ChatGPT

# Automated Home Lab Orchestration Plan

## Introduction

This document outlines a comprehensive plan for building an **automated home lab orchestration system**. The goal is to create a stable homelab environment that can **spin up and tear down services on demand** for learning, experimentation, and portfolio-building purposes. The setup will leverage Infrastructure-as-Code (IaC) practices, using tools like **Proxmox VE** for virtualization, **Terraform** for resource provisioning, **Packer** for image templating, and **Ansible** for configuration management. By automating the deployment of various self-hosted services (from media servers to productivity tools), we ensure a reproducible and documented process. The end result will be a personal infrastructure that not only supports home use (e.g. replacing third-party cloud services or streaming platforms) but can also showcase solutions valuable to small businesses (e.g. file sharing, automation, and AI-driven insights). All steps and configurations will be well-documented, forming the basis of blog articles or a portfolio site to demonstrate our skills and lessons learned.

## Objectives

- **Infrastructure as Code**: Define the homelab's infrastructure in code for consistency and easy rebuilds. This includes automating VM creation (rather than using Proxmox GUI repeatedly) to avoid manual, repetitive work [1]. Tools like Terraform will be evaluated to provision VMs/containers in Proxmox from templates.
- **Automated Configuration**: Use Ansible playbooks to configure VMs and install services automatically. Each service (Nextcloud, media servers, etc.) will have an Ansible role/playbook, enabling one-command deployments and ensuring identical setups every time [1].
- **On-Demand Service Deployment**: Enable quick spin-up or shutdown of various services (e.g. deploy a new app for testing, then remove it) to practice and experiment. This includes having a library of ready-to-use scripts or playbooks (leveraging resources like Proxmox VE Helper Scripts by the community [2] and the Awesome Self-Hosted list of applications) to deploy common services efficiently.
- **Secure Networking & Access**: Implement best practices for network configuration, including a **reverse proxy with SSL** termination for safe external access to services, and an **internal DNS** solution for convenient service discovery on the LAN. All web services should be reachable via friendly URLs (with proper SSL certificates) rather than IPs and ports.
- **Local AI and Automation**: Integrate a local Large Language Model (LLM) and automation workflows to augment the homelab's capabilities. This includes deploying a local AI assistant that can utilize our knowledge stores (Obsidian vault notes, etc.) via Retrieval-Augmented Generation, and setting up automation pipelines for tasks like data extraction (e.g. transcribing and summarizing videos) and email/text parsing. The AI should be accessible through convenient interfaces (perhaps a chat UI or even a Discord bot for voice/text commands) to perform tasks agentically on the homelab.
- **Self-Hosted Replacements for Cloud Services**: Provide self-hosted alternatives to common cloud and entertainment services to increase privacy and self-reliance. For example, use Nextcloud instead

of Google Drive [3], Jellyfin/Plex instead of Netflix [4], etc. This not only offers learning opportunities but also ensures the homelab remains useful even during internet/cloud outages [4].
- **Business-Oriented Services**: Beyond personal use, deploy and test platforms that have business value – such as project management, CRM, or automation tools – to build experience with solutions that could be offered professionally. The homelab can serve as a demo environment for these open-source business applications, adding credibility to a personal tech portfolio.
- **Comprehensive Documentation**: Meticulously document each step, configuration, and troubleshooting process. This will be done in an Obsidian vault (for personal records) and also published (as blog posts or a knowledge base) to demonstrate our work publicly. Documentation will not only cover the "how" but also the "why" of design decisions and best practices followed.

## Architecture & Tooling Overview

**Proxmox VE (Virtualization Platform)** – The foundation of the homelab. Proxmox will host all Virtual Machines (VMs) and Linux Containers (LXC) that run our services. We already have Proxmox running; it will be the target for our automated provisioning. Proxmox's robust API allows external tools to create and manage VMs programmatically. This means instead of manually clicking through the Proxmox GUI to create VMs, we can automate it via scripts or Terraform [1], saving time and ensuring repeatability.

**Terraform (IaC Provisioning)** – *Optional, but recommended for codifying infrastructure*. Terraform, using the Proxmox provider, can define Proxmox resources (VMs, containers, networks) in declarative configuration files [1]. If used, we will write Terraform configs for each VM (specifying parameters like base image, CPU/RAM, networking, etc.). Terraform can then create or destroy those VMs on Proxmox as needed. This approach streamlines VM creation for repetitive tasks: "Terraform and Ansible come in handy, as they let me deploy and configure virtual machines using reusable configuration files" [1]. We'll evaluate if Terraform adds value in our case (since Proxmox is a single node and not cloud-scale, some homelabbers script directly with the API). If not using Terraform, we may use Proxmox's CLI or API in custom scripts to achieve similar IaC benefits. The key idea is to **avoid manual VM setup** beyond the initial Proxmox host install.

**Packer (Image Templating)** – We will use HashiCorp Packer to build standardized VM images (templates) for quick deployment. Using Proxmox's support for **Cloud-Init**, we can create a template VM (e.g., Ubuntu Server) that is pre-configured to our liking (user accounts, SSH keys, basic tools) and then convert it to a template. Packer can automate this process: it will create a VM on Proxmox from an ISO, run provisioning steps (using shell commands or Ansible) to install and configure the OS, then output it as a reusable template [5] [6]. This means whenever we need a new VM, we can just clone from this template in seconds rather than installing an OS from scratch each time. *For example, a workflow might be:* Packer builds an Ubuntu 22.04 template with cloud-init enabled; Terraform then clones that template for each new service VM, injecting a cloud-init config (hostname, IP, etc.); finally Ansible does application setup. This Packer->Template approach ensures consistency across all VMs. (In practice, Packer can also directly integrate Ansible as a provisioner to pre-install certain packages in the image, but we'll likely keep most app installation in the Ansible playbooks for flexibility.)

**Ansible (Configuration Management)** – Ansible is crucial for automating the setup of services on each VM or container. We will write **Ansible playbooks** or roles for each major service (e.g., one for Nextcloud server, one for a media downloader, etc.). Once a VM is provisioned (via Terraform or manually), Ansible will: update the system, install required packages or Docker containers, configure the application, set up users, etc. This approach transforms one-off setup steps into code. According to a homelab case study, using Terraform with Ansible allowed an author to “automatically spin up and configure disposable VMs” much faster than hand-configuration [7]. We aim for a similar outcome: a library of playbooks that can set up any service in our catalog with minimal manual intervention. We may also incorporate **Ansible Semaphore** (a web UI for Ansible) or CI pipelines to trigger playbooks easily [7], providing a user-friendly way to deploy services. The Ansible inventory can be dynamic, pulling host details from Proxmox or Terraform outputs.

**Proxmox VE Helper Scripts** – In addition to our own Ansible roles, we will take advantage of community automation where sensible. **Proxmox VE Helper-Scripts** (by tteck and the community) provide over 400 scripts to streamline homelab tasks on Proxmox [2]. These scripts can create LXC containers pre-configured for many popular self-hosted apps (e.g., a one-liner to deploy a Nextcloud LXC, etc.). We might use these scripts as references or even directly for quick wins, especially for complex apps. For example, if there’s a script to set up an LXC with Frigate NVR or Home Assistant, we can leverage it to save time and then incorporate the result into our Ansible workflow. The philosophy is not to reinvent the wheel – these helper scripts can bootstrap certain services quickly, which we can then further customize via Ansible if needed.

**Awesome Self-Hosted Applications** – We will maintain a curated list of target applications inspired by the **Awesome-Selfhosted** community list. *Awesome-Selfhosted* is a well-known collection of free software and web applications that can be self-hosted, covering everything from media streaming to business tools. This will guide what we choose to deploy. Our focus will be on applications that either replace common consumer services (media, cloud, home automation) or provide capabilities useful to businesses (project management, knowledge bases, etc.). By referencing this list, we ensure we’re aware of the best-in-class open-source options for each category.

**Version Control & Collaboration** – All code (Terraform configs, Packer template definitions, Ansible playbooks, etc.) will be kept in a Git repository. This provides change tracking and the ability to collaborate or use CI/CD. For instance, we could have a GitHub (or local Gitea) repository for our homelab IaC; pushing changes could trigger a pipeline that applies Terraform or runs Ansible. While a full CI/CD pipeline is optional for a personal lab, using Git will impose structure and enable rollback if something breaks. It also makes it easier to share our setup or open-source parts of it for the community.

**Orchestration AI Agent (Blueprint & Sub-Agents)** – *(Meta aspect of the project)* We plan to experiment with an AI “blueprint” agent to coordinate the build-out of this lab. The idea is to feed the above plan (or a refined prompt of it) to an AI orchestrator which can then break the work into tasks and perhaps generate code or configuration via sub-agents working in parallel. For example, one sub-agent could start writing an Ansible role for Nextcloud while another prepares the Terraform file for a Nextcloud VM, etc., under the direction of a central planning agent. This approach, while experimental, could speed up development and also demonstrate cutting-edge use of AI in DevOps. The prompt you’re reading is intended as that initial input for the blueprint agent, describing the desired end state and requirements in detail. The orchestrator will then create parallel work streams based on these requirements. (Even if we end up doing most of it manually, this is a great opportunity to explore AI-assisted infrastructure as a side project.)

## Networking and Access Configuration

A solid networking setup is critical for a multi-service homelab:

- **Reverse Proxy with SSL**: All external access to our services will be routed through a central reverse proxy. This proxy will handle HTTPS encryption (using Let's Encrypt certificates) and route requests based on domain name to the appropriate internal service. We have two primary options here:
- *Nginx Proxy Manager (NPM)* – a user-friendly web UI on top of Nginx that makes it easy to add proxy hosts and obtain SSL certs via Let's Encrypt. NPM is excellent for quick setups and is beginner-friendly. However, since we want to embrace Infrastructure-as-Code, NPM's click-ops approach could become a limitation (its config is stored in a database and not as easily managed via Git).

- *Traefik* – a cloud-native, file-driven reverse proxy. Traefik integrates very well with Docker and dynamic environments. We can define Traefik routing rules and SSL settings in YAML or as labels on Docker containers, which means our **certificate and routing configuration stays in code** rather than only in a UI [8]. As one homelabber noted, using Traefik allowed them to treat SSL and proxy config as part of their Docker Compose files and version control everything, whereas Nginx Proxy Manager required manual GUI configuration outside of code [8]. Given our automation focus, we are inclined to use Traefik for this project, possibly running in a Docker container or an LXC. Traefik will automatically fetch/renew SSL certs and route traffic to services (by hostname or subdomain). We'll still have a dashboard to monitor it, but configuration can be through a Git-tracked file.
  For initial simplicity, we might start with Nginx Proxy Manager to get things working (since it's quick to set up via a container and offers a simple UI for our internal/external DNS names). But as we scale or if we want a pure IaC pipeline, migrating to Traefik could be beneficial.

- **Internal DNS**: We will set up an internal DNS resolver to handle local domain names for our services (e.g., `nextcloud.home.lab` resolving to the internal IP of the Nextcloud VM). This avoids relying on external DNS for internal traffic and lets us use nice URLs even when offline. A popular choice is **Pi-hole** combined with **Unbound** (Pi-hole for DNS-level ad blocking and local DNS management, Unbound as a recursive DNS resolver for privacy) [9]. Pi-hole provides a web interface where we can define local DNS entries (or override external ones) and see all DNS queries on the network. Unbound, when paired with Pi-hole, ensures DNS queries don't go to third-party servers but instead are resolved from the root DNS servers directly, improving privacy [9]. This combination would give us network-wide ad blocking (useful for all devices) and full control of internal name resolution. We will run Pi-hole + Unbound likely in a lightweight container or VM. If ad-blocking is not a priority, we could alternatively use just **Unbound or Dnsmasq** for DNS, but including Pi-hole is a nice quality-of-life addition for a homelab.
  Additionally, if we register a custom domain for our homelab (for example, `mydomain.com` with subdomains for services), we could use a split-horizon DNS (internal DNS resolves those to local IPs, while external DNS might point to our public IP). But since this is a home setup, we might simply use something like `.local` or `.lan` TLD internally.

- **Network Topology**: Proxmox allows creating virtual networks; we'll likely have at least one **bridged network** that connects VMs to the home LAN (so they get an IP in our main network). This is necessary so that services can be accessed by other devices and so that the reverse proxy (which might run on a VM or container on Proxmox) can reach them. If security segmentation is needed, we could create isolated VLANs for certain groups (e.g., cameras on one VLAN, IoT devices on another, etc., with firewall rules between), but initially, it might all reside in one trusted network for simplicity. We will ensure the Proxmox host's firewall is configured (Proxmox has a built-in firewall that can apply at VM or host level) to restrict access to management interfaces from outside. For external access, if our ISP doesn't provide a static IP or if behind CG-NAT, we might use a VPN or Cloudflare Tunnel – but assuming we can do standard port-forwarding, the reverse proxy will listen on ports 80/443 forwarded from the router. Certificates (Let's Encrypt) might require DNS challenge if behind NAT, but that's detail to solve later.

- **Access & Authentication**: We will enforce authentication on services appropriately. For instance, if some services are meant to be accessed over the internet (e.g., a personal Nextcloud or Vaultwarden), we will secure them with strong passwords and possibly two-factor auth. We might set up a **single sign-on (SSO)** solution if managing many user accounts becomes cumbersome – e.g., using **Authelia** or **Keycloak** as an identity provider to gate access to web apps. This could be overkill for one user, but it's a nice enterprise-like feature to explore. At minimum, enabling OAuth or SSO for services that support it could streamline logins.

- **Monitoring and Logging**: For networking and services, we should implement basic monitoring – perhaps **Uptime Kuma** (a self-hosted uptime monitor) can run to ping our services and alert if something is down. Proxmox itself can send email alerts for issues. If we get ambitious, setting up **Prometheus + Grafana** for metrics or a centralized log system (ELK/Opensearch stack) would be educational, but might be phase 2.

## Core Service Deployments

We categorize the services to deploy into several groups: **Storage/Cloud**, **Media**, **Home Automation/ Security**, **Productivity/Business**, and **AI/Knowledge**. Each service will be automated via Ansible (or scripts) and run either in its own VM or a container, depending on resource needs and isolation requirements.

### 1. Storage & Cloud Services

- **Network Storage (NAS)**: A reliable NAS solution will manage our disks and shares. We plan to use *OpenMediaVault (OMV)* or a similar lightweight NAS OS in a VM to handle storage pooling (RAID/ZFS) and share files over SMB/NFS. OMV provides a friendly web UI to create network shares, manage users/permissions, and even Docker containers. This will be the backbone for storing: media files (for Plex/Jellyfin), camera footage (for Frigate), backups of configurations, etc. An alternative is to use **TrueNAS** (for ZFS), but TrueNAS is heavier; OMV might be sufficient for our scale. The NAS VM will pass-through our data disks (Proxmox allows passing physical disks or using ZFS pools). We'll automate as much of OMV's setup as possible (perhaps using OMV's Ansible role or omv-salt stack, or simply document manual steps if needed).

- **File Sync and Personal Cloud**: **Nextcloud** will be deployed to replace services like Dropbox/Google Drive and to provide a web-accessible personal cloud storage [3]. Nextcloud offers file syncing, sharing, and an ecosystem of plugins (calendar, contacts, tasks, notes, even built-in video chat and collaborative document editing through Collabora or OnlyOffice). It's a cornerstone of a self-hosted toolkit. We'll create a VM (or LXC container) for Nextcloud, likely using an Ubuntu base. Through Ansible, we'll install the LAMP/LEMP stack needed (or use Docker Compose – though running Nextcloud on bare metal PHP might be fine). We will integrate Nextcloud with our NAS storage (e.g., mount the NAS storage into Nextcloud as external storage for big files). Also, consider Nextcloud's **Office integration**: we might deploy *Collabora Online* in a separate container to allow online editing of documents in Nextcloud – demonstrating a self-hosted alternative to Google Docs [3]. Nextcloud will be accessible via our reverse proxy at e.g. `cloud.mydomain.com` with HTTPS. We'll also harden it (strong passwords, fail2ban, etc.). In our docs, we can highlight that while Office365 was down, our Nextcloud + Collabora kept working [10], emphasizing the resilience benefit of self-hosting.

- **Backup Solutions**: To protect data, we should set up some backup mechanism. For the VMs, **Proxmox Backup Server (PBS)** can be used if we have spare storage – it's an enterprise-grade incremental backup system for VMs and containers. We could run PBS as a container or VM, or simply use Proxmox's built-in backup to a NAS directory. For application data, we might use Nextcloud's built-in backup or have periodic rsync snapshots of important volumes (databases, etc.). While not explicitly requested, demonstrating an automated backup (and restore test) in the lab would be a good practice (businesses care about backups). This could be a future addition.

## 2. Media & Entertainment Suite

One major objective is to **replace commercial streaming services** by hosting our own media library. This involves multiple components working together (often referred to collectively as the "*arr" suite and a media server):

- **Automated Downloaders (Sonarr/Radarr/Lidarr/etc.)**: We will deploy the suite of *Arr applications for content acquisition:
- **Sonarr** for TV shows (automatically finds new episodes of series we follow),
- **Radarr** for movies,

- **Lidarr** for music, and possibly **Readarr** for e-books/audiobooks.
These apps manage indexes of media and can trigger downloads from various sources. Each will run as a service (likely in Docker containers, since the *arr ecosystem runs well in Docker). We can either have a single VM (called "media automation VM") running all these containers, or split them up. Given resources, one VM with Docker Compose for all* arr apps is convenient. These apps require a **search indexer** to find content – we will deploy **Jackett** or **Prowlarr** as well. *Jackett* can aggregate many torrent indexers, and *Prowlarr* is a newer unified indexer manager that can feed all *arrs. We might prefer Prowlarr for its integration.*
*The* arr suite will be configured to use a download client and to sort files to our media library. This entire pipeline can be automated such that we specify a show or movie once, and new content is fetched and added to our library without manual steps [11]. Our Ansible playbook will configure each *arr with the proper settings (indexer API keys, quality profiles, library paths, etc. – some sensitive info we'll input via vaulted variables).

- **Download Clients and VPN**: To actually download the media, we'll set up a **BitTorrent** and/or **Usenet** client. Likely **qBittorrent** (in a container) for torrents, and perhaps **NZBGet** or **SABnzbd** for Usenet (if we use Usenet). These will download the files that Sonarr/Radarr request. Because downloading via public internet has privacy and ISP-throttling considerations, we will run these clients behind a VPN. A popular solution is the **Gluetun** container, which acts as a VPN gateway for other containers. We will deploy Gluetun with a reputable VPN provider and route qBittorrent's container network through it (so all torrent traffic is encrypted via VPN). This prevents our ISP from seeing our torrent activity and avoids disclosing our home IP on torrent swarms. We might configure it such that when the VPN is down, it kills the traffic (for safety). This is a standard setup in many homelabs for media servers.

- **Media Library Server**: For serving and streaming the collected media to our devices and TVs, we'll deploy a media server like **Jellyfin** or **Plex**. Jellyfin is fully open-source and free (which aligns with our open-source ethos), whereas Plex has some proprietary aspects but is very polished. We'll likely choose **Jellyfin** to avoid any proprietary lock-in (and because it supports streaming music and videos without requiring internet). Jellyfin will point to the media folders where Sonarr/Radarr put the completed downloads (e.g., a movies folder on the NAS, a TV folder, etc.). We'll configure Jellyfin with users for family, etc., and use hardware acceleration if available (our Proxmox host's GPU can be passed through or used via VAAPI for transcoding). Jellyfin will be accessible via web and apps for streaming our content anywhere. As an example of its value: during a recent cloud outage, a homelabber noted *"I do almost all of my movie and TV show streaming with my Jellyfin server... no third party servers that can bring it down"* [4] . We expect similar independence – our media is available even if Netflix or others go down (or if we cancel those services). In addition, we might deploy **Kodi** on local devices or use Jellyfin's integration with player apps for a seamless experience.

- **Media Request/Notification Bot**: To integrate with our social/communication channels (like Discord) and make the media system more "AI agentic," we will set up tooling for requesting and managing content. One option is **Ombi** or **Overseerr** – these are web apps that allow users to request new movies or shows which then send the request to Sonarr/Radarr. They also support notifications. Alternatively, since the user specifically mentioned Discord and "telling an AI to go look for specific items," we can create a custom Discord bot or use something like **Notifiarr**. Notifiarr is a service that ties into the *arr apps and can relay notifications (to Discord, Pushover, etc.) and accept commands. We could configure Notifiarr to post updates (download completed, etc.) to a Discord channel and potentially listen for commands like "!search movie <title>". Another approach: use n8n* or a custom script with Discord's API to watch a channel for requests and then add those to Radarr via its API. The AI angle might be: having a local LLM monitor Discord chat and interpret a natural language request ("download the latest season of X show") then trigger the appropriate API calls. This might be complex initially, so we can start with simpler webhook-based commands.

- **Library Management**: We won't forget **Calibre** for e-books management, as mentioned. We can run a Calibre server (or Calibre-Web, which is a nicer web UI) to manage and serve e-books/PDFs and even audio books. This, combined with *Readarr* (which can fetch e-books and audio books similar to Sonarr/Radarr), gives a complete self-hosted Kindle/Audible alternative. Our homelab could then provide an interface to browse and download books in our collection. We will attempt to automate Calibre setup (perhaps via a container), and connect it with our Nextcloud or Jellyfin (Jellyfin can serve audiobooks and ebooks as well).

Overall, the media suite will demonstrate an end-to-end pipeline: from content discovery to consumption, fully automated. By the end, we should be able to say goodbye to Netflix, Hulu, Spotify, Kindle, etc., and use our own infrastructure for these needs. The process of setting this up will be documented, noting any challenges (like organizing storage, dealing with ISP speeds, etc.). It's also a great showcase to others: for instance, explaining how self-hosting Jellyfin and Nextcloud kept one user's services up when big providers had outages [4] [10] will resonate in our eventual blog write-ups.

## 3. Home Automation & Security

This category elevates the homelab from serving data to also interacting with the physical world (IoT devices, cameras, etc.), as well as ensuring security of our home.

- **Home Assistant (Automation Hub)**: We will deploy **Home Assistant (HA)** as the central brains of home automation. Home Assistant is an open-source platform that can integrate with thousands of smart devices, allowing us to create complex automations and control devices from a unified interface. Given we plan "automations and cameras" and possibly custom behaviors (like facial recognition, sensor-driven actions), Home Assistant is ideal. It can run in a Proxmox LXC or VM (the Home Assistant OS or the supervised version for ease of add-ons). We prefer a VM running Home Assistant OS, which comes with Supervisor and add-on store, making it easier to add things like Node-RED, the File Editor, etc. HA will manage things like smart lights, switches, motion sensors, and will integrate with our security cameras via the next item (Frigate). We'll use Ansible to set up Home Assistant (though HA OS might be installed manually; if so, we document steps). Once up, we will configure integrations such as Philips Hue lights, smart plugs, perhaps thermostats or any existing IoT. This allows automations like turning on lights on motion or scheduling appliances – aligning with the user's interest in "email automations" and "natural language dates," we could integrate our AI such that sending an email or message like "remind me to water plants next Monday" gets parsed and added to an HA automation or calendar (this bridges into the AI section).

- **Frigate NVR (AI Camera Security)**: For camera surveillance with AI capabilities, we'll deploy **Frigate NVR**. Frigate is an open-source NVR that performs real-time object detection on camera feeds using local AI models [12]. Its key features include detecting people, cars, animals, etc., and it can be configured to recognize specific objects or even faces (with companion tools). We will attach some IP cameras (or even USB webcams) to the system. Frigate will ingest their RTSP streams. With a Coral AI accelerator (USB or PCIe) or using the CPU/GPU, Frigate can run a model (TensorFlow or ONNX) to identify objects of interest in each frame [12]. This drastically reduces false alarms compared to simple motion detection because Frigate can tell *what* caused the motion (e.g., a person vs. a tree's shadow) [13]. We'll configure zones and alerts – e.g., notify us (via Home Assistant or directly) if a person is detected in the backyard at night, etc. Frigate integrates very well with Home Assistant (it can send MQTT messages for events and has a HA integration for viewing cameras and clips). We will aim to set up **double-take** and **CompreFace** as well, which are add-ons to enable facial recognition in concert with Frigate (so we could get alerts like "John Doe is at the front door" if we train it on family faces). This satisfies the "cameras with right facial recognition" requirement. All processing is local, addressing privacy and avoiding cloud fees, which is a major advantage Frigate has over cloud camera services [12].

We plan to document tuning Frigate (which can be CPU intensive). The deployment could be either a VM with Docker (Frigate runs as a Docker container typically) or an LXC with Docker. If we have a Coral TPU, we'll passthrough the USB to that VM. Our Ansible script will set up the container with the proper docker-compose (including volumes for config and recordings). We'll allocate a fast disk or use the NAS for storing video clips.

- **Other Security/Automation**: We might also deploy **Zigbee2MQTT** or **Z-Wave JS** if we have Zigbee/Z-Wave devices, using a USB radio. This would integrate with Home Assistant for door sensors, motion sensors, etc., which the user's narrative about door notifications and motion-triggered lights (like from the Medium article scenario) resonates with [14] [15]. For instance, if someone approaches after dark, we could emulate the described automation: cameras detect person -> notify HA -> HA turns on pathway lights gradually and sends a notification [16]. These kinds of advanced automations become possible with the combined power of Frigate (detection) and HA (orchestration). We'll script some as demos (e.g., flashing lights when a known face is recognized vs unknown).

- **Internal Communication**: In a smart home context, setting up something like **Node-RED** (which is available as a Home Assistant add-on) could help create more complex logic or integrate external triggers (like an email or calendar event triggers a device action). Also, we could integrate **voice assistants**: for example, Home Assistant can connect to local voice control (Rhasspy) or Alexa/ Google (if we allow cloud) to enable voice commands for our services ("Hey Google, ask Home Assistant to start the movie mode" which could dim lights, turn on Jellyfin on TV, etc.). Implementing a "Movie mode" or "Party mode" as mentioned in the narrative [17] would be a fun way to tie together media and automation – e.g., when we start playing a movie on Jellyfin, HA could automatically set the scene (lights off, thermostat adjusted, notifications muted). We'll explore these as creative demonstrations of our integrated system.

In summary, this home automation and security segment of the lab will show how we can use open-source tools to achieve a smart home that rivals commercial solutions, without sacrificing privacy. The combination of Home Assistant + Frigate is particularly powerful: **local AI-powered surveillance** ensures we only get meaningful alerts, and **automations** can be as simple or complex as we desire. We'll ensure to document some example automations (with YAML or Node-RED flows) to highlight the capabilities.

## 4. Productivity & Business Tools

A key differentiator for our project is focusing on services that might appeal to business use-cases, thereby creating a portfolio of solutions we could offer professionally. We will deploy a few such applications to experiment with and demonstrate:

- **Project Management**: Deploy an open-source project management or collaboration platform. A strong candidate is **OpenProject** (an open-source alternative to Jira/Asana) which provides project tracking, Gantt charts, issues, etc. Another option is **Wekan** (an open-source Trello-like kanban board) or **Taiga**. We can choose OpenProject for its comprehensive features. It's a Ruby on Rails app, so we'd run it in a VM or container and use Ansible to configure it. This can be used to track tasks for our homelab project itself (dogfooding the tool). It shows potential clients/employers that we know how to implement and use PM tools.

- **Knowledge Management**: We plan to integrate closely with our **Obsidian Vault**, but also might deploy a wiki for structured documentation. **Wiki.js** or **BookStack** are both modern wiki platforms. BookStack is PHP/Laravel and user-friendly for creating a documentation site (with books/chapters). We can use it to publish the documentation of our project in a polished form (in addition to blog posts). Another tool is **Outline** which provides a wiki with an easy interface. This would simulate an internal documentation site as used in companies. We'll secure it and possibly integrate it with any SSO we set up. If the Obsidian vault is mostly for personal use and local AI, an internal wiki might be a better format to share knowledge publicly (the blog/portfolio can be separate as well).

- **CRM / ERP**: To further target business use-cases, consider deploying a CRM (Customer Relationship Manager) like **SuiteCRM** (open-source fork of SugarCRM) or **EspoCRM**. This would allow us to showcase, for example, how a small business could manage leads and clients with open-source tools. Similarly, an ERP like **ERPNext** (which covers accounting, inventory, etc.) or a simpler invoicing system could be installed. These are heavier applications, so perhaps just one of them to start. The user's mention of "the financial one" might have been referencing an open-source finance or ERP tool – ERPNext or **Firefly III** (an open-source personal finance manager) come to mind. Firefly III is specifically for tracking personal finances (budgets, expenses) – which is more personal than business. However, it could be included as a personal finance tool in the suite, which also demonstrates the capability to self-host something for budgeting.

- **Communications & Collaboration**: We can deploy internal communication tools such as **Mattermost** or **Rocket.Chat** (self-hosted Slack alternatives) to demonstrate team collaboration setups. Even if it's just for ourselves, running a Mattermost server and integrating it with our other services (like receiving alerts from Home Assistant or Frigate into a Mattermost channel) can simulate an office environment. However, since we already are keen on Discord integration (for media and AI), we might not need an internal chat – we could stick with using our Discord (or a private server) as the collaboration hub.

- **Email and Automation**: Instead of hosting our own mail server (complex and not very rewarding for a homelab due to spam issues), we'll focus on *email automation* using existing email accounts. For example, we can create an automation where sending an email to a certain address (or even a Telegram/Discord message) triggers actions: e.g., "email `todo@homelab` with subject 'Pay bills next Friday' -> gets parsed by AI and added to a task list or calendar." To do this, we might use **n8n** (a workflow automation tool, similar to Zapier but self-hosted). n8n can watch an IMAP inbox or a webhook, then run a workflow. We can incorporate AI in that workflow for natural language understanding. For instance, **date parsing**: the email body says "next Friday" – we use a small NLP script or AI service to turn that into an actual date on the calendar. **Natural language processing of tasks and data** can be achieved using either local LLMs or cloud ones if needed. We will set up **n8n** (or Node-RED) in our homelab to handle such automation scenarios. As a concrete idea, an n8n workflow might take a plain English request (via email or chat), feed it to an LLM prompt that extracts intent and details, then perform actions like creating a calendar event, or updating a project board, etc. This kind of integration shows how the homelab can not only host services but also connect them in intelligent ways, which is valuable for businesses looking to save time.

- **Data Analysis and AI Tools**: We will deploy tools to extract and analyze information, aligning with the idea of *"extracting wisdom out of YouTube videos and infographics."* One approach is to use our automation tool (n8n) or custom scripts to:

- **Transcribe and Summarize Videos**: Utilize a speech-to-text engine (like OpenAI's Whisper model or DeepSpeech) to transcribe YouTube videos or other media. Then feed the transcript to an LLM summarizer. In fact, there are templates and community examples for exactly this: e.g., an n8n workflow that takes a YouTube URL and returns a structured summary using GPT analysis [18]. We can implement a similar pipeline locally: download video or audio, run a transcription model, then summarize via our local AI or an API, and store the summary (perhaps in Obsidian or send to us via email/Discord). This will allow us to quickly glean insights from long videos without manually watching everything.

- **Infographic/Text Image Analysis**: For infographics, the challenge is to read text and interpret visuals. We can use OCR (Optical Character Recognition) like Tesseract to extract any text in an image. For the visual elements or any charts, a fully automated "insight extractor" is non-trivial, but we could attempt to use an image captioning model (like BLIP or CLIP interrogator) to describe the image content, then have an LLM interpret that. Another idea: if the infographic is essentially conveying data (like a chart or diagram with labels), we might rely on the fact that many infographics come with an accompanying article or description. For our purposes, we will demonstrate at least OCR + summarization for images containing text. This extracted "wisdom" can be appended to our knowledge base for future query.

- **Local AI & Knowledge Base**: This deserves its own focus, as it ties many pieces together:

- We will deploy a **local LLM** (Large Language Model) to avoid reliance on external AI services for most tasks. With the rapid development of models like Llama 2, it's feasible to run a fairly capable model on high-end consumer hardware. Our homelab server might run a smaller GGML model (for example, a 13B-parameter model quantized) to handle queries. Tools like **Ollama** or **text-generation-webui** or **OpenWebAI** can host the model and provide an API. We will set up one of these, possibly containerized. This local AI will be used for things like summarizing text, answering questions, and even engaging in chat-based commands (like an assistant).
- **Obsidian Vault Integration**: We have an Obsidian vault where we keep notes and documentation. We want our AI to be able to use that as a knowledge source. The plan is to implement a **Retrieval-Augmented Generation (RAG)** system for the vault. Concretely, we can use a pipeline where:
    1. We regularly (or on-demand) index the markdown files in the Obsidian vault. This means converting the vault into a vector database: each note is split into chunks, embeddings are generated for each chunk, and stored (perhaps in a local **FAISS** index or a light SQLite-based vector store) [19] .
    2. When the AI assistant is asked a question, we use similarity search to retrieve relevant note chunks from the vault, and feed those (as context) into the LLM prompt [20] . The LLM then can craft an answer using the actual content from our notes, rather than just its trained knowledge. This ensures the AI's answers about our personal knowledge are accurate and specific, essentially creating a personalized **"second brain" chat assistant**. There are existing tools (as per Obsidian forums) such as *Obsidian Copilot* that implement RAG on vaults [21] . We might build our own using LangChain or simply using Flowise (a no-code tool) as described in a guide [22] [19] . In one example, a user set up a Flowise workflow to do exactly this: fetch markdown files from the vault, split and embed them, then use a conversational QA chain with a local model [19] [20] . We will aim to replicate that either with Flowise or custom code.
    3. The outcome: we can ask (via a chat interface or voice) questions like "What steps did I follow to install Nextcloud?" or "Summarize what I learned about Traefik vs Nginx Proxy Manager" and the AI will pull from our actual documented notes to answer. This is immensely useful for quickly recalling information without searching manually.
- **Assistant Interfaces**: We will expose the AI assistant through convenient means. One idea is a **Discord bot** (since Discord was mentioned) – we can run a bot that, when you DM it or mention it in a channel, it will take your query, run it through the local LLM (with RAG if needed), and reply. This way, from a phone or anywhere, we could interface with our homelab AI. Another interface could be a simple web app or Telegram bot. For voice, we could integrate with Home Assistant (HA has a conversation/intents system) or run something like **Jarvis** on a tablet. But starting with text-based interaction is easier.

- **Agentic Automation**: Pushing further, we want the AI to not just answer questions, but also perform actions ("agentically"). For instance, you might tell the AI in Discord, "Spin up an instance of OpenProject for me to test." If our orchestrator AI is advanced enough, it could interpret that, then call the necessary automation (Terraform/Ansible) to deploy a new VM and set up OpenProject, then report back when done. This is exactly the realm of AI agents controlling infrastructure, which is cutting-edge but within reach using tools like LangChain's agents or custom scripting with the APIs of our tools. We will attempt a simple scenario, like an AI agent that can create a notes file in Obsidian, or trigger an Ansible playbook on command. This ties into the earlier concept of the blueprint/orchestrator AI that might assist in building the environment itself.

- **DevOps/CICD Tools** (Optional): If time permits, we can incorporate some CI/CD or dev-oriented tools such as **GitLab CE** or **Jenkins** (for practicing automated builds), or **Portainer** to manage Docker containers easily. While not requested, these could be bonus additions to demonstrate a full-fledged lab capability.

All these productivity and business-oriented deployments will be documented as if we were delivering a solution to a client. For example, "deploying OpenProject on Ubuntu using Ansible – how to configure email notifications, etc." or "setting up SuiteCRM and integrating it with Mattermost for notifications." By doing this, we reinforce our understanding and also create content for our blog that might attract others interested in self-hosted business apps.

## 5. Future Enhancements (Beyond MVP)

*(This section outlines possible extensions once the core system is up, many of which have been touched on above. They ensure the system remains forward-looking and continuously improving.)*

- **High Availability**: Our current plan assumes a single Proxmox server. In the future, to practice enterprise concepts, we could add a second Proxmox node and set up a cluster, and possibly use Ceph or GlusterFS for distributed storage. This would allow live migration of VMs and higher uptime. It's likely overkill for a home lab with one user, but as a learning exercise it's invaluable (and could be something to blog about). Similarly, exploring **Kubernetes** by deploying k3s or microk8s on top of our VMs could be a next step, containerizing some of our apps in a cluster way.

- **Security Enhancements**: Implement **Zero Trust Network Access** for remote connections (like using Cloudflare Access or Tailscale for VPN). Also, penetration-test our setup (run OpenVAS or use security tools) and then harden it (close unused ports, enforce HTTPS/HSTS, etc.). This could become a write-up on homelab security best practices.

- **Energy Efficiency & Scheduling**: Integrate with smart plugs and UPS to manage the homelab's power usage. We could schedule non-critical VMs to shut down at night or when not in use to save energy (maybe orchestrate that via Home Assistant automations, given the mention of power management in the narrative [23]). Also, monitor the server's power consumption and include that in our Grafana metrics.

- **Offer Services to Friends/Family**: As a practical test, onboard a couple of external users (friends/family) to use some of the services (maybe give them accounts on our Jellyfin, Nextcloud, or a guest dashboard for cameras when they visit). This provides feedback on usability and also is an opportunity to practice user support and access control.

All these potential enhancements keep the project dynamic and ensure we continue learning and showcasing new capabilities.

## Implementation Plan and Task Breakdown

To achieve the above, we'll break the work into stages and tasks. Many tasks can proceed in parallel, which is ideal for using an AI orchestrator or simply for organizing work with multiple threads.

**Phase 1: Planning and Environment Setup**
1. **Define Requirements & Design (Completed)** – Compile the objectives and architecture (essentially what this document contains). Ensure we have an inventory of hardware (Proxmox host specs, storage available, network setup) and any necessary accounts (e.g., domain name, VPN provider for Gluetun, etc.). *Outcome:* a clear blueprint (this document) to follow.
2. **Set Up Base Proxmox Host** – Since Proxmox VE is already running, verify it's up-to-date and configured (Proxmox VE 8.x, updated kernel, etc.). Configure networking on Proxmox: create any necessary Linux Bridges (e.g., vmbr0 bridged to LAN). If VLANs will be used, set those up. Ensure Proxmox storage is configured (we may want a ZFS pool for VMs, etc.). Also create a Proxmox API Token or login for automation tools to use when provisioning VMs (Terraform or scripts will need access).
3. **Prepare Automation Tools Environment** – Set up a management VM or use the Proxmox host (if safe) for running Terraform, Ansible, and orchestration scripts. Possibly create an "infra" VM on Proxmox that has Terraform, Packer, and Ansible installed, and clone our Git repository. This VM can act as the control node where playbooks are run from. Alternatively, use a local workstation for this role. *Security:* store API keys (Proxmox token, etc.) in vaults or environment variables on this node.

**Phase 2: Infrastructure as Code Implementation**
4. **Packer Image Creation** – Develop a Packer template for our base OS (likely Ubuntu 22.04 cloud-init image). Packer steps: - Write Packer template JSON/HCL specifying Proxmox builder (with Proxmox API endpoint and credentials) [24] . - Provide an Ubuntu ISO path and cloud-init settings (user, SSH key, etc.). Use an autoinstall config to automate OS install [5] . - Have Packer run Ansible (or shell) to do minimal provisioning: create our admin user, install updates and maybe QEMU guest agent. - Packer then converts the VM to a template in Proxmox [6] .
This yields "Ubuntu-22.04-CloudInit-Template" ready for cloning. We'll test it by manually cloning a VM off it to ensure cloud-init works (SSH in with our key, etc.). *Parallelization:* This task can be done by one agent while another works on writing Terraform configurations (since the two will integrate). 5. **Terraform VM Definitions** – Write Terraform code for creating VMs/containers for each planned service: - Define Proxmox provider configuration (target our Proxmox host) [25] . - Write `proxmox_vm_qemu` resources for each VM (or create modules to reuse settings). For example, one resource for "vm_nextcloud" (using the Ubuntu template, set CPU/RAM/disk, cloud-init network config, etc.), another for "vm_media", "vm_homeassistant", etc. Similarly define any LXC containers if we go that route for certain apps. - Use `proxmox_lxc` resources for things like Pi-hole or simple containers if needed. - Include provisioning hooks if possible: Terraform can execute cloud-init user-data; we might embed a cloud-init snippet to run an Ansible bootstrap or script on first boot (or simply ensure the VM is created and then rely solely on Ansible after). - Validate by running `terraform plan` and then `apply` with perhaps one test VM first. - Store the Terraform state (probably locally, unless we set up a remote state).

*Note:* If Terraform proves tricky or unnecessary, an alternative is to write a custom script (Python with `proxmoxer` library) to clone VMs. But we'll try Terraform for the learning experience and to treat infra as code.
*Parallel tasks:* While Terraform config is written, others can start writing Ansible roles for the services (since those are independent of how VMs are created). 6. **Core Networking Services** – Build the networking core: - Set up a VM or container for **Pi-hole + Unbound**. This can be done via Ansible role or by using an official Pi-hole Docker. Possibly use a pre-made Ansible role for Pi-hole. Ensure this VM has a static IP on LAN. After deployment, test that Pi-hole is resolving DNS (with Unbound upstream) [9] and configure DHCP (or set router to hand out Pi-hole's IP as DNS to clients). - Set up the **Reverse Proxy**. Decide NPM vs Traefik. Initially, we may deploy Nginx Proxy Manager via Docker (as it's quick to stand up with a one-compose-file). Use Ansible to deploy NPM container on a small VM (even could co-host with Pi-hole on a "utilities" VM). Alternatively, deploy Traefik with a static config file and wildcard certificates. We'll likely need a domain name; for lab, a free DNS or just local hostnames might suffice. This step is crucial to test early – e.g., set up a test website container and verify we can access it via the proxy using a domain name with valid SSL.
(These networking tasks can be done in parallel with VM creation tasks; they are mostly independent, aside from needing some infrastructure ready.) - **Internal DNS entries**: Once Pi-hole is up, add DNS records for all services (e.g., `nextcloud.home.lan` -> IP). If using real domain, configure external DNS too (maybe via Cloudflare API and Let's Encrypt DNS challenge if we can't open port 80). Ensure reverse proxy uses those hostnames.

**Phase 3: Service Deployment & Configuration**
7. **Ansible Roles for Services** – For each major service or category, create an Ansible role or playbook. These can be developed concurrently by different sub-agents or team members since they don't depend on each other. For example: - Role: `nextcloud` – tasks to install Docker & docker-compose (or LAMP stack if not containerizing), deploy Nextcloud, configure data volume (which might reside on NAS), and perhaps install an admin account. Possibly include installing Collabora CODE if we integrate that. - Role: `media_arr` – tasks to deploy Sonarr, Radarr, Lidarr, Jackett/Prowlarr, qBittorrent+Gluetun, etc. Likely done via docker-compose (there are many existing compose examples for a full media stack). The role will also drop in preconfigured config files (or use environment variables) for things like where to put downloads (the NAS mount path), and set up services to start on boot. - Role: `jellyfin` – install Jellyfin server (Docker or apt package). Ensure it has access to the media directory. - Role: `home_assistant` – if using Home Assistant OS VM, this might just be a step to import an OVA or run a script. Alternatively, use the Home Assistant Container (Docker) on a generic Linux VM. The supervised install script could be run via Ansible. We'll also deploy Zigbee2MQTT if needed, possibly as an HA add-on. - Role: `frigate` – deploy Frigate (Docker) and configure it to use our camera streams. Ensure Coral is passed if available. This role will manage the `config.yml` for Frigate (detectors, camera zones). - Role: `openproject` – install prerequisites (Postgres, etc.), then OpenProject (they have a Docker or an installer). Configure the domain for it. - Role: `wiki` – e.g., BookStack: deploy MariaDB, the PHP app, etc. - Role: `mattermost` (if doing chat) or skip if not needed. - Role: `n8n` – use Docker to run n8n. Then import or configure some of our workflows (maybe via API or just document manual creation of flows if not easily automated). - Role: `llm` – set up the local LLM service. This could involve downloading model files (many GB, might not put in Ansible due to size). Perhaps just pulling a Docker image for an API like text-generation-webui or installing `ollama` CLI and an open model. We will also prepare our RAG setup here: e.g., ensure `faiss` or another vector DB is installed, and perhaps schedule a job to re-index Obsidian vault daily. - Role: `obsidian_index` – (if not included above) to run a script that reads vault files and builds the index. Maybe use Python with LangChain for this. This might be executed on the LLM server or another util container.

Each role will be written and tested one by one, but since multiple services can be worked on in parallel, this is a prime area for parallel execution. For example, one person/agent can tackle media stack while another does Nextcloud, etc. We should use a consistent approach (where possible, use Docker for apps unless there's a reason to install directly) to simplify environment management.

1. **Deploy Services** – Once roles are ready, we run the playbooks to actually deploy onto the VMs:
2. Possibly use Terraform to bring up all the VMs in one go (`terraform apply` to create all defined VMs).
3. Then run Ansible site playbook which includes all roles, targeting the respective hosts. This will set up all services in one execution (or we do them step by step and verify each).
4. As each service comes up, configure the reverse proxy to expose it (Ansible can add Nginx Proxy Manager hosts via its API or for Traefik just the labels are set via docker, so it auto-registers).
5. After deployment, perform checks: e.g., access Nextcloud web UI and complete its web installer if needed; open Jellyfin and add media library; check Sonarr can connect to qBittorrent; ensure Home Assistant UI is up; etc. This is the verification stage.

We expect some iterations here to fix issues. But ultimately, we should reach a state where all targeted services are running properly.

**Phase 4: Integration & Testing**
9. **Integrations and Workflow Testing** – Now that individual services work, we connect them and test cross-service functionalities: - Mount the NAS shares on the relevant VMs (Nextcloud external storage, media download path, etc.) and test file access. - In Home Assistant, add integrations for Frigate (receiving camera events), and for media (we can add Plex integration or at least notifications from *arrs via webhook). Also possibly add Nextcloud (WebDAV) integration or feeds from other apps if relevant. - Set up the Discord/ communication bots: e.g., configure Notifiarr with our* arr apps and a Discord webhook, test that a movie download sends a Discord message. Similarly, test any command functionality we implemented. - Test the n8n workflows: e.g., input a YouTube link into the system (maybe via an HTTP request or UI) and see if it outputs a summary. Or send a test email with a natural language task and verify it gets processed. - Exercise the Obsidian QA: ask the assistant questions that we know are answered in our notes, see if it retrieves correctly. Tune the prompts if needed. - Perform user acceptance tests: simulate normal usage — stream a video from Jellyfin, upload a file to Nextcloud and share it, trigger an automation by walking past a camera, etc. This will reveal any performance issues (like do we need more CPU for Jellyfin transcoding? Does Frigate overwhelm the server? etc.).

1. **Documentation & Knowledge Base** – As we go, we will have been documenting, but here we finalize documentation:

    - Write a **blog series** or a comprehensive article for each major part (e.g., “Homelab Part 1: Infrastructure as Code with Proxmox and Terraform”, “Part 2: Media Server Setup”, “Part 3: Smart Home Integration”, etc.), complete with code snippets and learnings. Use our wiki or blog platform to publish these. We will cite external references we used, and include screenshots (like Proxmox GUI, or sample dashboards).
    - Update our Obsidian vault with any changes, and perhaps publish a sanitized version of the vault or at least the relevant sections to the public wiki.
    - Create diagrams of the architecture to include in docs (showing how everything is connected).
    - Possibly record a short video tour of the system for the portfolio.

2. **Optimization & Hardening** – Before considering the project “done,” ensure to clean up any rough edges:
    - Enable auto-updates or at least update notifications for the apps (maybe set up Watchtower for Docker containers to update images regularly).
    - Harden security: e.g., enforce HTTPS-only, set up fail2ban on services that allow login (Nextcloud, etc.), ensure default passwords changed, etc.
    - Optimize resource usage: pin CPU/RAM for some VMs if needed, adjust Frigate’s detection frame rate to reduce CPU if necessary, etc.
    - Implement monitoring: configure Uptime Kuma to ping each service URL and alert if down, as a basic check that our automation hasn’t broken anything over time.
3. **Demonstration & Next Steps** – With everything running, prepare a demonstration (for ourselves or potential audience). This could be a live demo of using the AI assistant to control the lab, or a narrative (in our blog) describing a day in the life of using the homelab for various tasks. Gather any remaining ideas (like those future enhancements with Kubernetes or additional services) and outline them for a Phase 2 project, ensuring the homelab has continuous growth.

Throughout these phases, using an AI agent orchestrator could assist by generating boilerplate code or even running some tasks concurrently. For instance, an AI writing an Ansible role while another debugs a Terraform script. We will leverage such tools carefully, reviewing all AI-generated code for correctness and security.

Finally, we consider the project a success when the homelab can be largely controlled through our “single pane of glass” – whether that’s Home Assistant for home-related tasks, or our AI assistant for knowledge tasks – and when adding a new service is as straightforward as writing a small config and letting the automation handle it. We’ll have a system that is **flexible, documented, and powerful**, ready to impress both the homelab community and potential business clients who might benefit from similar setups.

## Conclusion

By following this plan, we will build a multifaceted homelab that showcases modern infrastructure automation and a wide range of self-hosted services. We will have replaced numerous cloud services with open-source equivalents under our control, implemented advanced features like AI-driven automation and surveillance, and created a rich knowledge base of our experiences. Not only does this environment serve as a personal playground for skill development, but it also becomes a demonstrable product – a “reference architecture” for a small-scale, smart IT environment. The combination of **Terraform + Packer + Ansible** gives us a reproducible infrastructure [1] [24], the inclusion of community scripts and Awesome Self-Hosted apps ensures we’re using well-regarded solutions [2], and the integration of **AI assistants** positions the project at the cutting edge of homelab capabilities.

We are confident that implementing this will greatly enhance our skills in DevOps, networking, and system design. Moreover, by documenting and sharing the journey, we add value to the community and to our professional portfolio. This homelab will continuously evolve, but the prompt and plan laid out here form the solid foundation for that evolution – and perhaps, with the help of an AI orchestrator, the journey to build it will be accelerated and innovative.

---

[1] [3] [4] [7] [10] [25] Tux Machines — Self-Hosted NAS, Proxmox, Homelab, and Photos Server
http://news.tuxmachines.org/n/2025/10/26/Self_Hosted_NAS_Proxmox_Homelab_and_Photos_Server.shtml

[2] Proxmox VE Helper-Scripts
https://community-scripts.github.io/ProxmoxVE/

[5] [6] [24] Create Cloud-Init VM Templates with Packer on Proxmox | The Uncommon Engineer
https://ronamosa.io/docs/engineer/LAB/proxmox-packer-vm/

[8] I Replaced Nginx Proxy Manager with Traefik in My Home Lab and It Changed Everything - Virtualization Howto
https://www.virtualizationhowto.com/2025/09/i-replaced-nginx-proxy-manager-with-traefik-in-my-home-lab-and-it-changed-everything/

[9] Pi-hole with Unbound DNS: Complete Docker Setup for Privacy & Ad-Blocking | The Uncommon Engineer
https://ronamosa.io/docs/engineer/LAB/pihole-docker-unbound/

[11] [14] [15] [16] [17] [23] From Tech Hoarder to Home Lab Hero: A Geek's Guide to Digital Independence | by Bryan Cruse | Medium
https://medium.com/@UnfilteredInk/from-tech-hoarder-to-home-lab-hero-a-geeks-guide-to-digital-independence-1aa409b3148e

[12] [13] Frigate NVR: Revolutionizing AI-Powered Video Surveillance for Smart Homes and Businesses
https://skywork.ai/skypage/en/Frigate%20NVR%3A%20Revolutionizing%20AI-Powered%20Video%20Surveillance%20for%20Smart%20Homes%20and%20Businesses/1972898992821563392

[18] ⚡AI-Powered YouTube Video Summarization & Analysis | n8n workflow template
https://n8n.io/workflows/2679-ai-powered-youtube-video-summarization-and-analysis/

[19] [20] [22] Talking to your second brain: Build a Flowise RAG to chat with your Obsidian Vault | by Mart Kempenaar | Medium
https://medium.com/@martk/talking-to-your-second-brain-build-a-flowise-rag-to-chat-with-your-obsidian-vault-00645106e73b

[21] Vault QA (Basic) - Copilot for Obsidian
https://www.obsidiancopilot.com/docs/vault-qa