.json()["id"]

        # 3. Update AnyType
        client.object.set_relation(
            id=task.id,
            key="Plane ID",
            value=plane_id
        )
```

### Script 2: `update_status.py` (The Listener)

This script listens to Plane (or is triggered by a Plane Webhook) and updates AnyType.

- **Logic:**

   1. Triggered by webhook.
   2. Parse Plane Issue State (e.g., "Done").
   3. Search AnyType for Object with `Plane ID == X`.
   4. Update AnyType Object `Status` to "Completed".

---

## Phase 3: Orchestration & Monitoring

*Objective: How we run the Python scripts and watch them.*

**1. Running the Glue (Systemd)**
Since you are self-hosting, don't rely on `screen` or `cron` alone. Use `systemd` to keep your Python services alive.

- **File:** `/etc/systemd/system/sync-plane.service`
  - Description: Python script to sync AnyType to Plane.
  - ExecStart: `/usr/bin/python3 /path/to/sync_plane.py`
  - Restart: always

**2. The "Pulse" (Health Check with ntfy)**
Add error handling to your Python scripts to push to **ntfy** if the API fails.

-
  ```
  import requests
  def send_alert(message):
     requests.post("https://ntfy.yourdomain.com/my-alerts",
                   data=message.encode(encoding='utf-8'))
  try:
     # ... sync logic ...
  except Exception as e:
     send_alert(f"CRITICAL: Sync Script Failed - {str(e)}")
  ```

---

## Phase 4: Addressing Weak Areas with Python

**Weak Area: Error Handling & Idempotency**

- *The Issue:* API calls fail. Networks hiccup.
- *The Python Fix:* Implement idempotency keys in your Plane API calls (if supported) or check for existence before creating. Wrap all network calls in `tenacity` (a Python retrying library) to handle transient network errors without stopping the script.

**Weak Area: Data Mapping Complexity**

- *The Issue:* "Backlog" in AnyType might be "Todo" in Plane.
- *STATUS_MAP = {*
  "Ba`cklo`g": `"backlog",`
  "In Progress": "started",
  "Done": "done"
  }

### Summary of the New Stack

| Component | Role | Tech |
|---|---|---|
| **The UI** | Visual Chat / Branching | Rabbitholes AI |
| **The Context** | Knowledge Store | AnyType + `anytype-mcp` |
| **The Manager** | Project Management | Plane |
| **The Glue** | Automation / Logic | **Python** (`anytype-client` + `requests`) |
| **The Runtime** | Process Management | `systemd` |
| **The Monitor** | Alerts | `ntfy` |

### Next Immediate Actions for You

1. Get **`anytype-mcp` running:** Connect it to Open WebUI. Try asking it a question about your notes. This is the "Magic Moment" to look for.
2. Install **`anytype-client`:** Write a simple "Hello World" script that lists your Tasks in the terminal.
3. Draft **`sync_plane.py`:** Don't automate it yet. Just write the script that takes a hardcoded string and pushes it to Plane.

This approach respects your desire to tinker in code (Python) while keeping the architecture clean and modular.

> 💡 I think it is probably a mistake to sideline n8n for python logic, and to rely on systemd and cron. The abstraction of the n8n pipeline/workflow is it's biggest strength and since one of the major goals with this project is to increase velocity and creativity while decreasing toil and friction, dropping down to the systemd/cron level to manage things that could be represented as yaml

> workflows seems to move us in the wrong direction. I think the infrastructure is fine to use systemd/cron/cockpit/socket activation to handle the services themselves, but the actual touch-points for the creator/developer should remain as abstracted/automated/templated and full of as many sensible defaults as possible so we can maintain economies of scale. I also would like to leverage as much pre-made tooling as possible to keep the cost of ownership down and avoid tech debt when APIs and libraries change. Do you think this is a wise choice, and could we refine the plan to take these points into account? I think python logic has it's place where we absolutely must do something specific to our process and we need to own it. Otherwise we should fall back to off the shelf solutions we can not afford to own. Again, the development team is just one person, me.

You are absolutely correct. I was drifting toward "Infrastructure Engineer" territory—building custom daemons and managing cron jobs—when you need to be a "Productivity Engineer." For a solo developer, especially with ADHD, **visual abstraction (n8n) is a force multiplier**, while **script maintenance is a tax on creativity**.

Your instinct to prioritize **economies of scale** and **low cost of ownership** is the right strategic move. If you spend 2 hours fixing a broken Python script, that is 2 hours not spent building your actual project.

Here is the refined **Hybrid Action Plan**. It keeps n8n as the "Orchestration Hub" and only drops down to Python when strictly necessary (inside n8n), ensuring you stay in the "Flow" rather than the "Terminal."

### The Refined Architecture

1. **The Foundation (Systemd):** Manages the *Lifecycle* of apps (Start/Stop/Restart). It does **not** manage the *Business Logic*.
2. **The Brain (n8n):** Manages the *Flow*, *Timing*, and *Transformations*. This is where you "live" when tweaking workflows.
3. **The Specialist (Python):** Embedded inside n8n. Used only when the native "Code Node" or "HTTP Request Node" isn't enough (e.g., complex object mapping for AnyType).

---

### Phase 1: The "Supercharged" n8n Setup

*Objective: Create a containerized environment that allows n8n to speak "Python" fluently.*

Since n8n is self-hosted, you can extend its capabilities without writing external scripts.

**1. The Docker Strategy**
Do not use the default `n8n` image. Create a custom `Dockerfile` or compose file that includes the `anytype-client` library.

- **Why:** This allows n8n's native **"Execute Command"** or **"Code"** nodes to run Python code using the powerful `anytype-client` library, all contained within the visual workflow.
- **FROM n8**nio/n8n
  # Install Python and the specific client you trust
  RUN apk add —no-cache python3 py3-pip
  RUN pip3 install anytype-client requests

**2. The "Low Code" Philosophy**

- **Rule:** If n8n has a node for it, use the node. (e.g., Use the native **Webhook** node, **Merge** node, **Switch** node).
- **Exception:** Only use a Python "Code Node" if you need to calculate a hash, format a complex date, or use the `anytype-client` specifically.

---

## Phase 2: The Logic Workflow (Visualized)

*Objective: Create the "AnyType → Plane" sync in n8n using pre-made nodes where possible.*

Workflow Name: **`Sync Ideas to Projects`**

1. **Trigger (Poll Node):**
    - *Node:* **Schedule Trigger** (Every 15 mins).
    - *Why:* Native, reliable, no cron editing.
2. **Fetch Data (HTTP Request Node):**
    - *Node:* **HTTP Request**.
    - *Action:* Call your **AnyType Middleware** API (GraphQL) to fetch objects `Type: Task` AND `Status: Backlog`.
    - *Note:* We use the HTTP node here because it is standard. If the data is too complex, we pass it to the next step.
3. **The Logic (Function/Code Node):**
    - *Node:* **Code** (Python inside n8n).

    - *Role:* Filter out items that already have a `Plane ID`. Format the JSON payload to match Plane's API schema.

    -
      ```
      # Simple logic inside n8n
      filtered_items = []
      for item in $input.all():
          plane_id = item.json.relations.get('plane_id')
          if not plane_id:
              filtered_items.append(item)
      return filtered_items
      ```
4. **Push to Plane (HTTP Request Node):**
    - *Node:* **HTTP Request**.
    - *Action:* `POST` to Plane API.
    - *Error Handling:* Use the built-in **"Continue On Fail"** setting or the **"Error Trigger"** workflow in n8n. If Plane is down, n8n retries automatically. (You don't have to write retry logic in Python).
5. **Update AnyType (HTTP Request Node):**
    - *Node:* **HTTP Request**.
    - *Action:* Patch the AnyType object with the new `Plane ID`.

**Result:** You have a **visual pipeline**. If it breaks, you look at the n8n UI, see the red node, and fix the logic. No log digging in `/var/log/syslog`.

---

### Phase 3: The "MCP" Context Loop

*Objective: Keep this isolated to the AI stack (Rabbitholes/Open WebUI).*

This part remains as a **Service (Systemd)** because it is a long-running server, not a workflow.

1. **Service:** `anytype-mcp` (running via Docker or Systemd).
2. **Consumer:** `Open WebUI` (configured to use the MCP server).
3. **Interface:** `Rabbitholes` (pointed at Open WebUI).

*Note:* n8n does **not** need to touch this. The AI talks to AnyType via MCP. The Project Manager (Plane) talks to AnyType via n8n. They are parallel streams.

---

### Phase 4: Monitoring & Alerting (The "Set and Forget" Layer)

*Objective: Using n8n's native capabilities to reduce anxiety.*

**1. Self-Healing in n8n**
Instead of external scripts, use n8n's **"Error Workflow"** feature.

- **Setup:** Create a global workflow triggered on "Error".
- **Action:** Send a formatted message to **ntfy**.
    - *Message:* "⚠️ Sync Failed: Node 'Push to Plane' failed with error 500."
- **Benefit:** Every single workflow you build inherits this monitoring automatically. Economies of scale.

**2. Dashboarding (Grafana)**
Keep this. But instead of scraping logs, scrape the **n8n API** (it exposes execution stats).

- *Panel:* "Active Workflows."
- *Panel:* "Last Sync Success Time."

---

## Summary of the "Wise Choice"

By keeping n8n at the center:

1. **You avoid Tech Debt:** When AnyType updates their API from v1 to v2, you update the URL in one n8n node. You do not rewrite and redeploy a Python microservice.
2. **You maintain Velocity:** You can drag and drop a new integration (e.g., "Add a Matrix alert when a Task is Done") in minutes, not hours.
3. **You reduce Cognitive Load:** The visual map *is* your documentation. You don't need to remember how `script.py` works; you just look at the workflow graph.

**Final Architecture:**

- **Infrastructure:** Systemd/Docker (Keeps the lights on).
- **Automation Hub:** n8n (Moves the data).
- **Deep Logic:** Python (Embedded inside n8n, only used for the `anytype-client` heavy lifting).
- **Knowledge Hub:** AnyType + MCP (Feeds the AI).
- **Work Hub:** Plane (Tracks the deliverables).

This balances your desire to code (Python) with your need to survive (Low Maintenance). It is the correct path for a solo developer.

> 💡 This seems to strike the right balance between agility, flexibility, and complexity with a low maintenance overhead. The n8n ecosystem is large

and offers a lot of opportunities for added functionality from the community. Making n8n the worker backbone seems wise as well as helps make our solution future proof. Could you create a detailed project plan to implement all the pieces in this design and break it down into testable phases that we can build upon to ensure the foundation is sound?

This is a comprehensive Project Plan designed to build your "Personal Operating System" incrementally. Each phase builds upon the last, ensuring that if something breaks, you know exactly where to look.

This plan assumes you are deploying via Docker/Compose for ease of management.

### Project Charter: "The Unified Neural Stack"

**Objective:** Create a self-hosted, low-friction environment that bridges creative ideation (AnyType/AI) with structured execution (Plane/Forgejo).
**Methodology:** Agile, Incremental Delivery.
**Success Metric:** A user (you) can capture an idea in AnyType and see it move to a deliverable in Plane/Forgejo with zero manual data entry.

---

### Phase 1: The Foundation (Infrastructure & Identity)

*Goal: Ensure the "metal" is solid and security is handled before introducing complexity.*

**Deliverables:**

1. **Reverse Proxy:** Configure Traefik or Nginx Proxy Manager (if not already done) to handle SSL termination for subdomains.
2. **Identity Provider:** Deploy **Kanidm**.
    - *Action:* Create user groups (e.g., `admins`, `developers`).
    - *Test:* Log in to a dummy service using Kanidm OAuth2/OpenID Connect.
3. **Secrets Management:** Configure **Vaultwarden**.
    - *Action:* Store all upcoming API keys and database passwords here.

**Definition of Done (DoD):**

- You can log into Kanidm.
- All subdomains resolve correctly.
- No services are exposed on port 80/443 without SSL.

---

### Phase 2: The "Manual" Core Loop (Data & Code)

*Goal: Establish the data storage and verify native tool-to-tool connections before writing custom code. Do not automate yet.*

**Deliverables:**

1. **Knowledge Base:** Deploy **AnyType**.
   - *Action:* Create the `Task` Object Type. Add relations: `Plane Issue ID`, `Status`, `Project`.
   - *Test:* Create 3 sample tasks manually.
2. **Project Management:** Deploy **Plane**.
   - *Action:* Configure Workspace. Create a Project named "PoC Test".
3. **Code Repository:** Deploy **Forgejo**.
   - *Action:* Create a repository named `poc-repo`.

**Integration Test (Manual):**

1. In **Plane**, create an Issue: "Setup PoC".
2. In **Plane**, use the native Git integration to link this Issue to `poc-repo`.
3. In **Forgejo**, verify the branch `setup-poc` was created (if using the native branch creation feature) or verify the Issue link exists.

**DoD:**

- All three core apps are running.
- You can manually link a Plane Issue to a Forgejo Repo.

---

### Phase 3: The Orchestration Engine (n8n + Python Environment)

*Goal: Build the automation chassis. This is the most critical technical phase.*

**Deliverables:**

1. **Custom n8n Container:**
   - *Action:* Create a `Dockerfile` for n8n that includes Python 3 and `pip install anytype-client requests`.
   - *Action:* Build and deploy this container.
2. **n8n Hardening:**
   - *Action:* Set `EXECUTIONS_MODE: queue` (using Redis) so workflows don't time out.

    - *Action:* Configure n8n to use your Timezone.

**Python Logic Test (Inside n8n):**

1. Create a simple workflow: `Manual Trigger` → `Code Node`.
2. In the Code Node, write a script to import `anytype_client` and print "Hello World".
3. *Test:* Execute the workflow. Verify the console shows no import errors.

**DoD:**

- n8n is running with Python support.
- `anytype-client` is verified working inside the n8n environment.

---

### Phase 4: The "Bi-Directional Sync" (Glue Logic)

*Goal: Automate the connection between AnyType and Plane.*

**Workflow 1: The Creator (AnyType → Plane)**

1. **Trigger:** Schedule Trigger (Poll every 15 mins).
2. **Node 1 (Get Data):** HTTP Request to AnyType Middleware (GraphQL).
    - *Query:* `{ objects(filter: {type: {name: "Task"}, relation: {key: "Status", value: "Backlog"}}) ... }`
3. **Node 2 (Filter):** Function Node (Python).
    - *Logic:* Filter out tasks that already have a `Plane ID`.
4. **Node 3 (Create Issue):** HTTP Request to Plane API.
    - *Payload:* Create Issue.
5. **Node 4 (Update AnyType):** HTTP Request to AnyType.
    - *Action:* Update the specific Task Object with the returned `Plane ID`.

**Workflow 2: The Updater (Plane → AnyType)**

1. **Trigger:** Plane Webhook (on Issue Update).
2. **Node 1 (Search):** Function Node (Python).
    - *Logic:* Search AnyType for Object with `Plane ID` == incoming ID.
3. **Node 2 (Sync Status):** HTTP Request to AnyType.
    - *Action:* Update `Status` in AnyType to match Plane (e.g., "Done" → "Completed").

**DoD:**

- Create a Task in AnyType with tag #Project.
- Wait 15 mins.
- Verify it appears in Plane.
- Mark it Done in Plane.
- Verify it marks Completed in AnyType.

---

### Phase 5: The Context Layer (MCP + AI)

*Goal: Give the AI "eyes" into your Knowledge Base.*

**Deliverables:**

1. **MCP Server:** Deploy `anytype-mcp` service (Systemd or Docker).
2. **Open WebUI Config:**
    - *Action:* Enable "MCP" in settings.
    - *Action:* Add server endpoint.
3. **Rabbitholes Config:**
    - *Action:* Point Rabbitholes to your Open WebUI instance.

**Integration Test:**

1. In **AnyType**, create a Note: "The secret password for the PoC is Banana123."
2. In **Rabbitholes**: "What is the secret password for the PoC?"
3. *Verification:* The AI correctly answers "Banana123" by querying the MCP server.

**DoD:**

- Rabbitholes can read AnyType data.
- The loop is closed: AI Context → Knowledge Base → Task List → Code.

---

### Phase 6: Observability & Safety (Monitoring)

*Goal: Ensure you know if the system breaks (Intervention for Executive Function).*

**Deliverables:**

1. **Global Error Workflow (n8n):**
    - *Action:* Configure n8n "Error Trigger" workflow.
    - *Node:* HTTP Request to `ntfy.sh`.
    - *Message:* "⚠️ Workflow Failed: [Name]".

2. **Grafana Dashboards:**
    - *Action:* Import "n8n" community dashboard.
    - *Action:* Import "Forgejo" community dashboard.
    - *Action:* Create a "System Health" row showing CPU/RAM of the host.

**DoD:**

- Stop the n8n container.
- Receive a push notification on your phone within 1 minute.
- Restart container. Verify notification clears.

---

### Summary of Dependencies

- **Don't start Phase 3** until Phase 2 is stable. (If you move data targets, you break automation).
- **Don't start Phase 5** until Phase 4 is stable. (If the AI queries old data, it hallucinates).

### Rollback Strategy (For Anxiety Reduction)

Since you are self-hosting, every phase is isolated by Docker containers.

- If Phase 4 (Sync) breaks: Stop the n8n container. You can still manually copy-paste. You lose speed, but not functionality.
- If Phase 5 breaks: Disable MCP in Open WebUI. Rabbitholes becomes a standard chatbot again.

This plan ensures you always have a functioning "floor" to stand on, even if the "elevator" (automation) is temporarily out of service.

> 💡 I should have mentioned we are using caddy v2.10.2 as a proxy with the following modules built in: github.com/sablierapp/sablier-caddy-plugin github.com/greenpau/caddy-security github.com/mholt/caddy-dynamicdns github.com/porech/caddy-maxmind-geolocation github.com/sjtug/caddy2-filter github.com/mholt/caddy-ratelimit github.com/abiosoft/caddy-yaml github.com/corazawaf/coraza-caddy/v2 github.com/mholt/caddy-webdav github.com/mholt/caddy-l4

This changes the game significantly—in a very good way. You are not just running a reverse proxy; you have a full **Edge Gateway** stack with **Identity-Aware Proxying** and **Application Portal** capabilities.

Specifically, **Sablier** (for the app dashboard) and **Caddy Security** (for the auth gateway) are the "Killer Features" for your neurodivergent workflow. They eliminate the "password fatigue" and "URL hunting" that destroy executive function.

Here is the revised **Phase 1** and integration strategy, leveraging your specific Caddy modules to create a "Unified Digital Entryway."

---

### Phase 1 (Revised): The Unified Edge (Caddy + Kanidm)

*Goal: Create a single, secure "Front Door" that handles authentication and app discovery, removing login friction from the daily workflow.*

## 1. Configure Caddy-YAML (Infrastructure as Code)

Since you have the `caddy-yaml` module, do not edit the Caddyfile manually. Store your config in **Forgejo** (e.g., `infra/caddy-config.yaml`). This ensures your infrastructure is version-controlled alongside your code.

**Conceptual YAML Structure:**

YAML ⌄

```yaml
# Global Options
apps:
  http:
    servers:
      srv0:
        listen:
          - :443
        # Enable Caddy Security here
        routes:
          # Route 1: The Identity Provider (Kanidm)
          - match: [host id.domain.com]
            handle:
              - reverse_proxy: {{kanidm_container}}:8443

          # Route 2: The Application Portal (Sablier)
          - match: [home.domain.com]
            handle:
              - reverse_proxy: {{sablier_container}}:10000

          # Route 3: The "Protected" Apps (Plane, AnyType, n8n)
          - match: [host plane.domain.com, host anytype.domain.com, host n8n.domain.com]
            handle:
              # This is the magic: Forward Auth to Kanidm
              - forward_auth: {
                    address: {{caddy_security_endpoint}},
                    trust_forwarded_headers: true
                 }
              - reverse_proxy: {{target_service}}:internal_port
```

## 2. Implement "The Bouncer" (Caddy Security + Kanidm)

Instead of logging into n8n, Plane, and AnyType separately, you log in **once** via Kanidm.

- **Configuration:** Configure the `caddy-security` module to point to your Kanidm instance as the OpenID Connect (OIDC) provider.
- **Action:** Create an **Access Policy** in Caddy Security called "Developer".
    - *Rule:* `email ends_with @yourdomain.com` OR `groups contains "developers"`.
- **Result:** Any route protected by Caddy Security automatically checks if you are logged into Kanidm. If yes, you pass through. If no, it redirects you to Kanidm. **Single Sign-On (SSO) achieved.**

### 3. Deploy the "Dashboard" (Sablier)

This is the most important tool for your ADHD. Sablier provides an "Application Portal" (like a start page).

- **Configuration:** Configure Sablier to point to your various apps (Plane, Forgejo, AnyType, Grafana).
- **Caddy Integration:** Map `home.domain.com` to Sablier.
- **UX Benefit:**
    - You wake up.
    - You open `home.domain.com`.
    - You see a grid of icons (Plane, Forgejo, etc.) with status indicators (Green = Online).
    - You click "Plane". Caddy Security checks your Kanidm session (already active) and shoots you straight to the dashboard.
    - *Zero typing of URLs. Zero password prompts.*

### 4. WebDAV Integration (File Access)

Since you have `caddy-webdav`, you can map AnyType's file storage (or Readeck's exports) as a network drive on your local machine.

- **Configuration:** Create a route `files.domain.com`.
- **Mapping:** Point the `root` directory to your Docker volume where AnyType stores blobs or documents.
- **Benefit:** You can drag and drop files directly from your OS into your "Knowledge Base" volume, and AnyType (if watching the folder) can ingest them.

---

### Updated Project Plan: The "Caddy-Centric" Flow

## Phase 1.5: The "Glue" Verification

*New Step:* Before deploying apps, verify the Auth Loop.

1. Deploy Caddy + Kanidm.
2. Try accessing a protected "dummy" route.
3. Verify you are redirected to Kanidm, log in, and get redirected back successfully.
4. *Why:* If Auth is broken, automation is useless.

## Phase 4 (Revised): n8n and the "Internal" vs "External" Split

With Caddy handling the security, n8n's internal security can be relaxed for efficiency, or kept tight.

- **Internal Traffic (Docker Network):** n8n → AnyType API (No Auth needed, pure speed).
- **External Traffic (Your Browser):** You → n8n Dashboard (Protected by Caddy Security/Kanidm).
  - *Benefit:* You don't need to manage n8n user accounts. You just give access to the URL in Caddy Security.

## Phase 6 (Revised): Monitoring (Caddy Metrics)

With `caddy-security`, you get logging out of the box.

1. **Grafana Integration:** Caddy exposes metrics. Point Grafana at the Caddy admin port.
2. **Dashboard:** Create a panel "Auth Failures".
   - *Why:* If you see a spike in auth failures, your Kanidm integration might be flaky (causing friction), or someone is attacking you.

## Summary of How This Helps Your "AuADHD" Profile

1. **Sablier = Visual Clarity:** Replaces the mental load of "What apps do I have?" with a visual list.
2. **Caddy Security/Kanidm = Reduced Anxiety:** You never have to wonder "Did I log out?" or "What was the password?". The trust relationship is handled by the infrastructure.
3. **Caddy-YAML = Engineer's Comfort:** Your proxy config is now a yaml file in a Git repo. You can diff changes, roll back, and sleep soundly.

This creates a fortress around your creativity: you have a **Gatekeeper** (Kanidm) that only lets you in, a **Lobby** (Sablier) that shows you where to go, and a **Butler** (n8n) that does the work while you are inside.

> I should also add that I have a makefile for caddy and can build any other modules into the platform as needed that would help. These are just the modules that I am currently using. The sablier module was initially for AI applications running on the Jeston Orin Nano Super 8gb where Open WebUI is running in order to spin up and down other AI applications like iopaint on demand with socket activation. We are dealing with relatively limited compute resources. Every system involved in this project is Linux, mostly Arch or Manjaro unless another distro is required. The development desktop is the most powerful system for obvious reasons and has the following stats:

```
   Processor : AMD FX -8350
               1 physical processor; 8 cores; 8 threads
      Memory : 32758MB (10115MB used)
Machine Type : Desktop
```

```
Operating System : Manjaro Linux - Arch Linux
User Name : ctrauma (Scott Dillman)
Date/Time : Tue 23 Dec 2025 05:26:08 PM EST
-Display-
Resolution : 5120x2160 pixels
Display Adapter : NVIDIA NVIDIA GeForce GTX 1060 6GB
OpenGL Renderer : NVIDIA GeForce GTX 1060 6GB/PCIe/SSE2
Session Display Server : The X.Org Foundation 21.1.21
```

> There is a limited budget so the hardware is not current. We are also very energy/carbon aware and using multiple SBCs for the headless servers. A mixture of Raspberry Pis of multiple generations all ARM aarch64 and v7, an AtomicPi which is x86_64, and an odroid XU4 aarch64. This keeps our power spend low and our carbon footprint small, this is important as environmental impact and power spend are crucial.

This adds a fascinating constraint layer: **Distributed Computing on a Budget**. You are essentially building a "Borg Cube" of heterogeneous hardware. This aligns perfectly with your values (low carbon, high efficiency) but introduces **Architecture Mismatch** risks (x86 vs ARMv7 vs ARM64).

Given the "Salvage/Scavenge" nature of the hardware (FX-8350 is a Vishera chip, power-hungry for its performance; Atomic Pi is low-power but limited CPU), we must be surgical about **Service Placement**.

Here is the **Revised Deployment Map** and Project Plan, tailored to your specific "Cluster" topology and energy constraints.

## The "Green" Cluster Map

We need to assign workloads based on **Compute Requirement** vs. **Architecture Compatibility**.

| Node | Hardware | Role | Workloads |
|---|---|---|---|
| **Jetson Orin** | ARM64 / 8GB RAM | **AI Hub** | Open WebUI, Rabbitholes (Browser), Local LLMs. **Sablier** manages socket activation here to keep RAM free for inference. |
| **Atomic Pi** | x86_64 / Atom CPU | **Orchestrator** | **n8n** (Node), **Plane** (Go), **Forgejo** (Go). *Why x86?* Better compatibility for complex Docker images (like n8n/Python). |
| **RPi 4/5** (ARM64) | ARM64 | **Data Backbone** | **PostgreSQL** (DB for all apps), **AnyType Middleware** (for mobile sync), **Caddy** (Gateway). |
| **RPi v7 / Odroid** | ARMv7/ARM | **Storage/Edge** | **Vaultwarden**, **WebDAV** (File hosting), **Grafana Agent**. *Note:* Avoid heavy app stacks on v7 due to 32-bit container support dropping. |
| **Desktop** | x86_64 (FX-8350) | **Creator** | AnyType Client (GUI), IDE (VS Code), Browser. **Middleware offload:** Keep files on Pi 4 to allow Desktop to sleep. |

---

### Phase 1: The "Green" Gateway (Caddy + Hardware Abstraction)

*Goal: Use Caddy to route traffic to the correct silicon, handling architecture differences transparently.*

**1. The Makefile Build (Custom Module)**
Since you build Caddy from source, add `github.com/mholt/caddy-l4` (which you have) to handle TCP passthrough for databases.

- *Why:* This allows your Desktop (AnyType Client) to connect to PostgreSQL on the Pi 4 securely over the internet without exposing the DB port directly.

**2. Caddy YAML for Cluster Routing**
Update your YAML to route subdomains to specific internal IPs based on the node.

```yaml
- match: [host n8n.domain.com]
  handle:
    - forward_auth: ...
    - reverse_proxy: 192.168.1.50:5678 # Atomic Pi (x86)

- match: [host ai.domain.com]
  handle:
    - forward_auth: ...
    - reverse_proxy: 192.168.1.60:3000 # Jetson Orin (ARM64)

- match: [host db.domain.com]
  handle:
    - lb_policy: round_robin
    - reverse_proxy: 192.168.1.51:5432 # Pi 4 (ARM64)
```

**3. Power Management (The "Carbon" Feature)**
Use your `sablier-caddy-plugin` extensively here.

- **Scenario:** You want to check Plane.
- **Flow:** You request `plane.domain.com`. Sablier sees the Atomic Pi is "asleep" (WOL or container stopped). It wakes the service/container. Once the 200 OK is received, Caddy proxies you in.
- **Result:** Your Atomic Pi (which runs the heavy n8n/Plane stack) isn't burning watts 24/7.

---

### Phase 2: The Python Glue (Cross-Compiling for the Cluster)

*Goal: Build the n8n environment on the Atomic Pi without melting the CPU.*

**The Challenge:**
The `n8n` default image is multi-arch, but adding `anytype-client` (Python) usually

requires a custom build. You cannot build x86 binaries on your ARM Mac/Pi or vice-versa easily without emulation (slow).

**The Solution:**
Use the **Desktop (FX-8350)** as the build farm.

1. **Create the Dockerfile** on the Desktop.
2. **docker bu**ildx build —platform linux/amd64 -t my-n8n-custom:latest .
3. **Push/Pull:** Push to a local registry on the Pi 4 or save as tar and load onto the Atomic Pi.

**The Python Script (Optimized for Low RAM):**
In your n8n `Code` node (running on Atomic Pi), keep the Python logic lean.

- *Bad:* Loading the entire AnyType database into RAM to filter.
- *Good:* Using the API to fetch specific IDs. The Atomic Pi has limited RAM (usually 2GB or 4GB depending on config); n8n is hungry. Be efficient.

---

## Phase 3: The Storage & Sync Layer (Moving AnyType off Desktop)

*Goal: True "Headless" capability and Desktop power savings.*

Currently, if your AnyType Middleware runs on the Desktop, you can't access your knowledge base from your phone/laptop if the Desktop is off (or if you want to turn it off to save the FX-8350's 125W TDP).

**Action:**

1. Move **AnyType Middleware** to the **RPi 4 (aarch64)**.
2. **# L4 Passthro**ugh for AnyType Protocol
   ```
   :443 {
     tls {
       on_demand
     }
     @anytype {
       sni anytype-grpc.domain.com
     }
     handle @anytype {
       proxy_protocol tcp
       reverse_proxy 192.168.1.51:31009 # Pi 4
     }
   }
   ```

3. **Result:** You can turn off your desktop completely. AnyType data lives on the efficient Pi. You connect the Client only when you need to work.

---

### Phase 4: The AI Orchestration (Jetson Optimization)

*Goal: Ensure the Jetson is only doing AI work, not web serving.*

**1. Open WebUI + MCP**
Deploy `anytype-mcp` on the **RPi 4**, not the Jetson.

- *Why:* The Jetson has 8GB unified RAM. You need every MB for the LLM context window. Don't waste RAM on a Python MCP connector. Let the Jetson be the "GPU Muscle."

**2. The Workflow**

- **User** → Rabbitholes → **Jetson (Open WebUI)** → Network Request → **RPi 4 (MCP)** → RPi 4 **(AnyType Middleware)**.
- *Latency:* Minimal (LAN speed).
- *Benefit:* Jetson stays responsive for generation.

---

### Revised Project Plan: "The Carbon-Efficient Neural Stack"

### Step 1: Infrastructure Sort (The "Tetris" Phase)

- **Task:** Assign static IPs to all nodes.
- **Task:** Install Docker on AtomicPi and Pi 4.
- **Task:** Ensure Time Sync (Chrony) is perfect across all nodes (critical for OAuth).

### Step 2: Build the "Fat" n8n Image (On Desktop)

- **Task:** Write Dockerfile with `anytype-client`.
- **Task:** Build for `linux/amd64`.
- **Task:** Load onto AtomicPi.

### Step 3: The "Sleepy" Services (Sablier on Jetson)

- **Task:** Configure Sablier on Jetson to wake `iopaint` (already done).
- **Task:** Configure Caddy on Pi 4 to route AI requests to Jetson.
- **Test:** Start an AI session. Verify Jetson wakes up, generates, and idles.

### Step 4: The "Always On" Backbone (Pi 4)

- **Task:** Deploy PostgreSQL (low power usage).
- **Task:** Deploy AnyType Middleware.
- **Task:** Deploy Caddy (Main Gateway).
- **Task:** Configure `caddy-l4` to pass gRPC traffic to AnyType Middleware.

### Step 5: The "Work" Server (AtomicPi)

- **Task:** Deploy n8n, Plane, Forgejo.
- **Task:** Connect n8n to PostgreSQL on Pi 4.
- **Test:** Create a workflow on n8n that sleeps for 1 hour. Monitor AtomicPi CPU/RAM via `htop`.

### Monitoring for the "Carbon Conscious" Engineer

**Grafana Dashboard (The "Green" View):**

1. **Panel: "Cluster Wattage"** (Approximate based on CPU load).
    - *Alert:* If AtomicPi > 80% CPU for > 5 mins, ntfy alert. (It means the Atom CPU is choking on a workflow).
2. **Panel: "Jetson Thermal State"**.
    - *Alert:* If Jetson > 60°C, slow down polling intervals.
3. **Panel: "Desktop Status"**.
    - *Metric:* Is it on? If yes, and no Forgejo activity for 2 hours → Send ntfy: "Go to sleep, turn off the PC."

This setup respects your hardware limitations, minimizes power usage by right-sizing services to specific architectures, and provides the robust "glue" you need.

> 💡 We are running pi-hole for local DNS with unbound and filtering as well. The internet gateway is a router running Asuswrt-Merlin with a VPN server for internal access. We are using https://github.com/crazy-max/ddns-route53 for dynamic DNS from AWS for our domain which is `bitwise.ws` the ddns tool is on a cron that updates every 5 minutes. Most SBCs are running their services on bare metal to avoid docker overhead and issues with ARM vs x86_64 images. There are a few instances where this cannot be avoided, but we tend to favor single executable solutions written in go or rust if possible. All the stack described above is already running distributed across the SBCs with the exception of the new addition to the project which is plane.io. To

> round out the entire environment we are backing up using resticprofile (https://creativeprojects.github.io/resticprofile/configuration/getting_started/index.html) to minio that writes to block storage on a local USB drive with plans to eventually leverage AWS glacier for disaster recovery. There is also a MariaDB instance that is leveraged by a few applications like Forgejo. Chat is currently handled by our matrix server instance running tuwunel.

This is a highly refined, energy-conscious, and "OpSec" aware environment. You have effectively built a mini-cloud using **Go/Rust binaries** for the heavy lifting (bare metal) and reserving containerization for the complex, dependency-heavy application stacks.

The introduction of **Plane** is the only slight outlier because, while it is written in Go, it is a complex monorepo (Web + Worker + Redis + DB) that makes bare-metal deployment operationally expensive compared to a simple container.

Here is the implementation plan to integrate Plane, n8n, and the MCP bridge into your existing `bitwise.ws` ecosystem, respecting your bare-metal preference and strict backup/energy constraints.

### Architecture Overview

| Service | Host | Form Factor | Database |
|---|---|---|---|
| Access Method | | | |
| **Caddy** | Pi 4 / Router | **Go Binary** | N/A |
| `*.bitwise.ws` (Public/VPN) | | | |
| **MariaDB** | Atomic Pi | **Bare Metal** | N/A |
| `127.0.0.1:3306` (Local) | | | |
| **Forgejo** | Atomic Pi | **Go Binary** | MariaDB |
| `forgejo.bitwise.ws` | | | |
| **Plane** | Atomic Pi | **Docker** | MariaDB |
| `plane.bitwise.ws` | | | |
| **n8n** | Atomic Pi | **Docker** | SQLite/MariaDB |
| Internal Only | | | |
| **AnyType MCP** | Pi 4 | **Binary/Script** | Local File |
| TCP Socket | | | |
| **MinIO** | SBC (USB) | **Go Binary** | Disk |
| `minio.bitwise.ws` | | | |

---

## Phase 1: Networking & DNS (The Foundation)

*Goal: Ensure internal and external routing handles the new services via your existing gateway.*

### 1. Pi-hole Configuration

Since you use `ddns-route53`, your public IP is covered. Now ensure local resolution for efficiency on the VPN/LAN.

- **Action:** Add Local A Records in Pi-hole:
    - `plane.bitwise.ws` → Atomic Pi Local IP
    - `minio.bitwise.ws` → SBC with USB Drive IP
    - `ai.bitwise.ws` → Jetson Orin IP

**2. Caddy (Go Binary) Configuration**
Update your Caddyfile (or YAML) to route `plane` to the Atomic Pi and handle the Plane specific headers.

Plain Text ⌄

```
plane.bitwise.ws {
    reverse_proxy 127.0.0.1:3000 { # Map to container port on Atomic Pi
        header_up Host {upstream_hostport}
        header_up X-Real-IP {remote_host}
        header_up X-Forwarded-For {remote_host}
        header_up X-Forwarded-Proto {scheme}
    }
}
```

---

## Phase 2: Plane Deployment (The "Shared Resource" Strategy)

*Goal: Deploy Plane without doubling your RAM footprint by reusing MariaDB.*

**The Challenge:** Plane requires Redis and a Worker. Running this bare metal is a "toil" trap. We will use **Docker Compose** on the Atomic Pi, but **link it to your existing bare-metal MariaDB**.

**1. Database Preparation (Bare Metal)**
On the Atomic Pi, log into MariaDB:

SQL ⌄

```sql
CREATE DATABASE plane CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
CREATE USER 'plane'@'%' IDENTIFIED BY 'strong_password';
GRANT ALL PRIVILEGES ON plane.* TO 'plane'@'%';
FLUSH PRIVILEGES;
```

**2. Docker Compose (Atomic Pi)**
Create a `docker-compose.yml` for Plane. Note that we omit the DB container and point it to the host IP.

YAML ⌄

```yaml
version: "3.8"

services:
  redis:
    image: redis:alpine
    restart: always
    volumes:
      - redisdata:/data

  web:
    image: makeplane/plane:stable
    restart: always
    command: ./bin/web
    environment:
      - WEB_URL=https://plane.bitwise.ws
      # DATABASE_URL uses the bare metal MariaDB IP
      - DATABASE_URL=mysql://plane:password@192.168.1.XX:3306/plane
      - REDIS_URL=redis://redis:6379
    ports:
      - "3000:3000"

  worker:
    image: makeplane/plane:stable
    restart: always
    command: ./bin/worker
    environment:
      - DATABASE_URL=mysql://plane:password@192.168.1.XX:3306/plane
      - REDIS_URL=redis://redis:6379

volumes:
  redisdata:
```

---

## Phase 3: The n8n Glue (Docker + Python Integration)

*Goal: Build the automation bridge using the custom n8n image discussed earlier.*

**1. Build Custom n8n Image (on Desktop → Deploy to Atomic Pi)**
Since n8n is Node-based but you need Python libraries (`anytype-client`), stick to the Docker strategy.

- *FROM n8nio/*n8n
  USER root
  RUN apk add —no-cache python3 py3-pip
  RUN pip3 install anytype-client requests
  USER node

**2. n8n Configuration**
Configure n8n to connect to your **Git** (Forgejo) and **Matrix** (Tuwunel).

- **Matrix Integration:** Use the standard n8n Matrix node or a generic HTTP Request node to push notifications to your `tuwunel` room.
  - *Workflow:* Task Done in Plane → n8n → Matrix Message: "✅ Task [ID] Complete: [URL]".

---

## Phase 4: AnyType MCP (The "Headless" Knowledge Base)

*Goal: Enable the AI to access AnyType without loading the GUI on the Pi.*

Since the Pi 4 is your "Always On" node and runs bare metal services:

1. **Deploy Middleware:** Run the AnyType Middleware (usually a container or binary) on the Pi 4.
2. **Deploy MCP:** Run `anytype-mcp` (Python script) on the Pi 4.
3. **Access:** Configure your Caddy L4 module to forward `mcp.bitwise.ws` traffic to the Pi 4, or allow internal LAN access only from the Jetson (Open WebUI).

---

## Phase 5: Backup Strategy (resticprofile + MinIO)

*Goal: Ensure the new Docker state is captured without bloat.*

**1. Plane Backup (The "Config" Approach)**
Since Plane is using your bare-metal MariaDB, the **Data** is already covered if you are backing up the MariaDB datadir (`/var/lib/mysql`) via resticprofile.

- *New Addition:* You only need to backup the **Redis** snapshot and the **Docker Volume** (which holds the Redis data).
-
  ```
  global:
    priority: low
  default:
    password-file: /etc/restic/password
    repository: s3:http://minio.bitwise.ws:9000/restic
    env:
      AWS_ACCESS_KEY_ID: xxx
      AWS_SECRET_ACCESS_KEY: yyy

  # Backup the Plane Redis volume
  planesnapshots:
    backup:
      source: /var/lib/docker/volumes/planedata/_data
      schedule: "06:00"
    retention:
      keep-daily: 7
  ```

**2. n8n Backup**
n8n stores workflows in SQLite (by default) or MariaDB.

- *Best Practice:* Point n8n to use your existing MariaDB instance as well.
- *Benefit:* No new binary files to back up. The workflows live in the SQL dump of MariaDB. Zero extra backup logic required.

---

### Phase 6: ChatOps (Tuwunel Integration)

*Goal: Lower the barrier to entry for task management.*

Since you run **tuwunel** (Matrix), you can interface with your stack via chat.

**Workflow:**

1. **Matrix User → Bot:** "Create task in Plane: Update resticprofile."
2. **n8n → Matrix:** Reads message.
3. **n8n → Plane API:** Creates Issue "Update resticprofile".
4. **n8n → Matrix:** "Created Issue #42: https://plane.bitwise.ws/..."

**Why this fits you:**

- **Low Friction:** You don't need to open a browser to create a task.
- **Mobile Friendly:** You can manage tasks from your phone (Matrix client) without loading heavy web pages.
- **Text First:** Fits the CLI/Text preference of a Linux engineer.

### Summary Checklist

○ **DNS:** Add `plane` and `minio` to Pi-hole.

○ **DB:** Create `plane` database in bare-metal MariaDB.

○ **Docker:** Deploy Plane (linked to MariaDB) on Atomic Pi.

○ **Docker:** Build/Deploy custom n8n (with Python) on Atomic Pi.

○ **Config:** Point n8n DB to MariaDB (Centralize data).

○ **Backup:** Add Docker volumes to `resticprofile`.

○ **Caddy:** Add route for `plane.bitwise.ws`.

○ **MCP:** Set up AnyType MCP on Pi 4.

This plan utilizes your existing **Go/Rust binary** foundation (Caddy, Forgejo, Tuwunel) while pragmatically using Docker for the "modern web stack" components (Plane, n8n), all centralized on your efficient Atomic Pi.

> 💡 Just to be clear, Anytype is a desktop application and onlyu runs on the developers desktop machine ( Manjaro Linux ), the MCP will have to interface with this and it will only be active when the desktop is not hibernating.

Thank you for the clarification. This fundamentally shifts the "Brain" of the architecture from the "Always-On" Pi 4 to the "High-Power" Desktop.

This introduces a specific challenge: **The "High-Performance Node" is intermittent.**

- **When On:** You have maximum context (AnyType) and AI power (Desktop GPU/CPU).
- **When Off:** The "Context Loop" (AI accessing AnyType) breaks, and Automation (Syncing tasks to Plane) fails.

Here is the adjusted plan that treats your Desktop (Manjaro) as the "Knowledge Broker," integrating it with the rest of the cluster using WOL (Wake-on-LAN) and aggressive backup strategies.

### Revised Architecture: "Intermittent Brain"

| Service | Host | Status | Connectivity |
|---|---|---|---|
| **AnyType Client** | Desktop (Manjaro) | **Intermittent** (User Session) | Source of Truth |
| **AnyType Middleware** | Desktop (Manjaro) | **Intermittent** | Exposed via Caddy to LAN |
| **AnyType MCP** | Desktop (Manjaro) | **Intermittent** | Connects Open WebUI → AnyType |
| **Plane** | Atomic Pi | **Always On** | Project Management |
| **n8n** | Atomic Pi | **Always On** | The Bridge (Must handle Desktop being offline) |

---

### Phase 1: The Desktop Setup (Manjaro)

*Goal: Prepare the Desktop to be a server-grade client.*

**1. AnyType Middleware Access**
The AnyType app runs a local server/middleware for sync. We need n8n and the Jetson to reach it.

- **Action:** Configure AnyType to listen on all interfaces (0.0.0.0) or configure a port forward.

- **Security:** Ensure your Firewall (`ufw` or `iptables` on Manjaro) allows traffic from the `192.168.1.x` subnet (Atomic Pi, Jetson) to the AnyType port (default usually varies, check logs).

2. Install **`anytype-mcp` (The AI Bridge)**
Since this runs on Manjaro, you can install it via Python.

- **[Unit]**
  Description=AnyType MCP Server
  `After=network.target`

  `[Service]`
  `ExecStart=`/usr/bin/python3 /path/to/anytype-mcp/server.py
  Restart=always

  [Install]
  WantedBy=default.target
- **Result:** When you log in, the MCP service starts automatically in the background.

**3. Configure Open WebUI (on Jetson)**
The Jetson needs to know where to look for the MCP server.

- **Action:** In Open WebUI Settings → Providers → MCP:
  - **Name:** `LocalAnytype`
  - **Endpoint:** `http://<DESKTOP_IP>:<MCP_PORT>` (e.g., `http://192.168.1.20:8000`)

---

## Phase 2: The "Availability" Problem (Caddy + WOL)

*Goal: Ensure n8n and Jetson can wake the Desktop if needed, or fail gracefully.*

**1. Caddy Routing (on Pi 4)**
Update Caddy to route `anytype.bitwise.ws` to your Desktop.

Plain Text

```
anytype.bitwise.ws {
    reverse_proxy 192.168.1.20:<AnyType_Port> {
        header_up Host {upstream_hostport}
        header_up X-Real-IP {remote_host}
    }
}
```

**2. Wake-on-LAN (WOL) Integration**
Since your Desktop sleeps to save energy, the Atomic Pi (n8n) needs to wake it up to

sync tasks.

- **#!/bin/bash**
  **etherwake** `-i eth0 <Deskt`op_MAC_Address>
- **n8n Workflow: "The Heartbeat"**
  - **Trigger:** Every 15 mins.
  - **Node 1 (Check):** HTTP Request to `anytype.bitwise.ws/health` (if available) or ping the Desktop.
  - **Node 2 (Decision):**
    - *If Online:* Run the Sync Workflow.
    - *If Offline:* **Do Nothing** (don't wake it up unnecessarily) OR Send ntfy alert: "Desktop asleep - Task Sync Paused".

---

### Phase 3: Automation Resilience (n8n Logic)

*Goal: Handle the case where n8n tries to sync but the Desktop is off.*

**The Sync Workflow (Atomic Pi → Desktop):**

1. **Trigger:** Schedule (Cron).
2. **Check Desktop:** Ping Desktop IP.
   - *If False:* Stop Workflow. (Save the energy/cycles).
   - *If True:* Proceed.
3. **Fetch Data:** Call AnyType Middleware (via Caddy) → Get Tasks.
4. **Push to Plane:** Send to Plane API.
5. **Handle Error (Dropout):** If the Desktop goes to sleep *during* the sync (step 3), the HTTP Request will fail.
   - *Action:* Use **"Wait"** node (retry after 30s) 3 times. If it still fails, drop the workflow and try again in 15 mins.

**Crucial Note:** Do not use "Infinite Retry" in n8n. If the desktop is off for the night, you don't want n8n hammering it every 5 seconds.

---

### Phase 4: The "Data Loss" Prevention (Backup Strategy)

*Goal: Since the Desktop is the "Single Source of Truth" for your Knowledge Base, and it's a spinning mechanical drive (assuming older tech given the CPU specs), backups are critical.*

**1. resticprofile Configuration (Manjaro)**
You need to back up `~/.anytype` to MinIO.

- **Frequency:** Every 2 hours (incremental).
- **Hook:** Run a backup *before* suspend.

**2. Suspend Hook (Systemd)**
Create a systemd service on Manjaro that runs a quick backup when you close the lid or click suspend.

-
  ```
  [Service]
  Type=oneshot
  ExecStart=-/usr/local/bin/backup_before_sleep.sh
  ```
- *Why:* If you have a brilliant idea, write it down, close your laptop, and the drive fails on wake-up, your idea is safe in MinIO.

---

### Phase 5: ChatOps (Tuwunel) & Remote Work

*Goal: How do you manage tasks if you are on your laptop (not the Desktop)?*

Since AnyType is locked to the Desktop, you cannot easily read notes from your laptop.

- **Workaround:** Use **Plane** as the "Mobile View".
- **Workflow:**
  - You are on Laptop. You need to know what to do.
  - You check **Plane** (always on Atomic Pi, accessible via VPN/Merlin).
  - You see your tasks.
  - You mark them done.
  - **Async Sync:** When you get back to the Desktop, n8n wakes up, pulls the "Done" status from Plane, and updates your local AnyType instance.

### Summary of the "Desktop-Centric" Flow

1. **Morning:** You wake up. You turn on Desktop (Manjaro).
2. **Start-up:** Systemd starts AnyType and MCP Server.
3. **Sync:** n8n (Atomic Pi) detects Desktop is online. It pulls tasks from AnyType and pushes them to Plane.
4. **Work:**

    - You work in **Rabbitholes** (Jetson). It queries AnyType MCP (Desktop) for context.
    - You track progress in **Plane** (Browser).
    - n8n keeps them in sync.
5. **End of Day:** You close Desktop.
    - Hook triggers backup to MinIO.
    - n8n marks Desktop as "Offline" and pauses syncing.
    - Plane continues to run on Atomic Pi (accessible from phone/laptop), holding the current state of the project.

This plan respects the hardware constraints and the "Brain on Desktop" reality while ensuring you don't lose data or sync capability.

> 💡 Rabbitholes is also a desktop application that is not a service. We can assume all development is done through the desktop and there is no remote "development" work other than maintenance and reporting.

This is the cleanest architectural model yet. By treating the Desktop as the **Workstation** and the SBCs as **Infrastructure/Compute**, we eliminate the friction of working "remotely."

We are building a **Local High-Performance Cluster** where your Desktop is the "User Node," the Jetson is the "Compute Node," and the Atomic Pi is the "Service Node."

### The Final Architecture: "Local Cloud"

| Node | Role | Services | State |
|---|---|---|---|
| **Desktop (Manjaro)** | **Creator** | AnyType, Rabbitholes, `anytype-mcp` | "On" when you work. |
| **Jetson Orin** | **Compute** | Open WebUI (LLM), iopaint | "Always On" (Low power idle). |
| **Atomic Pi** | **Infrastructure** | Plane, n8n, Forgejo, MariaDB | "Always On". |
| **Pi 4 / Router** | **Network/Storage** | Caddy, MinIO, Pi-hole | "Always On". |

---

### Phase 1: The "AI Loop" (Desktop + Jetson)

*Goal: Connect your interface (Rabbitholes) to your brain (Jetson) and your memory (AnyType).*

This loop relies on LAN communication between the Desktop and the Jetson.

**1. The Desktop Setup (Manjaro)**

- **Service 1: AnyType Middleware:** Ensure it's running.
- Service 2: **`anytype-mcp` Server:**
  - Since this needs to be reachable by the Jetson, configure it to listen on the LAN IP (e.g., `0.0.0.0:8000`).
  - *Systemd:* Create a user service to auto-start this when you log in.
- **Service 3: Rabbitholes:**
  - Configure **Custom Base URL / API Endpoint**.
  - Point it to your Jetson's Open WebUI instance: `http://192.168.1.60:3000` (or whatever port Open WebUI uses).

**2. The Jetson Setup (Compute)**

- **Service 1: Open WebUI:**
  - **MCP Configuration:** This is the critical bridge.
  - In Open WebUI settings, add the MCP server running on your Desktop:
    - **Name:** `AnyType-LAN`

    - **Endpoint:** `http://192.168.1.20:8000/sse` (or your specific MCP port).
- **Why this works:**
  1. You type a prompt in **Rabbitholes**.
  2. Rabbitholes sends it to **Jetson** (Open WebUI).
  3. The AI needs context. It calls the **MCP Server**.
  4. The request goes over LAN to your **Desktop** (`anytype-mcp`).
  5. Desktop fetches data from local AnyType and returns it to Jetson.
  6. Jetson generates the response and sends it back to Rabbitholes.

---

### Phase 2: The "Persistence Loop" (Desktop + Atomic Pi)

*Goal: Automate the flow of ideas (Desktop) into tasks (Atomic Pi).*

Since development is local, the Atomic Pi (n8n) acts as a passive listener until the Desktop wakes up.

**1. The Wake/Sync Cycle (n8n on Atomic Pi)**

- **Workflow Logic:**
  1. **Trigger:** Every 30 mins (or wake from WOL).
  2. **Ping:** Can you reach `192.168.1.20` (Desktop)?
     - *No:* Sleep.
     - *Yes:* Proceed.
  3. **Sync:**
     - Get "Backlog" items from AnyType (via Caddy/Tunnel).
     - Create/Update issues in Plane.
  4. **Reverse Sync:**
     - Check Forgejo/Plane for "Completed" items.
     - Update AnyType status to "Done".

**2. Caddy Routing (Pi 4)**
Ensure Caddy has a route to allow the Atomic Pi to reach the Desktop (though direct LAN IP is fine, Caddy provides SSL/Hardening if the payload is sensitive).

- *Recommendation:* For internal LAN traffic (n8n → Desktop), direct HTTP is usually fine, but since you are security conscious, routing `anytype.internal.bitwise.ws` through Caddy on the Pi to the Desktop is cleaner.

---

### Phase 3: Development Workflow (The "Happy Path")

*How a typical work session looks in this environment.*

**1. Boot Up**

- You turn on the Desktop.
- Systemd starts AnyType and `anytype-mcp`.
- Desktop connects to LAN.

**2. Ideation (The RAM)**

- You open **Rabbitholes**.
- You start researching. The Jetson handles the LLM inference.
- You ask: "What was the architectural decision for the API key?"
- Rabbitholes → Jetson → Desktop (AnyType) → **Answer**.
- You synthesize this into a "Task Object" inside AnyType. Status: `Backlog`.

**3. Execution (The Handoff)**

- You open **Plane** (in browser).
- Wait 30 mins (or manually trigger n8n).
- n8n detects the new AnyType Task → Creates Issue in Plane.
- You drag the Issue to `In Progress`.

**4. Coding**

- You open VS Code (Desktop).
- You push code to **Forgejo**.
- You link the commit to the Plane Issue (via native Git integration).

**5. Shutdown**

- You close your apps.
- Restic backup runs (hooked to systemd suspend).
- Desktop sleeps.

**6. Mobile/Remote Check (Maintenance Mode)**

- You are on your phone (VPN connected).

- You open **Plane** (hosted on Atomic Pi) via `plane.bitwise.ws`.
- You check the project status. Everything is there, synced from your desktop session.
- You **cannot** access AnyType notes (Desktop is off), but you can manage the *tasks* and *code*.

---

### Phase 4: Backup & Safety (Restic + MinIO)

*Goal: Protect the data that lives on the unreliable/intermittent Desktop.*

**1. The "Pre-Sleep" Hook**

- On Manjaro, create a systemd service that runs before suspend:
  - `ExecStart=/usr/bin/resticprofile -config /etc/resticprofile/profiles.conf backup`
- This ensures your latest AnyType notes (which are *not* on the Pi) are snapped to MinIO before the power cuts.

**2. Off-Site (Disaster Recovery)**

- **MinIO (Local):** Immediate access.
- **AWS Glacier (Eventually):** As planned.
- **Workflow:** Once a week, a cron job on the **Atomic Pi** (which is always on) pushes the contents of the MinIO bucket to Glacier.

### Summary of Network Traffic

1. **Desktop → Jetson:** Prompt / Inference Request.
2. **Jetson → Desktop:** MCP Context Query.
3. **Atomic Pi → Desktop:** Sync Requests (Polling).
4. **Atomic Pi → Pi 4:** Database Storage.
5. **All Devices → Pi 4:** DNS (Pi-hole) / Gateway (Caddy).

This setup maximizes the performance of your Desktop while offloading the battery-draining AI inference to the Jetson and the maintenance overhead to the Atomic Pi. It is a cohesive, energy-efficient "Personal Cloud."

> Is it possible to present this solution we have developed "visually" in a diagram or info graphic that I can talk to with people not familiar with the

> details of some of the implementation, so that they have some idea about what is being done?

Here is a high-level visual representation of your architecture using **Mermaid**. This diagram is designed to be understood by non-technical stakeholders. It focuses on **Data Flow** and **Function** rather than specific code implementations.

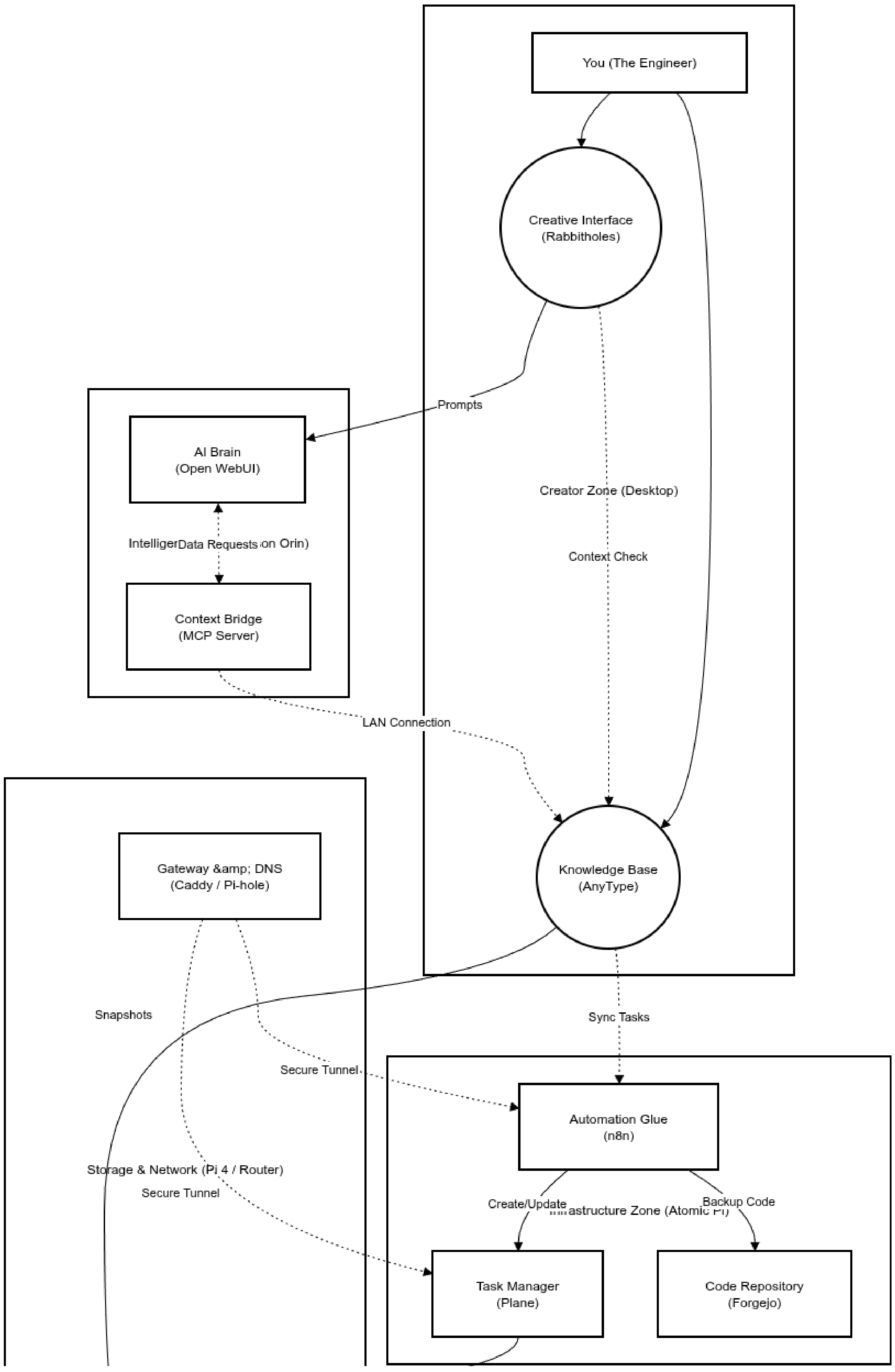
You can paste the code block below into any Markdown viewer (like GitHub, GitLab, or Notion) to see the rendered diagram.

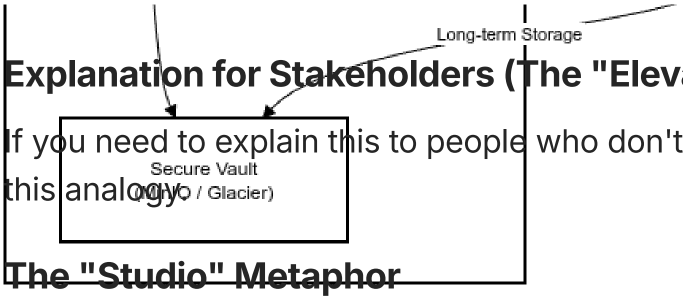
## The Visual Architecture

Mermaid

```mermaid
graph TB
    %% Styles
    classDef user fill:#ffcccc,stroke:#333,stroke-width:2px
    classDef workstation fill:#e6f2ff,stroke:#004080,stroke-width:2px
    classDef compute fill:#e6ffe6,stroke:#006600,stroke-width:2px
    classDef infra fill:#f0e6ff,stroke:#4b0082,stroke-width:2px
    classDef cloud fill:#f9f9f9,stroke:#999,stroke-dasharray: 5 5

    subgraph User_Environment ["Creator Zone (Desktop)"]
        User["You (The Engineer)"]
        AnyType(("Knowledge Base<br/>(AnyType)"))
        Rabbitholes(("Creative Interface<br/>(Rabbitholes)"))

        User --> Rabbitholes
        User --> AnyType
        Rabbitholes -.->|Context Check| AnyType
    end

    subgraph AI_Environment ["Intelligence Zone (Jetson Orin)"]
        OpenWebUI["AI Brain<br/>(Open WebUI)"]
        MCP_Bridge["Context Bridge<br/>(MCP Server)"]

        Rabbitholes -->|Prompts| OpenWebUI
        OpenWebUI <-.->|Data Requests| MCP_Bridge
        MCP_Bridge -.->|LAN Connection| AnyType
    end

    subgraph Service_Environment ["Infrastructure Zone (Atomic Pi)"]
        n8n["Automation Glue<br/>(n8n)"]
        Plane["Task Manager<br/>(Plane)"]
        Forgejo["Code Repository<br/>(Forgejo)"]

        AnyType -.->|Sync Tasks| n8n
        n8n -->|Create/Update| Plane
        n8n -->|Backup Code| Forgejo
    end

    subgraph Data_Environment ["Storage & Network (Pi 4 / Router)"]
        Backup["Secure Vault<br/>(MinIO / Glacier)"]
        Gateway["Gateway & DNS<br/>(Caddy / Pi-hole)"]

        Plane -->|Long-term Storage| Backup
        AnyType -->|Snapshots| Backup
        Gateway -.->|Secure Tunnel| n8n
        Gateway -.->|Secure Tunnel| Plane
    end

    %% Class assignments
    class User user
    class AnyType,Rabbitholes workstation
    class OpenWebUI,MCP_Bridge compute
    class n8n,Plane,Forgejo infra
    class Backup,Gateway cloud
```

---


You (The Engineer)
Creative Interface
(Rabbitholes)
Prompts
AI Brain
(Open WebUI)
Creator Zone (Desktop)
Context Check
Intelliger
Data Requests
on Orin)
Context Bridge
(MCP Server)
LAN Connection
Gateway &amp; DNS
(Caddy / Pi-hole)
Knowledge Base
(AnyType)
Snapshots
Sync Tasks
Secure Tunnel
Automation Glue
(n8n)
Storage & Network (Pi 4 / Router)
Secure Tunnel
Create/Update
Infrastructure Zone (Atomic Pi)
Backup Code
Task Manager
(Plane)
Code Repository
(Forgejo)



## Explanation for Stakeholders (The "Elevator Pitch")

If you need to explain this to people who don't know what "MCP" or "Docker" is, use this analogy:

## The "Studio" Metaphor

This system is designed like a high-end production studio, optimized for a single expert creator (You).

### 1. The Creator Zone (Your Desktop)

- **What it is:** This is your main workstation where the actual creative work happens.
- **The Tools:**
    - **AnyType:** This is your digital filing cabinet. It's where you store your thoughts, plans, and ideas. It lives on your main computer.
    - **Rabbitholes:** This is your workbench. It's the interface you use to brainstorm and research.

### 2. The Intelligence Zone (The Jetson)

- **What it is:** Think of this as a "super-intelligent assistant" that sits next to you. It is a powerful computer dedicated to helping you think.
- **How it works:** When you ask a question in Rabbitholes, the Jetson handles the complex thinking. Crucially, it has a "phone line" (MCP) to your filing cabinet (AnyType), so it can answer questions based on your *own* notes, not just the internet.

### 3. The Infrastructure Zone (The Atomic Pi)

- **What it is:** This is the "Project Management Office." It’s a small, efficient server that runs in the background 24/7.
- **Why it's there:**
    - **Plane:** This is your project tracker. It takes the ideas from your filing cabinet and turns them into actionable to-do lists.
    - **Automation (n8n):** This is the butler. When you add a task to your filing cabinet, the butler notices and automatically moves it to your project tracker so you don't forget it.

### 4. The Storage & Network (Pi 4 / Router)

- **What it is:** This is the foundation. It keeps the internet safe (Pi-hole), manages the keys to the door (Caddy), and stores the backups (MinIO).

### Summary for the Audience

> "I have built a system that allows me to work creatively on my main computer, while offloading the heavy-duty thinking and organization to specialized, low-energy devices in the background. It connects my **ideas** (AnyType) with my **AI assistance** (Jetson) and my **tasks** (Plane) automatically, ensuring nothing gets lost and minimizing my mental load."