# Table of Contents

# 1. Introduction

This report aims to provide an overview of the author's efforts in deploying a Nomad orchestration cluster. The project's objective was to construct a virtualized lab environment for a single-zone container orchestration cluster using HashiCorp solutions. The author opted to using infrastructure as code approach at the project's core: infrastructure and configurations should be defined and managed using code and be reproducible. For the orchestration cluster using HashiCorp Nomad, the author's approach diverged from the all-batteries-included industry standard approach of using Kubernetes emphasizing the exploration of containerized services to acquire expertise in designing, building, and managing contemporary dynamic infrastructure.

Following chapters will give an account of the motivations for building such a (seemingly overly) complex environment, the process of building it, key learning outcomes and proposed improvements.

# 2. Motivations for Orchestration Environments

Orchestrators streamline the deployment and reverting processes of containerized services across multiple hosts within the orchestrated cluster. Additionally, orchestration ensures the decoupling of containerized services from individual hosts, mitigating dependence on any specific host, whether virtual or physical, thereby offering resilience against physical host failures. In the event of a host failure, the orchestrator seamlessly reallocates services to alternative hosts, ensuring continuous service availability.

Hence, an orchestrated environment appeared appealing for its advantages:

- Resilience against host failures.
- Automated service deployments.
- Automated load balancing.
- Automated routing and certificate management for each service.
- Simplified container deployment compared to the manual configuration management process for individual services.

Before embarking on this project, the author possessed administrative experience with virtual machines and containers, alongside limited exposure to orchestration via Kubernetes. The project's motivation stemmed from a desire to assess orchestration's practicality within a home laboratory setting. The author initially speculated that the endeavor might not yield substantial benefits, given the perceived complexities and resource demands associated with orchestrating a small number of services at home. Nevertheless, regardless of the outcome, the gained insights would prove valuable for comprehending modern infrastructure paradigms and guiding future initiatives.

## 3. Building the Lab

The project utilized personal computing equipment already available in the author's home laboratory setup. This setup comprised a single 36-unit rack equipped with switches and routers supporting 1GB/s networking across all ports, alongside a modem connection to the internet with a duplex speed of 250mbit/s. To ensure network isolation from the home network, a secondary router implemented double NAT. Hardware employed in the project included two routers, two switches, an HP DL380 G9 server, and two Raspberry Pi 3B+ devices connected to the secondary laboratory router via a switch. A visual representation of the physical network connections is depicted in Figure 1.

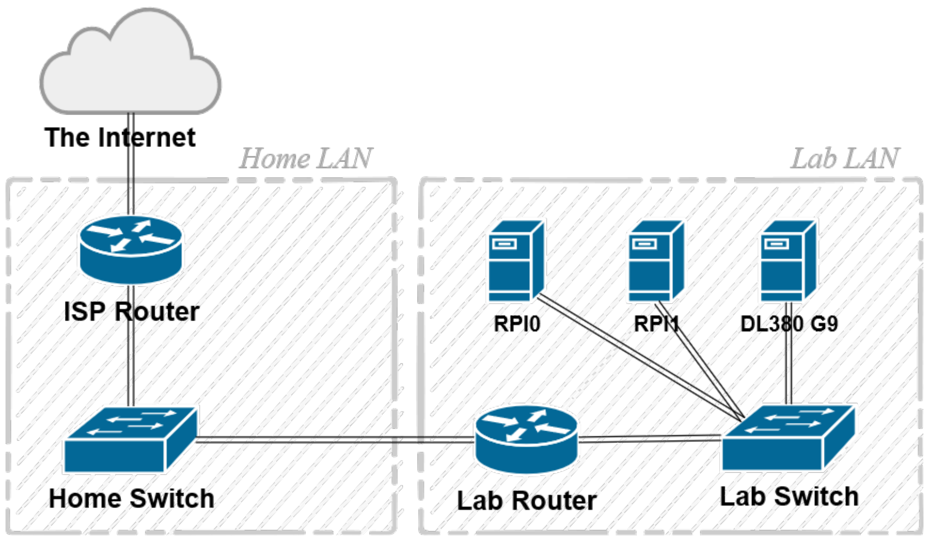


Figure 1. Physical network connections map

The cost-effective approach of utilizing existing hardware, originally not intended for orchestration, posed a notable limitation by necessitating the simulation of a multi-host environment through virtualization due to the presence of only one physical machine capable of running orchestrated services at scale. This reliance on virtualization influenced the realism of the experimental setup, potentially overlooking or simplifying certain nuances and intricacies that might have been observed in a dedicated hardware environment. Additionally, the proposed resilience against host failures in this simulated environment only extends to virtual machines, as individual failures are improbable; however, if the physical host were to encounter issues, the entire cluster would fail. Therefore, while simulating a multi-host cluster using virtualization can serve as a viable testing method, it cannot be deemed a production-ready solution for service provisioning.

Subsequent chapters delve into the hardware setup, operating system installation, and configuration management of the virtualization machine utilized for orchestration.

## 3.1 Hardware

To facilitate Nomad orchestration, a single HP DL380 G9 server was utilized, with the configuration of CPUs and storage proving relatively straightforward, focusing on maximizing the board's capacity. Despite the potential to utilize all 24 slots for a total memory capacity of 384GB, careful consideration was given to RAM allocation to optimize performance. While it may be tempting to fully populate all memory channels, it was found that doing so would significantly lower memory frequency due to the memory controller's behavior, as elucidated in a blog post analyzing the NUMA architecture [1]. Referring to the DL380 G9 user manual [2], it was noted that installing 24 modules of 16GB RAM would result in a reduction of memory speed from the specified 2133MHz to 1600MHz. To maintain optimal performance, a configuration of 16 modules of 16GB RAM was chosen, leaving the third slots of the memory channels unoccupied. This decision was based on the assessment that 256GB would likely suffice for the intended purposes, considering the uncertainty surrounding the final number of virtual machines to be deployed.

For the two Raspberry Pis, additional heatsinks were installed to enhance cooling performance, and a Power over Ethernet (PoE) Hardware Attached on Top (HAT) module was added to facilitate both power and networking over a single Ethernet cable, simplifying the cabling setup. No further configuration was undertaken for the Raspberry Pis.

### 3.1.1 BIOS

After configuring the hardware, the next step involved initiating the system setup and adjusting BIOS configurations. Upon system boot, the detection of a faulty RAM stick necessitated its immediate replacement. Virtualization was enabled, and power-saving and throttling settings were disabled to ensure maximal performance. Additionally, functionality of the Integrated Lights-Out (ILO) [3] hardware-based remote management platform was confirmed.

Subsequently, the HP Smart Storage Administrator [2] interface was utilized to configure the eight 600GB physical disks into a single logical volume employing RAID10 via the machine's installed storage controller. This setup facilitated seamless disk replacement without requiring system reboots in case of failure. Given the limited size of the disks, the decision was made to forego standby status for additional unconfigured disks, a feature typically utilized for disaster recovery scenarios.

### 3.1.2 Operating System: Proxmox Virtual Environment

For virtualization, the Proxmox Virtual Environment [4] server management platform was selected due to its free and open-source nature. Leveraging industry-standard solutions such as Debian [5] at its core, alongside standard Linux and OVS [6] networking, popular storage options, and the widely-used KVM hypervisor [7], aligned well with the author's prior experience and ensured readily available support and solutions. The nonrestrictive nature of the free version and its single-install approach made it a suitable and advantageous choice for the project's requirements, especially when compared to other virtualization platforms that may have necessitated investment in licenses or posed greater setup complexities.

### 3.1.3 Operating System: Raspberry Pi OS

Once again, a Debian-based operating system was selected for the Raspberry Pis, with Raspberry Pi OS [8] installed on both devices, designated as 'rpi0' and 'rpi1'.

To streamline management, Dnsmasq [9] was installed and configured on 'rpi0' to facilitate internal domain name resolution for all physical and virtual hosts.

## 3.2 Virtualization

In line with the infrastructure as code approach established by the author, HashiCorp Packer [10] was employed to generate operating system images for virtual machine clone templates, while HashiCorp Terraform [11] was utilized to provision virtual machines from these templates on the Proxmox platform.

### 3.2.1 Operating System Image Builds Using Packer

The code utilized for constructing the operating system image clone templates is available on GitHub [12].

For consistency in managing hosts within the lab, Ubuntu 22 [13], a Debian-based Linux operating system, was selected as the standard operating system for all virtual machines, aligning with the Debian-based systems running on both Proxmox and Raspberry Pis.

Using Packer, the following steps were automated and rendered reproducible as code:

- Downloading the latest Ubuntu 22 image.
- Configuring Cloud-init [14], which serves as the cornerstone of automation. Cloud-init enables customization of the machine upon provisioning and each boot, essential for automating the provisioning process.
- Applying upgrades and cleaning unnecessary files to conserve disk space, with free disk space being zeroed to minimize the image size.
- Locking the root account and prohibiting SSH root login.
- Converting the configured machine into a Proxmox clone template, subsequently used for creating the machines constituting the project's infrastructure.

### 3.2.2 Virtual Machine Provisioning Using Terraform

The code utilized for provisioning virtual machines from image clone templates can be accessed on GitHub [15].

Leveraging the template constructed with Packer, Terraform was employed to orchestrate the provisioning of virtual machines on the Proxmox platform and configure them using cloud-init. Within the Terraform definition file, configurations specifying virtualized hardware, hostname, IP, gateway, authorized SSH keys of the root user, and first boot commands were delineated.

Upon execution, the boot commands executed a script to expand each machine's disk to utilize the full capacity, accounting for the variance between the template's smaller disk size and the specified larger actual size required for running virtual machines.

During the application process, Terraform instantiated three base nodes comprising the control plane of the orchestration, three service nodes forming the service layer, and one support node catering to additional supporting services such as a certificate authority for internal certificates and Ansible for configuring all nodes. The support node was provisioned with 1GB of RAM, 2 CPUs, and a 50GB boot disk, while all base and service nodes were equipped with 12GB of RAM, 4 CPUs, and a 150GB boot disk each. These numbers of base and service nodes, along with their allocated resources, were determined based on the Nomad reference architecture for small clusters [16], which recommends a minimum of three machines for each layer to facilitate failover without manual intervention.

Following provisioning, the infrastructure comprised three physical machines—two Raspberry Pis (designated as rpi0 and rpi1) and one HP DL380 G9 (referred to as pve0)—with the Proxmox platform hosting seven provisioned virtual machines: three base nodes (vb0, vb1, vb2), three service nodes (vs0, vs1, vs2), and an additional head node. Figure 2 illustrates a simplified diagram depicting the physical and virtual nodes.

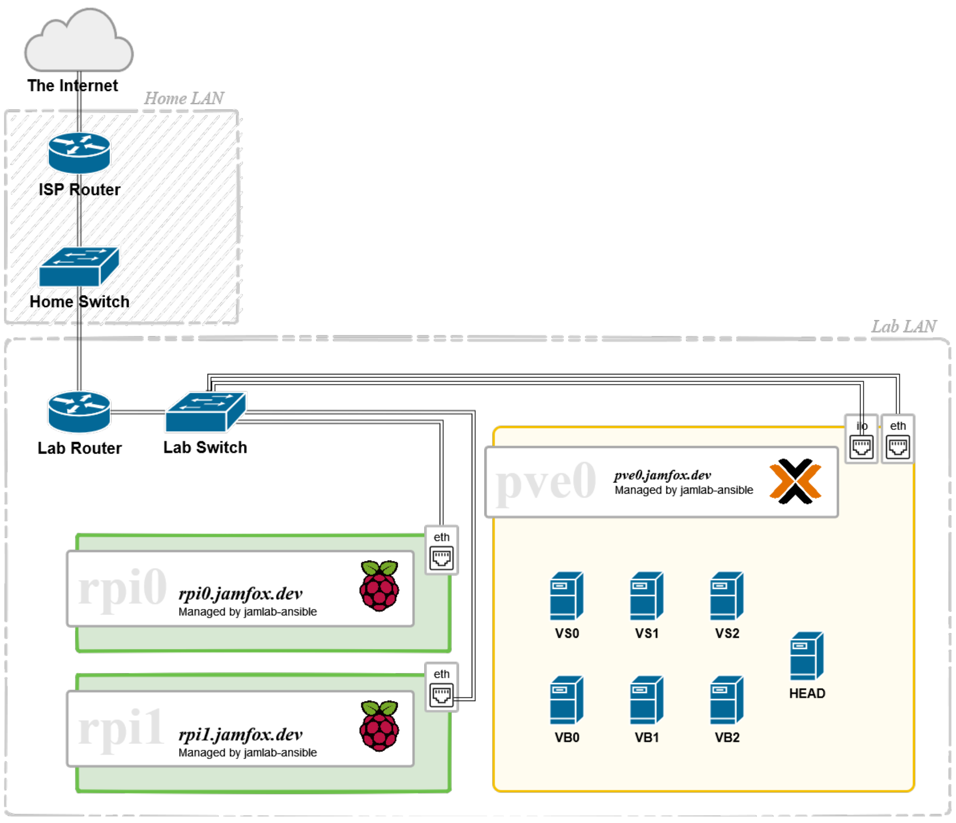


Figure 2. Simple overview of the physical and virtual machines

## 3.3 Configuration Management

Ansible [17] was utilized for the configuration of all physical and virtual hosts. It's important to note that all subsequent chapters assume that the described steps were documented using a reproducible infrastructure-as-code approach with Ansible, rather than manual configurations. The Ansible code employed for managing all machines can be accessed on GitHub [18].

The design of the Ansible code was intentionally kept simple: each host was assigned to only one host group, making it straightforward to discern the tasks being executed on each host at a glance.

Within the playbooks directory, each host group's playbook and variables file can be found. The playbook defines the connection parameters, additional variable files, and the list of roles to be applied to the hosts within the host group. Variable files then specify how each role should be executed on the respective host. Each role is separated to configure a specific service or aspect of a host. Ensuring idempotency of the roles was crucial in maintaining reproducibility and consistency of the configurations, guaranteeing that the resulting configuration on hosts remained consistent regardless of how many times the management system was run.

To streamline the execution of multiple playbooks, a bash script was developed. The script, available on GitHub [19], offered several advantages:

- Executes playbooks in parallel.
- Displays filtered logs containing only errors and changes for enhanced readability and actionable insights.
- Facilitates targeting specific playbooks using playbook names or string matches.
- Automatically fetches the latest version of specified git branches for execution.
- Allows for the specification of particular git branches for execution.
- Skips unreachable hosts to ensure seamless execution.
- Executes a post-exit script upon completion.

# 4. The Orchestration Environment

Out of the various orchestration solutions available in the industry [20], HashiCorp Nomad [21] was selected over the all-batteries-included industry standard approach of Kubernetes [22]. This decision was made with a focus on exploring containerized services and gaining expertise in designing and building orchestration environments from scratch, while also emphasizing the management aspect.

## 4.1 Nomad as Orchestrator

Nomad is centered around cluster management and scheduling, aligning with the Unix philosophy of maintaining a focused scope. It seamlessly integrates with tools like HashiCorp Consul [23] for service discovery and service mesh functionality.

In essence, HashiCorp describes Nomad as providing scheduling, self-healing, rollouts and rollbacks, storage, and auto-scaling capabilities. Consul complements Nomad by adding service discovery and service meshing features, addressing aspects that Nomad may lack. Notably, Nomad does not inherently include load balancing and configuration management functionalities, although they can be implemented and run on Nomad. As described earlier, the Ansible configuration management process detailed in Chapter 3 operated externally to the Nomad cluster, specifically from the virtual machine named 'head'.

### 4.1.1 Configuring Nomad

Consul was employed for service discovery and service meshing, seamlessly integrated alongside Nomad to form the core foundation of the orchestration framework.

The Ansible Community Nomad [24] and Consul [25] roles were utilized for the installation, configuration, and management of the orchestrator and its nodes.

A significant challenge in configuring Nomad and Consul via roles revolved around establishing precise configurations to enable encrypted communication both within the cluster and between Consul and Nomad.

Step CA [26] was implemented on the support node named 'head' as the certificate authority. This CA facilitated the issuance of internal certificates to all orchestration cluster nodes, ensuring encryption and verification of traffic among Nomad nodes.

Additionally, Podman [27] was adopted as the modern container engine instead of the default Docker engine [28]. This choice ensured that only a rootless user with finely-tuned permissions could operate the daemon and containers, enhancing security and isolating containerized services from the host operating system.

## 4.2 Load balancing

Load balancing involves routing and distributing incoming network traffic and workloads across multiple servers to enhance system performance, availability, and scalability. When employing orchestration tools like Nomad, load balancing becomes complex due to the dynamic nature of service deployment across multiple nodes.

To address this complexity, HashiCorp's recommended architecture [29] was followed, utilizing an external load balancer to route traffic to live orchestration nodes, each equipped with internal load balancers directing traffic within the cluster.

HAProxy [30] was installed and configured on an Raspberry Pi ('rpi1') to serve as the external load balancer, routing ingress Nomad traffic and checking the availability of Nomad nodes. HAProxy terminated HTTPS connections and forwarded HTTP requests to live Nomad node internal load balancers.

Wildcard certificates for lab domain names were configured on HAProxy using certbot [31] with Let's Encrypt [32] as the provider, ensuring secure communication with services exposed to the internet.

Considering the dynamic IP nature of the home environment, Cloudflare Tunnel [33] was utilized to connect the external balancer to Cloudflare's global network securely, enabling DNS routing through Cloudflare for enhanced security and fine-grained access control.

For internal load balancing, Fabio [34] was chosen due to its native integration with Consul service discovery. Nomad configurations were designed to be dynamic and self-healing, with Fabio deployed as a system job to ensure failover and even traffic distribution across all nodes.

A sequence diagram illustrating the communication flow from client to containerized service is depicted in Figure 3.

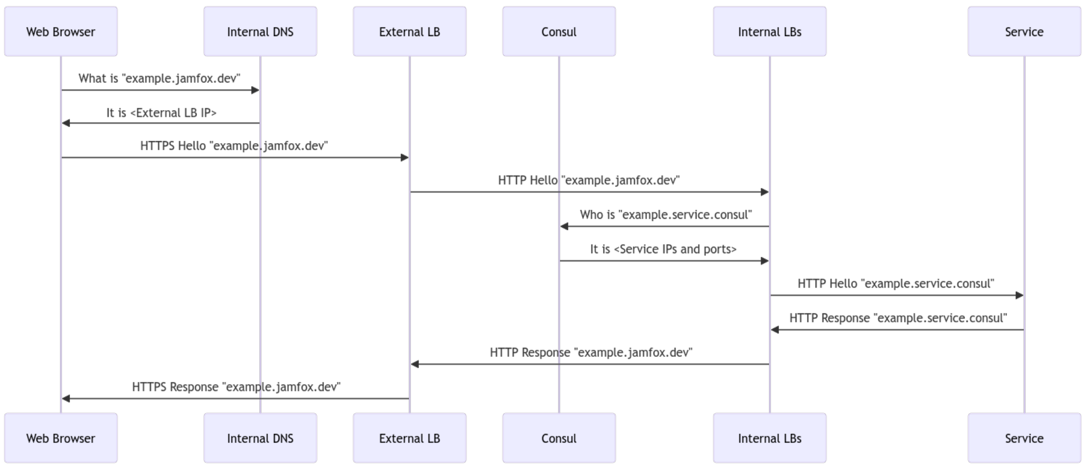

Web Browser
Internal DNS
External LB
Consul
Internal LBs
Service
What is "example.jamfox.dev"
It is <External LB IP>
HTTPS Hello "example.jamfox.dev"
HTTP Hello "example.jamfox.dev"
Who is "example.service.consul"
It is <Service IPs and ports>
HTTP Hello "example.service.consul"
HTTP Response "example.service.consul"
HTTP Response "example.jamfox.dev"
HTTPS Response "example.jamfox.dev"
Web Browser
Internal DNS
External LB
Consul
Internal LBs
Service

Figure 3. Sequence Diagram of Service Communication

# 5. Key Outcomes

## 5.1 Overview of the Final Architecture

In summary, the infrastructure comprised three physical machines:

- `rpi0`: Provided internal DNS resolution for the laboratory network.
- `rpi1`: Hosted the HAProxy external load balancer, handling client requests for containerized services.
- `pve0`: The Proxmox Virtual Environment machine where virtualized orchestration services were provisioned and operated from.

On `pve0`, there were seven virtual machines:

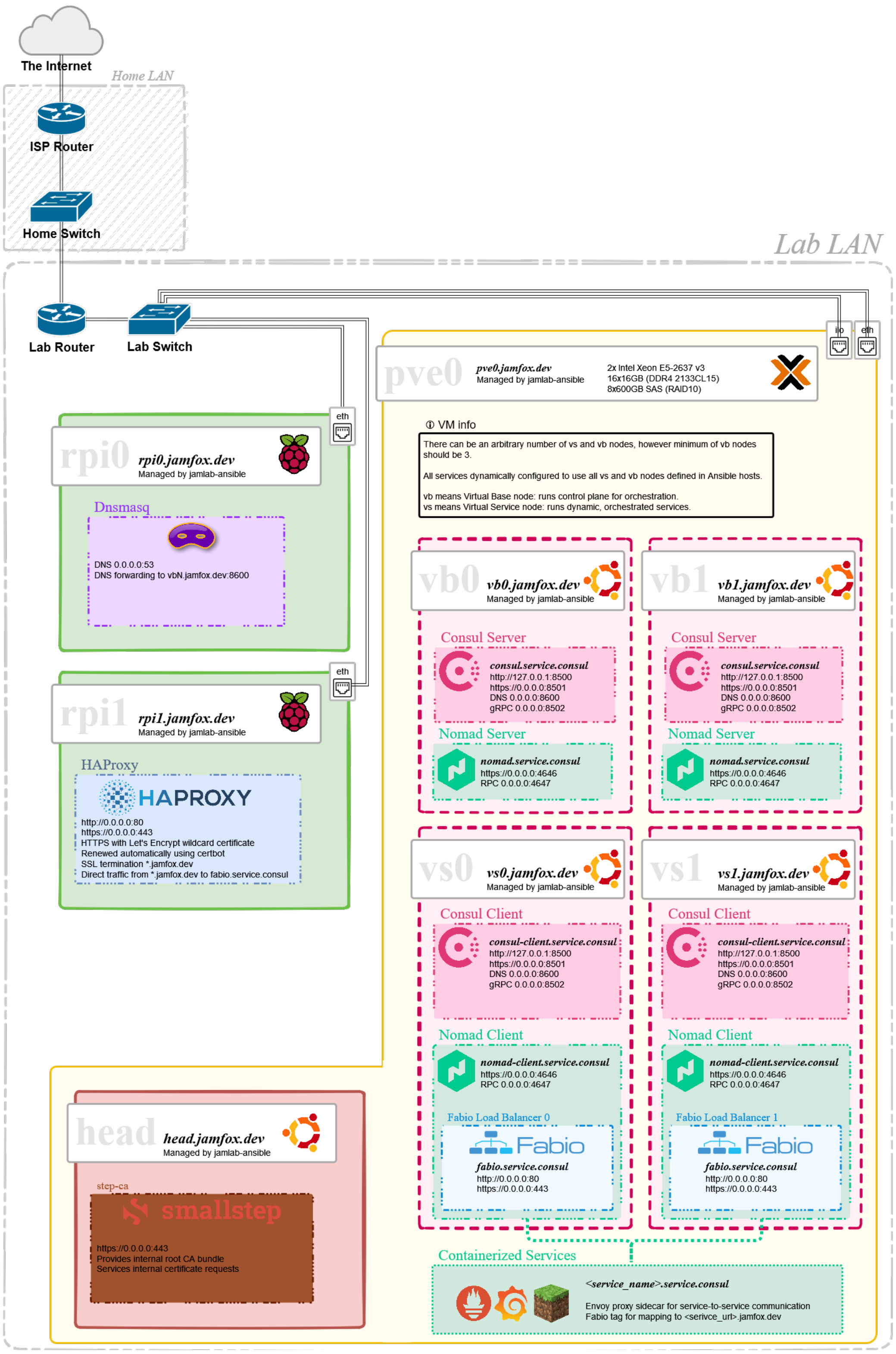
- Three base nodes forming the control plane of Nomad orchestration, responsible for scheduling, self-healing, roll-outs and rollbacks, storage, and auto-scaling of containerized services.
- Three service nodes constituting the service layer of the orchestration, where the services were deployed and operated.
- One support node, which was used to run the Step CA internal certificate authority and Ansible configuration system.

A diagram illustrating the entire project with both physical and virtual machines, along with key services, can be seen in Figure 4.

## 5.2 Testing failures

During testing, when a virtual machine from the control plane was shut down to simulate a host failure, the cluster continued to function seamlessly. Upon reboot, the previously shut down machine automatically rejoined the cluster.

However, when two virtual machines from the control plane were shut down to simulate a catastrophic failure, the cluster operation ceased. This occurred because having only one node available resulted in a loss of quorum and potential data loss, as a minimum of two servers is required for voting and data comparison. Functionality was restored



There can be an arbitrary number of vs and vb nodes, however minimum of vb nodes should be 3.

All services dynamically configured to use all vs and vb nodes defined in Ansible hosts.

vb means Virtual Base node: runs control plane for orchestration.
vs means Virtual Service node: runs dynamic, orchestrated services.

Figure 4. Overview of the physical, virtual machines and key services

after rebooting the shut down nodes and forcibly instructing the cluster to elect the last remaining node as the leader and utilize its data.

Testing also involved shutting down service nodes. The containerized services persisted until at least one node remained operational. This indicated that the project could have been implemented with only two service nodes instead of three, as the fault tolerance requirement for services differs from that of base nodes. Service nodes do not require a leader, thus two servers provide fault tolerance of one machine. However, services that need to be on multiple servers would not function if only one remained operational.

The downtime observed when a node service was shut down and a service without multiple instances was transferred to a remaining node was less than a minute. Although significant, this downtime was notably fast and automatic. In contrast, without an orchestrator, administrators would likely face longer downtime as they would need to be notified of the failure and manually address it.

## 5.3 Load Balancing Orchestrated services

Load balancing orchestrated services proved to be particularly intriguing for the author. Previous experience with relatively straightforward reverse proxy setups, where a static machine with a static port was defined for each service in the configuration, contrasted sharply with the dynamic nature of orchestrated services.

The absence of static hosts or ports emphasized the critical importance of robust service discovery mechanisms in dynamic environments like this. Additionally, integrating service discovery into load balancers requires meticulous attention, as most proxies do not inherently support service discovery integration and necessitate custom configuration. This custom configuration demands careful consideration to ensure both security and efficiency.

## 5.4 Viability of Orchestration Environments for Home Laboratory Purposes

The allure of orchestrated environments lies in their promise of eliminating concerns about host failures, automating roll-outs, load balancing, routing, and certificate management, as well as simplifying container management compared to manual configuration. Ultimately, it was determined that the resource and management overhead associated with orchestration outweighed its benefits. Running orchestration proved to be more time-consuming and costly compared to deploying plain containers or services directly on hosts manually.

While orchestration can offer significant advantages in larger environments where uptime is crucial, in smaller environments, the cost of operating the orchestration cluster often exceeds the requirements for all services running on the cluster many times over. However, for smaller environments like this project, Nomad proved to be exceptionally well-suited due to its simplicity compared to alternatives like Kubernetes. So, if the overhead of orchestration is justified for the specific requirements and scale of the project, Nomad emerges as a worthy choice.

# 6. Improvements

While the project provided valuable experience with dynamic orchestration and yielded a functional self-healing environment with a fault tolerance of one machine, three main criticisms emerged, suggesting areas for improvement.

Firstly, the approach to monitoring and logging lacked readability and actionable insights. The project relied solely on out-of-the-box solutions of installed services without implementing dedicated collection or analysis tools.

Secondly, there was a need for end-to-end encryption from the external load balancer to services. Currently, the HTTPS connection was terminated at the external load balancer, posing a security risk by placing excessive trust on the internal network's security.

Lastly, load balancing could be optimized by integrating Consul service discovery directly with the external load balancer. This approach would eliminate the need for internal load balancers on each machine, as the external load balancer is already a single point of failure.

## 6.1 Monitoring and Logging

In orchestration environments, the significance of monitoring and logging cannot be overstated. Traditionally, logging has been a fundamental aspect, capturing system events and activities. However, with the increased volume of logs and the complexity of orchestration environments, effectively finding and analyzing logs is challenging.

While Consul and Nomad provided logs for their operations that were readable, locating, accessing, and interpreting logs for orchestrated services running any number of Nomad service nodes proved to be a time-consuming task, making troubleshooting inefficient.

In the author's experience, slow networking and storage typically become bottlenecks in such environments. Despite having a network with a maximum speed of duplex 1GB/s, and utilizing 15K SAS disks, which are considered fast for spinning mechanical disks, these bottlenecks were not apparent during this project, especially since only lightweight services were tested. Nonetheless, without proper monitoring, it was impossible to provide empirical data to support claims of performance.

Efficient monitoring tools offer real-time visibility into various components of an orchestration environment, enabling tracking of resource utilization, system bottlenecks, and overall performance of orchestrated processes. This real-time feedback is invaluable for promptly identifying issues and optimizing system efficiency. Additionally, logging, when combined with advanced analysis tools, becomes a source for understanding system behavior and troubleshooting, rather than being overwhelmed by raw log data.

Monitoring and logging tools also play a significant role in security efforts. By tracking and analyzing system events, it becomes possible to detect and respond to security incidents in a more visible and timely manner. This proactive approach is essential for safeguarding potential sensitive data of services.

Investing time in monitoring and log analysis tools is a logical step towards enhancing security, as well as improving management and troubleshooting efficiency in this project.

## 6.2 HTTPS Termination at the External Load Balancer

In the pursuit of enhanced security within orchestration environments, the implementation of HTTPS encryption stands as a pivotal measure. However, terminating HTTPS at the external load balancer, as described in Chapter 4, necessitates careful consideration. While this approach ensures encryption of external communications, it exposes internal network traffic to potential vulnerabilities and trust issues.

When HTTPS termination occurs at the external load balancer, external traffic is decrypted upon entry into the internal network, leaving it vulnerable to unauthorized access or monitoring within the network. This creates a potential security risk, particularly if a node is compromised or a bug is introduced.

While encrypting external communications is imperative, it's essential to acknowledge the potential risks posed by decrypted internal traffic within the network. While network segmentation and trust in the laboratory environment may provide some level of assurance, implementing end-to-end encryption within the internal network is strongly recommended to mitigate the risk of unauthorized access.

Additionally, the approach to load balancing in the project involved using internal load balancers on each service node, orchestrated by an external load balancer due to limitations in direct integration with Consul service discovery. However, it would be more efficient to invest time in configuring service discovery to work directly with the external load balancer. This approach would eliminate the need for internal load balancers, reducing resource overhead and streamlining the load balancing process.

# 7. Summary

The goal of the report was to outline the author's efforts in creating a virtualized lab environment for a single-zone Nomad container orchestration cluster. This project diverged from the mainstream Kubernetes approach, opting for HashiCorp solutions to gain hands-on experience in designing, building, and managing dynamic infrastructure.

In conclusion the exploration revealed that the orchestration demands a significant upfront investment in time and effort. It calls for a more nuanced comprehension of underlying components compared to the straightforward manual deployment approach of utilizing plain containers without an orchestration layer. Furthermore, in the author's view the considerable physical infrastructure and management overhead associated with establishing and overseeing the orchestration layer imply that the advantages of scalability and flexibility become worthwhile only in larger environments and do not provide a lot in smaller environments (like the author's home lab) where a person may only be running a few of services at most. The resource overhead for running orchestration might exceed that of the service running on it, making it a very inefficient endeavour even if it does provide safety and ease of recovery from host failures. However, in scenarios where orchestration is necessary, Nomad emerges as an excellent choice due to its relative simplicity and ease of deployment and use.

Despite the author's decision not to employ the container orchestration approach for managing home services, the experience lead to numerous learning outcomes in building, administrating, and understanding modern dynamic infrastructure methodologies as well as ideas for further learning and improvement of the project.

## References

[1] Frank Denneman. *NUMA Deep Dive Part 4: Local Memory Optimization*. 2016. URL: https://frankdenneman.nl/2016/07/13/numa-deep-dive-4-local-memory-optimization/. Used 2023.28.11.

[2] L.P Hewlett-Packard Development Company. *HP ProLiant DL380 Gen9 Server User Guide*. 2014. URL: https://raw.githubusercontent.com/JamFox/docs.jamfox.dev/master/docs/content/homelab/attachments/DL380Gen9-UserGuide.pdf. Used 2023.28.11.

[3] Hewlett Packard Enterprise Development LP. *HPE Integrated Lights-Out (iLO)*. 2023. URL: https://www.hpe.com/us/en/hpe-integrated-lights-out-ilo.html. Used 2023.28.11.

[4] Proxmox Server Solutions GmbH. *Proxmox Virtual Environment - Open-Source Server Virtualization Platform*. 2023. URL: https://www.proxmox.com/en/proxmox-virtual-environment/overview. Used 2023.28.11.

[5] Inc. Software in the Public Interest. *Debian – The Universal Operating System*. 2023. URL: https://www.debian.org/. Used 2023.28.11.

[6] The Linux Foundation. *Open vSwitch: Production Quality, Multilayer Open Virtual Switch*. 2016. URL: https://www.openvswitch.org/. Used 2023.28.11.

[7] The Linux Kernel community. *Kernel Virtual Machine*. 2016. URL: https://www.linux-kvm.org/page/Main_Page. Used 2023.28.11.

[8] Raspberry Pi Ltd. *Raspberry Pi OS*. 2023. URL: https://www.raspberrypi.com/software/. Used 2023.28.11.

[9] Simon Kelley. *Dnsmasq*. 2023. URL: https://dnsmasq.org/. Used 2023.28.11.

[10] HashiCorp. *Packer by HashiCorp*. 2023. URL: https://www.packer.io/. Used 2023.28.11.

[11] HashiCorp. *Terraform by HashiCorp*. 2023. URL: https://www.terraform.io/. Used 2023.28.11.

[12] Karl-Andreas Turvas. *jamlab-packer*. 2023. URL: https://github.com/JamFox/jamlab-packer. Used 2023.28.11.

[13] Canonical Ltd. *Ubuntu 22.04.3 LTS (Jammy Jellyfish)*. 2023. URL: https://www.releases.ubuntu.com/22.04/. Used 2023.28.11.

[14] Canonical Ltd. *cloud-init: The standard for customising cloud instances*. 2023. URL: https://cloud-init.io/. Used 2023.28.11.

[15] Karl-Andreas Turvas. *jamlab-terraform*. 2023. URL: https://github.com/JamFox/jamlab-terraform. Used 2023.28.11.

[16] Hashicorp. *Nomad Reference Architecture*. 2023. URL: https://developer.hashicorp.com/nomad/tutorials/enterprise/production-reference-architecture-vm-with-consul. Used 2023.28.11.

[17] Inc. Red Hat. *Ansible is Simple IT Automation*. 2023. URL: https://www.ansible.com/. Used 2023.28.11.

[18] Karl-Andreas Turvas. *jamlab-ansible*. 2023. URL: https://github.com/JamFox/jamlab-ansible. Used 2023.28.11.

[19] Karl-Andreas Turvas. *ansible-parallel*. 2023. URL: https://github.com/JamFox/ansible-parallel. Used 2023.28.11.

[20] Jordan Webb. *The container orchestrator landscape*. 2022. URL: https://lwn.net/Articles/905164/. Used 2023.28.11.

[21] Hashicorp. *Nomad by Hashicorp*. 2023. URL: https://www.nomadproject.io/. Used 2023.28.11.

[22] The Linux Foundation and The Kubernetes Authors. *Kubernetes: Production-Grade Container Orchestration*. 2023. URL: https://kubernetes.io/. Used 2023.28.11.

[23] Hashicorp. *Consul by Hashicorp*. 2023. URL: https://www.hashicorp.com/products/consul. Used 2023.28.11.

[24] Brian Shumate. *Ansible-Nomad*. 2023. URL: https://github.com/ansible-community/ansible-nomad. Used 2023.28.11.

[25] Brian Shumate. *Ansible-Consul*. 2023. URL: https://github.com/ansible-community/ansible-consul. Used 2023.28.11.

[26] Inc. Smallstep Labs. *Step CA: Open-Source Certificate Authority & PKI Toolkit*. 2023. URL: https://smallstep.com/certificates/. Used 2023.28.11.

[27] Podman. *Podman*. 2023. URL: https://podman.io/. Used 2023.28.11.

[28] Docker Inc. *Docker Engine overview*. 2023. URL: https://docs.docker.com/engine/. Used 2023.28.11.

[29] Hashicorp. *Managing External Traffic with Application Load Balancing*. 2023. URL: https://developer.hashicorp.com/nomad/tutorials/load-balancing/external-application-load-balancing. Used 2023.28.11.

[30] HAProxy Technologies. *HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer*. 2023. URL: `http://www.haproxy.org/`. Used 2023.28.11.

[31] Electric Frontier Foundation. *Certbot*. 2023. URL: `https://certbot.eff.org/`. Used 2023.28.11.

[32] Internet Security Research Group (ISRG). *Let's Encrypt*. 2023. URL: `https://letsencrypt.org/`. Used 2023.28.11.

[33] Inc. Cloudflare. *Cloudflare Tunnel*. 2023. URL: `https://developers.cloudflare.com/cloudflare-one/connections/connect-networks/`. Used 2023.28.11.

[34] Frank Schröder and Education Networks of America. *Fabio LB*. 2022. URL: `https://fabiolb.net/`. Used 2023.28.11.