ChatGPT

Implementing a Custom Domain Extension in the ArquoLab Ecosystem

Introduction

Creating a custom **domain name extension** (i.e. a new top-level domain or TLD) for the Spectra/ArquoLab project can reinforce its identity and technical autonomy. The Spectra project is envisioned as an **ecosystem of modules and ideas** spanning science, art, and philosophy [1]. Its existing domains (e.g. *ecosys.mu*, *exosys.xyz*, *philmo.ch*, etc.) each embody a facet of this ecosystem [1] [2]. By establishing our own TLD, we can unify these facets under a single naming hierarchy while aligning with the project's ethos of **interconnection and decentralization**.

Currently, the network architecture consists of a **LAN environment** with Fedora 42 and Ubuntu 24 servers providing DNS and DHCP services, a static web host ("WEB-01"), and client devices (browsers, mobile) on the network [3] [4]. These servers also bridge between a private 10.10.10.0/24 subnet and a public static IP range, and they coordinate services like routing, firewalling, and even a WireGuard VPN for remote peers. This robust setup is the backbone on which we'll integrate the new domain extension.

In the sections below, we explore options for creating a new TLD—ranging from the official ICANN process to modern decentralized DNS—then detail how to **simulate and deploy** such a domain in our architecture using DNS/DHCP configurations, automation scripts (Fedora & Ubuntu), and additional services. We'll also ensure the solution reflects the project's ideology of an *"intertwined... mesh cluster network architecture"* (as described) by using decentralized, interoperable strategies. The goal is to produce a comprehensive deployment bundle (with **Bash**, **systemd**, and **Node.js** components) for managing the new TLD's lifecycle (registration, configuration, maintenance).

Options for Creating a Custom Domain Extension

Creating a new top-level domain can be approached in several ways. We compare three main strategies:

1. **Official ICANN New gTLD Process (Centralized)** – The conventional but costly route.
2. **Decentralized Naming Systems (Blockchain/Alt-DNS)** – Leveraging systems like Handshake or ENS to bypass central authorities.
3. **Private/Internal TLD Simulation** – Using our own DNS servers (no external visibility) for development or limited use.

1. Official ICANN gTLD Route – High Barriers

In the traditional DNS, all "real" TLDs are governed by ICANN. Launching a new gTLD requires applying to ICANN during their limited application windows, undergoing evaluations, and paying hefty fees. For example, the evaluation fee alone for the upcoming round is on the order of \$185k–\$227k [5] [6], plus infrastructure and compliance costs. The process is **complex, expensive, time-consuming, and**

centralized [7]. It's effectively out of reach for experimental or small-scale projects. We can safely rule this out as impractical for our needs (but it's important to note why alternative approaches arose in response to these barriers).

2. Decentralized Domain Systems – Owning Your TLD via Blockchain

Recent innovations allow creation of TLDs on **decentralized, permissionless networks** that function outside of ICANN's hierarchy. These systems shift trust to code and consensus, removing the single point of control at the DNS root. One prominent example is **Handshake (HNS)**.

Handshake is a blockchain-based naming protocol where every peer in the network validates and manages the root zone of the DNS [8] [9]. In other words, the root "directory" of TLDs is stored on a public blockchain and agreed upon by all nodes, rather than maintained by ICANN. This allows *anyone* to register a new TLD via an on-chain auction, with no central approval or exorbitant fee [10]. As Namecheap (an HNS-supporting registrar) describes, *"Handshake is a peer-to-peer network using blockchain technology — like a secure public registry… offering you more freedom, control, and security over your domain."* [11] In essence, Handshake makes the power of being a TLD registry accessible to the masses, reflecting the internet's original open ethos [10].

Some key points about Handshake and similar systems:

- **Owning a TLD**: On Handshake, you can bid on and purchase a TLD name (e.g. `.spectra` or `.ecosys`) using its native HNS coin. If you win, that TLD is yours in perpetuity on the blockchain [10] [12]. There's no renewal fee to a central authority – your ownership is secured by cryptographic consensus, not a WHOIS record [13]. This eliminates censorship or revocation risk; no authority can seize or block your TLD for "wrong" content if it's on the decentralized root [10]. This resonates with our project's emphasis on free expression and diversity (e.g. *e-libre.africa* stands for electronic freedom) [14].

- **Name Resolution**: Handshake's blockchain replaces the normal DNS root servers. However, below the TLD level it still relies on the existing DNS protocol. Importantly, *Handshake itself only manages TLD ownership/NS records, not subdomains* [15]. After registering a TLD on HNS, the owner must designate authoritative name servers (by setting NS records in the blockchain entry) for that TLD. Those name servers (which you operate or rent) handle the actual DNS lookups for any Second-Level Domains (SLDs) under your TLD [15]. For example, if we register `.spectra` on Handshake, we might set its NS to `ns1.spectra` pointing to our server. We would then run a DNS service on our server to resolve names like `app.spectra` or `ecosys.spectra`. This is similar to how traditional DNS delegates a zone, just that the delegation info lives on-chain.

- **Interoperability**: By design, alternate TLDs (like Handshake's) are **not recognized by default in the global DNS**. Users need to use a resolver that is Handshake-aware to resolve them [16]. This can be achieved by running a full node resolver locally, using a light client, or using public DNS services that support HNS. For instance, NextDNS and Cloudflare's 1.1.1.1 (with certain settings) have experimented with resolving Handshake names, and browsers like Brave have added limited support. There are also browser extensions and plugins that intercept ".hns" or other TLD queries. In our deployment, we can integrate a Handshake resolver in our DNS chain (more on this in integration steps) so that clients on our network (and VPN) can resolve the new TLD seamlessly. For

external users, we might provide instructions or even run a public recursive resolver that they can use. It's worth noting that **existing ICANN TLDs are reserved in Handshake** to prevent collisions [17] – e.g. `.com`, `.io` cannot be bought on HNS by random people; the rightful operators can claim them. New TLDs we create on HNS therefore live in an **alternative root** environment parallel to, but separate from, the ICANN root.

- **Other Decentralized DNS**: *Ethereum Name Service (ENS)* is another popular decentralized naming system. ENS lives on Ethereum and typically uses the `.eth` suffix, though it can also mirror existing DNS domains via DNSSEC proofs. ENS is mostly used to map names to Ethereum addresses or IPFS content hashes, and requires dApp browsers or plugins for resolution (normal DNS queries won't resolve `.eth`). *Unstoppable Domains* (.crypto, .zil, etc.) use NFT-based ownership for domains and similarly require special resolution (often via browser extensions). These systems are more about user-facing names (like replacing usernames/wallet addresses) and integrating with Web3. They are less suited for creating a whole new TLD hierarchy accessible to DNS clients. Handshake's approach – acting as a decentralized root for TLDs – aligns better with our goal of a full domain extension service.

Why choose a decentralized TLD? It provides **full autonomy** and aligns with the project's ideology of distributed authority and interoperability. For example, Octopuce (octopuce.ai in our domains list) symbolizes an intelligent entity with distributed cognition (an octopus' neurons spread across arms) [18]. In a similar spirit, a blockchain-based DNS distributes naming authority across many nodes (no single "brain" in control). It's resilient (no single point of failure at root) and censorship-resistant. Additionally, Handshake reinforces *"the original ethos of the internet as an open, free, and decentralized network"* [10] – a philosophy very much in line with Spectra's cross-disciplinary, open knowledge mission. Practically, it lets us **own our TLD outright**, rather than just renting subdomains. As the IPFS team puts it: *"Handshake allows anyone to register a top-level domain without a long approval process or prohibitively high financial barriers... You'd never have to worry about losing it for saying or doing the 'wrong thing.'"* [10].

One caveat: running our own TLD means we assume responsibility of a registry. Handshake provides the decentralized root and secure ownership records [19] [20], but we as TLD owners must manage the subdomains. We could keep all SLDs for internal use, or open them to collaborators/community. If we choose the latter, we'd essentially operate a mini **registrar**. Handshake's design allows this: we can either develop a custom registration system or leverage existing HNS registrar platforms that let others register names under our TLD [21]. (There are third-party services for some Handshake TLD owners to sell subdomains, analogous to how `.eth` subnames can be traded.) For our scope, we likely want a **public registry API** on our terms – perhaps a web portal or API where users (or project team members) can request subdomains under `.spectra` (or whatever TLD we choose). We'll detail this in the integration section, potentially using our Node.js server to handle such requests.

Blockchain as CDN/ledger: The user prompt also mentions *"decentralized hashed blocks tokenized ledger CDN."* This suggests integrating content addressing (like IPFS) with the domain. A powerful synergy is **Handshake + IPFS** for web content. We can deploy site content to IPFS (which gives us a content hash) and then use our decentralized domain to point to that content. In fact, the service Fleek (which we are already using for static hosting) has native support for Handshake domains: one can easily link an IPFS-hosted site on Fleek to a Handshake TLD [22] [23]. This means our new TLD's websites can be truly decentralized – the name is resolved via Handshake and the content fetched via IPFS. Fleek's integration ensures the content is persisted and delivered over a distributed network, functioning as a **decentralized CDN** for our assets [24]

[25]. By adopting this, when users access (for example) `spectra.gallery` or `ecosys.spectra` in a Handshake-enabled browser, the domain will resolve through the blockchain to our IPFS-hosted site, eliminating dependence on any single web server. This design perfectly echoes the project's theme of *"merging technology with social empowerment"* (as seen in e-libre.africa's description) [14] – we use cutting-edge distributed tech to empower free dissemination of our gallery's content.

Summary: Decentralized naming (Handshake) is a compelling option. It gives us control, aligns with our values, and has integration points with other decentralized tech (blockchains, IPFS). The trade-off is requiring some custom setup for resolution. But within our controlled environment (our servers and users), we can manage that via configuration and providing tools to resolve the new TLD.

3. Private/Internal TLD Simulation – Quick and Contained

The third strategy is to simply **create a custom TLD in our own DNS servers** for internal or testing purposes. This doesn't give us a globally recognized domain, but it's extremely useful for simulation, development, and ensuring our architecture supports the new domain structure before fully "going public." In fact, we have already done something similar: our DNS config defines a pseudo-domain `arquolab.io.net.lan` for the internal network [26]. This isn't a real TLD but a **local domain** ending in `.lan` that is served by our dnsmasq. Clients on the LAN resolve any `*.arquolab.io.net.lan` names via our server, which expands hostnames and provides local IPs [26]. We can take a similar approach to introduce, say, a `.test` or `.lab` TLD locally.

How to simulate a TLD locally: Using dnsmasq (which we run via NetworkManager), we can declare our DNS server authoritative for a fake top-level domain. For example, we could choose `.ecosys` (just as a test TLD inspired by our project name) or even use something like `.arpa` in a lab context. In dnsmasq, this is done with the `local` and `domain` directives. A config snippet might be:

```
local=/ecosys/
domain=ecosys,10.10.10.0/24,local
```

This would cause our server to treat any query ending in `.ecosys` as local and respond based on entries we provide, without forwarding to upstream DNS [26]. We could then add static host entries or DHCP-hosted names under that TLD. DHCP can even suffix the domain (using the `domain` setting) to hostnames it gives out. We saw this with `arquolab.io.net.lan` where dnsmasq appends that domain for DHCP leases in 10.10.10.x range [27]. Clients get the search domain via DHCP option 15 (if configured), so they can resolve short names.

Concretely, to test our new extension, we might configure `.dev` or `.lab` as a fake TLD on our two servers and add a few sample records (e.g. `test.dev` -> 10.10.10.100). By pointing our test clients' DNS to the servers (which DHCP already does, handing out 10.10.10.1 as DNS [28]), those clients can resolve the fake TLD names. This approach requires **no special software** beyond our existing dnsmasq. It is limited to our network (or anyone using our resolver), but it's great for validating the concept and integrating with internal systems (for instance, testing that our web apps and UIs can handle new domain patterns).

We can maintain this internal TLD indefinitely for intranet services if desired (many companies use internal domains like `.internal` or non-public `.corp` domains similarly). However, if we want external users to resolve our domain extension, we'd eventually need to either register it officially or use an alt-root like Handshake. One compromise solution some communities use is **OpenNIC** – a volunteer-run DNS network that includes some custom TLDs. If our goal was to make a custom TLD available to the public without blockchain, we could propose it to OpenNIC and, if accepted, their DNS servers would resolve it for any users who opt into OpenNIC DNS. But this has niche adoption and is less aligned with our long-term goals than the Handshake route, so we mention it only for completeness.

Comparison: In summary, the internal simulation is **simple and under our full control**, but not globally accessible. The decentralized approach is **powerful and future-forward**, enabling global reach to those using decentralized tech, and it future-proofs our project's web presence (should browsers natively support these in the future, as momentum suggests [29] [30]). Given Spectra's experimental and ambitious nature, a combination of **(2) and (3)** is appealing: we can develop and test internally, *then* launch on Handshake to share our new TLD with the world (especially the Web3 community) – all while avoiding ICANN bureaucracy.

Next, we'll assume the plan to use **Handshake for the official implementation** of our TLD (for example, registering ".arqlab" or ".spectra"), and simultaneously configure our local network to recognize this TLD (and perhaps a test TLD in the interim). We'll detail how to integrate this with our architecture, from DNS server configs to automation scripts and registry API.

Integrating the New Domain Extension with Our Architecture

Integrating a new domain extension involves two aspects: **(A) DNS/DHCP configuration** changes in our network, and **(B) running any new services** to support the TLD (e.g. blockchain node, authoritative DNS, and registry API). We will address these in turn, building on the existing infrastructure:

A. DNS and Network Configuration for the TLD

Our current DNS setup is managed by *dnsmasq* via NetworkManager on both Fedora and Ubuntu servers. They service two networks: the LAN (10.10.10.0/24, with local domain `arquolab.io.net.lan`) and the public IP subnet (our static range 144.2.68.224/27, with domain `arquolab.io`) [26] [31]. Key configuration files from the repo include:

- `01-DNS-arquolab-io-net-lan.conf` – defines the **local LAN domain** and DNS settings [26]. For instance, it sets `arquolab.io.net.lan` as the local domain for 10.10.10.0/24 and uses `addn-hosts=/etc/dnsmasq.hosts` to include static host mappings [26].
- `02-DHCP-arquolab-io-net-lan.conf` – sets up **DHCP** on the LAN interface, including IP range and options [32]. Our LAN DHCP is authoritative on interface `enp7s0f0`, leasing 10.10.10.150-200 and reserving some static leases (e.g. `koalab-router` at .2) [33].
- `03-DHCP-arquolab-io.conf` – configures DNS/DHCP for the **public subnet** on interface `eno8403` [34] [35]. Here, the domain `arquolab.io` is treated as local for IPs in 144.2.68.224/27 (meaning our DNS can provide authoritative answers for reverse lookups and hostnames in that range). Notably, it shows DHCP handing out addresses .27–.29 to specific MACs and using .225 as the gateway [35] (likely the ISP router) while setting 1.1.1.1 as DNS for those hosts [36]. This confirms our

server bridges the public block and that we rely on upstream DNS for external queries except our own domains.

To integrate a new TLD (let's call it `.spectra` for this discussion), we need to update DNS in two areas: resolution *within our network* and resolution *for external queries*.

1. Local Resolution (DNS Stub/Resolver changes): We want devices on our LAN (and VPN peers) to resolve "*.spectra" names. If we set up `.spectra` via Handshake, those queries normally would go to the root (which our server would forward to 1.1.1.1 and get NXDOMAIN since the normal root doesn't know it). Instead, we will configure our dnsmasq to intercept `.spectra`. We have two possible modes:

- **Full resolver mode**: Run a Handshake resolver locally and forward **all** unknown queries to it. For example, we could run `hnsd` (a lightweight Handshake DNS stub) on port 5350 on our server, configured to respond for Handshake TLDs. Then in dnsmasq config: `server=/./127.0.0.1#5350` with appropriate routing would send any query that isn't answered by cache or other rules to the handshake resolver. However, this might also try to send normal TLD queries to hnsd (which could forward to ICANN root if configured with ICANN fallback, or we can split horizon by domain). A more targeted approach is below.

- **Split-horizon (by TLD)**: Only forward queries for our specific TLD (or generally, non-ICANN TLDs) to the Handshake resolver. For instance: `server=/.spectra/127.0.0.1#5350`. This means "for any domain under .spectra, ask the local handshake resolver". We could also add known Handshake TLDs we plan to use (if more than one) similarly. This way, regular domains still go out to 1.1.1.1 (as configured) [37], and `.spectra` is handled internally. We will likely implement this, as it is simple and keeps normal DNS performance unchanged.

- **Local caching**: dnsmasq will cache results from the handshake resolver as well, which is good. We should ensure to enable DNSSEC validation in dnsmasq if we want security. Handshake's resolver (if full node) might also provide proof checking of name claims. In early phases, we might not enable DNSSEC in dnsmasq (since we are largely dealing with either our own domains or already validated Handshake data), but it's an option.

In practice, we will create a new config file in our repo, e.g., `configs/dnsmasq.d/30-DNS-spectra.conf`, with contents like:

```
# Handshake .spectra TLD resolution
server=/.spectra/127.0.0.1#5350
```

(assuming our handshake stub listens on 127.0.0.1:5350). We then include this in the deployment script so it gets installed to `/etc/NetworkManager/dnsmasq.d`. Both the Fedora and Ubuntu apply scripts already install a series of DNS config files [38]; we'd append our new file to that list in each script. Once added and NetworkManager restarted, our local resolver will direct ".spectra" queries appropriately.

Additionally, for **development** or immediate simulation, we could even skip the handshake stub initially and just define some hosts under `.spectra` statically to test our apps. For example, add to `/etc/dnsmasq.hosts`:

```
10.10.10.8   test.spectra
10.10.10.9   db.spectra
```

and a config `local=/spectra/` similar to above. That would let us ping `test.spectra` on LAN. However, final integration aims to hook into Handshake, so the stub/forward method is preferred, as it will dynamically resolve names under our TLD that are registered on-chain (and cached locally).

2. Authoritative DNS for the TLD (external visibility): For external users to resolve names in our TLD, the Handshake blockchain needs to know which DNS servers serve `.spectra`. When we won the TLD, we would have set an NS record on-chain pointing to (say) `ns1.spectra` and `ns2.spectra`. We must run DNS servers at those hostnames/IPs. Since we have a static public IP, we can use it for `ns1` (and perhaps get a second IP for `ns2` or use the same with different name – less redundancy but possible). **Our Fedora/ Ubuntu servers can act as these authoritative DNS servers.** In fact, we can dual-purpose them: they are currently acting as **resolvers** (for our clients) but can also run an authoritative service for the zone. Dnsmasq is mainly a lightweight resolver/cache and DHCP server; it is not typically used as a public authoritative server for a major zone (especially since it serves records from hosts/DHCP config without zone transfers, etc.). For a robust setup, we might introduce a dedicated DNS server software (like BIND, NSD, or Coredns) to serve the `.spectra` zone to the outside world. This authoritative server would load the zone data (the records for *.spectra) and answer queries from any internet host.

A simple plan is: **Use BIND9 or NSD for authoritative service** on the public interface (port 53). We can automate its config: generate a zonefile for `.spectra` containing our initial records (e.g. NS records, maybe some A records for key services or a wildcard). For example:

```
; Zone file for spectra TLD
$ORIGIN spectra.
@   3600 IN SOA ns1.spectra. admin.spectra.gallery. (2025080101 7200 3600
1209600 3600)
    3600 IN NS ns1.spectra.
    3600 IN NS ns2.spectra.
ns1 3600 IN A 144.2.68.227   ; our primary DNS server IP
ns2 3600 IN A 144.2.68.228   ; (if we have a second IP or use one of our VM's
IP)
; Example records
ecosys 3600 IN A 10.10.10.8    ; maybe pointed internally or to NAT
gallery 3600 IN A 144.2.68.227 ; pointing to our web server
*       3600 IN A 144.2.68.227 ; wildcard catch-all
```

And so on. The zone could also contain DNSSEC keys if we want to sign it and put the DS in Handshake (Handshake allows a DS record in its on-chain entry for a TLD). This would allow validating resolvers to ensure our zone responses are authentic [39], adding security (though this is an advanced step we can consider after initial deployment).

We should run this DNS service under systemd. We'll create a systemd unit for named (if BIND) or nsd. Our deployment scripts can drop the zonefile into `/etc/bind/zones/spectra.zone` and a named.conf

snippet to load it, then enable the service. The *Fedora & Ubuntu automation scripts* we have can be extended to install and configure BIND9 in a similar fashion to how they install WireGuard and dnsmasq. For instance, add `dnf install bind` (Fedora) / `apt install bind9` (Ubuntu), then copy config files from our bundle (which we will add to the repo, e.g. under `configs/bind/`).

Since we have two servers, we can run one as **primary (master)** and one as **secondary (slave)** for the zone. This achieves redundancy and fits the idea of a *"non-blocking peer-to-peer pool of servers acting as a quorum"*. In practical terms, we configure the zone as master on Fedora (ns1) and as slave on Ubuntu (ns2). Zone transfers (AXFR/IXFR) will propagate updates. If one server goes down, the other can still answer globally, and our Handshake NS records list both. This is how traditional DNS achieves resilience through multiple authorities. We will secure the transfer with TSIG keys. All these details can be automated in our bundle (generating a TSIG key and updating both configs with it, etc.).

DHCP considerations: Our DHCP servers currently hand out the search domain `arquolab.io.net.lan` (implicitly via the `domain=` config) to LAN clients and set DNS to 10.10.10.1 [40]. We might want to also push the new TLD as a search domain. For example, if we want internal hosts to resolve unqualified names like `ecosys` to `ecosys.spectra`, we could add `domain-search=spectra` in dnsmasq or adjust the `domain` option. However, mixing two search domains can be tricky; instead, we might simply rely on full queries for `.spectra` since it's short and recognizable. DHCP will continue to provide our server as DNS (option 6) [41], which is the main thing needed. We might not change DHCP options at all for this integration, aside from possibly including our new NS in any DHCP info if relevant (not really, since clients use recursive DNS, not NS directly).

Firewall: Ensure that UDP/53 and TCP/53 on the public interface are open on our server(s) so they can answer external DNS queries. In our Fedora script, we opened UDP 32237 (WireGuard) and enabled masquerading [42]; we'll similarly do `firewall-cmd --add-service=dns`. On Ubuntu, we'll add `iptables` rules for port 53 or adjust UFW if enabled. This will be reflected in the automation steps.

At the end of this DNS configuration, internally our network can resolve the TLD (via the forwarding to handshake stub), and externally any user pointing to our NS or using a Handshake resolver will resolve it as well. We'll effectively be running a local **split-horizon DNS**: our resolver knows about internal hostnames (.net.lan, etc.) and forwards `.spectra` to handshake, while our authoritative servers answer `.spectra` queries from outside and possibly also provide public records for our existing domains.

Integration of existing domains: We shouldn't forget to integrate our current domains (ecosys.mu, exosys.xyz, etc.) with the new architecture. Owning those, we can choose to point them to our new infrastructure. For example, we could set DNS A records for `ecosys.mu` to our static IP, and then handle it via our web server (perhaps redirecting or serving content). Another idea is to use them as CNAMEs or as alternate pointers to the same content that will be on `.spectra`. The domain descriptions show they complement each other (micro vs macro systems, etc.) [1]. A unified TLD `.spectra` could host subdomains matching those concepts (e.g. ecosys.spectra, exosys.spectra), creating a semantic grouping. We can document in our strategy that *ecosys.mu* might mirror *ecosys.spectra* for accessibility. Technically, that means our authoritative DNS for ecosys.mu (which we can also run on our server or manage via existing registrar DNS) would have records identical or pointing (CNAME) to those under spectra. This ensures interoperability between the old and new naming schemes – users can reach the services via either the legacy domain or the new TLD, which is useful during transition.

B. Deployment Automation (Fedora 42 & Ubuntu 24) and Service Integration

With the design in place, we now focus on automating the deployment and maintenance. We aim to create a **"bundle"** – a collection of scripts, config files, and possibly systemd unit definitions – that can set up everything on both Fedora and Ubuntu servers in a reproducible way. We already have a strong starting point in the `uniphilabs/ark/exosysmu` scripts:

- **apply-fedora.sh / apply-ubuntu.sh**: These install base packages (NetworkManager, dnsmasq, wireguard, etc.) and then copy our config files into place [43] [44]. They also bring up the network interfaces (bridge, uplink, port), enable forwarding, configure WireGuard, and set firewall rules [45] [46] [47]. After running these, a fresh Fedora/Ubuntu machine is essentially transformed into our router/DNS node. We will extend these scripts.

Here are the integration steps we will automate:

1. Package Installation: We need additional packages for our new services: - **Handshake node**: There isn't an off-the-shelf package for hsd (Handshake daemon) in Fedora/Ubuntu repositories as it's a niche software. We have options: use Docker, use a prebuilt binary, or build from source. Perhaps the easiest is using **HNSD** (the lightweight resolver) which is a C implementation that can run as a validating stub for handshake. If packaging is an issue, we can run Handshake via Docker container (many run hsd in a container). For automation, an approach is: download the latest hnsd release binary in the script or add a step to build it. For now, let's assume we can include an hnsd binary in our repo or fetch it. Our script will ensure it's placed in `/usr/local/bin` and set to run as a service. - **DNS server**: Install `bind9` on Ubuntu and `bind` (or `named`) on Fedora. Similarly, install `bind-utils` for tools if needed. Alternatively, use NSD (smaller footprint authoritative DNS) – but BIND is fine. We update `dnf -y install ...` and `apt-get install -y ...` lines in the scripts to include these packages.

2. Configuration Files: We add the following to our configuration directory (in repo) and have the script deploy them: - **NetworkManager dnsmasq config for TLD**: e.g. `30-DNS-spectra.conf` as discussed. Script copies it to `/etc/NetworkManager/dnsmasq.d/`. - **dnsmasq.hosts**: Update if we want any static host entries (like above test.spectra) – but if using handshake for resolution, we might not need to hardcode any `.spectra` host here. We will primarily rely on our authoritative DNS for actual records. We may still add some entries for convenience (like pointing `ns1.spectra` to our local IP so that from inside, ns1 resolves – although in our public DNS we'll have that anyway). - **BIND zone file**: e.g. `spectra.zone`. Script should install this to BIND's zone directory. We might templatize the IP addresses (the script can substitute the actual public IP or second IP if available). - **BIND config**: an excerpt to include the zone. For instance, `spectra.conf` with:

```
zone "spectra" {
    type master;
    file "/etc/bind/zones/spectra.zone";
    allow-transfer { 144.2.68.228; }; // Ubuntu server IP
    also-notify { 144.2.68.228; };
};
```

And on the Ubuntu (slave) side:

```
zone "spectra" {
    type slave;
    masters { 144.2.68.227; };  // Fedora server IP
    file "/var/lib/bind/spectra.slave";
};
```

We will adjust the script to detect if running on Fedora vs Ubuntu (they could use the presence of `dnf` vs `apt` or an env var) and copy the appropriate config. Our repo can have both in subfolders if needed. - **Systemd service for handshake**: We create a unit file `hnsd.service` with something like:

```
[Unit]
Description=Handshake Light DNS Resolver
After=network.target
[Service]
ExecStart=/usr/local/bin/hnsd -s 127.0.0.1:5350 -c /var/lib/hnsd -u nobody
# The above runs hnsd on localhost port 5350, using /var/lib/hnsd for its chain
data.
Restart=on-failure
[Install]
WantedBy=multi-user.target
```

The script will install this to `/etc/systemd/system/`, create the `/var/lib/hnsd` directory, and possibly preload the root of the handshake chain (hnsd can bootstrap by connecting to peers). We'll then `systemctl enable --now hnsd` to start syncing. **Note:** hnsd is a stub resolver; it will fetch name proofs on demand. Alternatively, if we ran a full node (hsd), the approach is a bit different (it would maintain a full blockchain). hnsd as a lightweight resolver is sufficient if we just need resolution and are okay with relying on peers for each query's proof. This keeps resource usage low, suitable for our servers which do many other things.

- **Systemd config for BIND**: On Ubuntu, the package will include a service. On Fedora, likewise. We just need to ensure it's enabled (`systemctl enable --now named`). Our script can handle that. No custom unit needed unless we containerize it.

3. Execution Order: The apply script or a new script (we might call it `apply-ext.sh` for "apply extensions") will perform steps in a safe order: 1. Install packages (including any from our bundle like hnsd). 2. Deploy config files (dnsmasq, bind zone, bind conf, etc.). 3. Reload/restart services: - Restart NetworkManager (to reload dnsmasq config) [48] . - Start/enable **hnsd** (handshake resolver). - Start/enable **named** (BIND). The master should be started before the slave so it's ready to transfer zone; or we handle that manually the first time. - (WireGuard and firewall are handled in original script – we keep those, adding DNS port rules as mentioned). 4. Validation: We can extend the `vrb-validate.sh` script to test the new TLD. For example, after existing checks, do:

```
echo "== DNS Handshake TLD test =="
dig +short @10.10.10.1 example.spectra || echo "Query failed"
```

Initially, `example.spectra` would fail (if no such name) but at least we see that our server did not error. We could add a known record in our zone (like `test.spectra`) and then test that it returns the expected IP from both local dnsmasq and via an external query through the authoritative server (the latter could be tested from a peer or using `dig @144.2.68.227 test.spectra`). These tests would give confidence the system is working. Our bundle's documentation will instruct the operator to run the validate script and observe outputs.

4. Node.js "Public Registry API" Integration: We have an Express/Node.js server in the repo (the *interwined-cluster* server) [49] [50]. We can extend this application to provide a UI or API endpoints for managing domain records. For instance: - **Domain Registration API**: An endpoint like `POST /domains` where authorized users can submit a subdomain name and maybe some metadata, and the server will attempt to register it. In the Handshake context, "register subdomain" means simply adding a DNS record in our zone (since subdomains aren't on-chain). So the API could validate the name (ensure it's not taken, meets character rules, etc.), then update the BIND zone file or (better) use a dynamic update mechanism. - We could use DNS update (RFC 2136) from the Node app – BIND can be configured to allow dynamic updates with a key. However, since we already are managing zone files in Git, an easier path for now: the Node server can append the record to a zone file template and reload BIND. This requires the Node process to have permission or to call a wrapper script. Alternatively, store desired entries in a database and periodically regenerate the zone file. For initial deployment, this might be overkill if only project admins will set records; they can edit zone files directly or via GitOps. But providing an API lays groundwork for future public participation (imagine a web form for community members to get theirname.spectra if we ever open it up). - **Integration with UI**: We could add a section in our Spectra Gallery or Uniphil Labs portal that shows the list of active subdomains under the TLD and perhaps what they point to. It could allow certain users to manage them. This is more a future enhancement; the key is we plan for an "intertwined" control – not completely manual editing of DNS, but through our unified software stack.

Given time constraints, a simpler approach is to use Infrastructure-as-Code/DevOps: manage the `.spectra` DNS zone in our Git repository (as we do with other configs) and treat adding a subdomain like a code change that goes through version control. This ensures traceability and fits into maintenance workflows (maybe with CI to auto-deploy the updated zone file to servers). We can outline this strategy in documentation as an alternative to building a full API right now.

However, since the user prompt explicitly says "public registry API," we should at least outline how it would work in Node: - Use our Express server to implement REST endpoints for domain operations (list, register, delete). - Protect them with authentication (our server already has an auth system with JWT/cookies from what looks like setupAuth, registerAuthRoutes in code [51] [52]). - On registration, the server code adds a DNS record. We could invoke a shell command to update DNS:

```
child_process.exec(`nsupdate -k /etc/bind/keyfile <<EOF
server 127.0.0.1
zone spectra
update add newname.spectra. 3600 A 10.10.10.123
send
EOF`);
```

This would dynamically tell BIND to add a record (if we configure BIND to allow updates from localhost with that key). This avoids editing zone file manually. - Alternatively, the Node app could simply write to a JSON or DB and an admin would periodically reconcile it.

We will include in documentation that such integration is possible and in line with the project's *"modular structure"* and web-centric control (the idea that managing infrastructure through code and APIs is a form of **augmented collective intelligence**, one of the themes mentioned). For now, our bundle's main deliverable is the functioning network and DNS; the Node API integration can be an optional extension.

5. Mesh and Consensus Aspects: In the architecture description, there's a lot of emphasis on a *"non-blocking peer-to-peer decentralized pool of servers acting as a quorum"* and decision trees that self-validate in parallel. We can map these concepts to our implementation as follows: - The **two DNS/DHCP servers** form a mini-cluster. We configure them such that one is primary and one backup for DHCP (or split scopes to avoid collision). For DNS, as discussed, they will share authority for the new TLD via master/slave replication – effectively a *quorum of two*. They also both run the handshake resolver, each validating blockchain data, so even if one had stale data, the other could still resolve (though typically both will sync via hnsd). This provides a level of fault tolerance (non-blocking in the sense that one node's downtime doesn't block name resolution entirely). - The **blockchain network** (Handshake) itself is a peer-to-peer consensus system – this is the parallel validation process ensuring the root zone integrity outside of our servers [19] [20]. So, while our authoritative DNS handles second-level queries in a hierarchical manner (decision tree), the existence of our TLD in the root is maintained by a decentralized mesh of Handshake nodes. This nicely mirrors the notion of a *"mesh meta cluster that can self-validate authority in parallel to the main decision tree"*. In plainer terms: any changes to who owns the TLD or its delegation require blockchain consensus, which runs in parallel to DNS queries that our servers answer. Both layers ensure consistency: if someone tried to spoof our TLD, it would fail because it wouldn't have the Handshake cryptographic proof; if our DNS tried to serve an unauthorized record, Handshake's DS/DNSSEC (if we use it) could catch it. Thus, authority is validated at multiple layers. - **Non-blocking architecture**: By using local caching and not relying on any single external resolver, clients on our network get fast responses. Even if the handshake node is momentarily unreachable, our dnsmasq can cache recent answers or our authoritative server will still serve known records. And even if the blockchain is momentarily slow (say, updating a name might take a block confirmation), normal DNS queries aren't blocked by that – they either resolve from existing info or not. This separation of concerns (slow-moving consensus vs fast queries) keeps the user experience smooth. - **Decision tree from transistors to interactive data**: If we wax philosophical (since the user prompt does so), one could say our solution implements a hierarchical naming **tree** (the DNS zone and domain structure) on top of a network of **nodes** (Handshake peers) that agree on the root. It's analogous to hardware logic gates (discrete yes/no decisions) supporting higher-level algorithms – here, the low-level consensus algorithm yields a high-level naming system that our applications use. This structured-yet-distributed design captures both **order and chaos**: the deterministic tree of DNS records and the dynamic, emergent consensus of the P2P network. As the *Kaophi* domain description notes, it's a fusion of *"chaos and φ (phi)"*, i.e. unpredictability and mathematical order [53]. Our domain extension architecture similarly blends the unpredictable decentralized domain auctions with a predictable DNS lookup structure.

Workflow: Deployment, Validation, and Maintenance

Finally, let's outline the **workflow** to implement and maintain this architecture step by step, as a guide for our team:

1. Preparation: Acquire the TLD on Handshake (e.g. via Namebase or running our own auction if not already done). Once secured, set its NS records to point to our server hostnames (e.g. ns1.spectra -> 144.2.68.227, ns2.spectra -> 144.2.68.228). Also, prepare our existing domain DNS settings if we plan to add records pointing to the new TLD (not strictly required, but for integration).

2. Server Setup (Automated by Bundle): On a fresh Fedora 42 or Ubuntu 24 server (or our existing ones if re-provisioning): - Run the unified deployment script (e.g. `./apply-full.sh`). This script will internally call the existing apply-fedora/ubuntu steps and then the new extension steps. - The script will output progress of installing packages, copying configs, and enabling services. We should watch for any errors (e.g. if BIND fails to start due to a zone file syntax issue). - It will conclude with a message like "[VRB] Apply complete" (from original script) [54] and perhaps additional info from our new steps.

3. Post-Deploy Configuration: If not baked into the script, perform any manual steps: - For master/slave DNS: initiate a zone transfer if needed (though if we started master first, the slave should fetch automatically). Check logs on slave to confirm it pulled the zone. - Double-check firewall rules: ensure port 53 is reachable from outside (we can test using an external server with dig). - If the handshake node needs to sync the blockchain (for a new TLD to be recognized), this might take a short time (hnsd will fetch proofs on first query, or we can preload it by querying our TLD once). We can do a dummy `dig NS <newTLD>. @127.0.0.1#5350` to trigger hnsd to retrieve the root info for that TLD.

4. Validation: Run our `vrb-validate.sh` or a similar test script: - It should confirm the network interfaces IPs (to ensure bridge and others are up) [55]. - Test a known local DNS name (like `koalab-router.arquolab.io.net.lan`) [56] – this should return 10.10.10.2 as configured. - Test the new TLD: - Inside the server: `dig +short test.spectra @127.0.0.1#5350` (direct to handshake stub) to ensure the stub is working (likely NXDOMAIN if not set, but no connection error). - From the server's own resolver: `dig +short test.spectra @10.10.10.1` – through dnsmasq. If we added a "test" record in the zone, this should return the IP. If not, it might return nothing or NXDOMAIN (which is fine for now). - External test: From an outside machine (or our laptop on the internet), use a handshake-aware resolver (or our server directly). For example, `dig +short NS spectra. @ns1.spectra` should show the NS records (which point to itself), or `dig +short test.spectra @144.2.68.227`. If we get a response, our authoritative DNS is successfully serving to the world. - The WireGuard and other services can also be tested (to ensure we didn't break them). E.g., ensure a WireGuard peer can still connect and resolve DNS through the tunnel (the peer config uses 10.10.10.1 as DNS [57], which now also knows about .spectra).

5. Maintenance and Monitoring: With everything running: - The **handshake node (hnsd)** will periodically need to be updated to maintain compatibility with network upgrades, if any. We should monitor its output (perhaps redirect logs to syslog) and ensure it stays connected to peers. Since it's a light client, it should be low-maintenance. If we prefer a full node (for independence), we'd have to monitor blockchain sync and allocate more disk. For now, hnsd is enough and we can rely on public peers. - **DNS servers:** BIND can be set up with logging (at least for queries or updates) to monitor usage. We should keep an eye on zone serial numbers. If using GitOps, any manual edit to zone should increment serial and be committed. We might implement a CI hook: whenever the zone file in repo changes, push it to servers (via `rndc reload` or such). - **DHCP/DNS on LAN:** These are stable, but if we add new devices that need static mapping, we update `dnsmasq.hosts` or the `dhcp-host` entries (e.g., we might eventually give our *WEB-01* host a spectra domain internally). - **Security**: We will generate a TSIG key for zone transfers and keep it safe. If enabling DNSSEC for the zone, we manage key rollovers. Also, ensure our servers are locked down (they already have firewalls – only necessary ports open). - **Backups:** The bundle's configuration (zone files, etc.)

lives in our git repository, so we have a backup of critical info. We should also back up any state like the Handshake node's data (though reconstructible) and perhaps an export of our zone records in a pinch.

6. Client configuration: Document for users (or internally handle) how clients can resolve the new TLD outside the LAN. Two scenarios: - **On our LAN / VPN**: Nothing special – they already use our DNS, which now knows the TLD. - **Outside (public internet)**: To truly resolve `.spectra`, a user either uses a DNS resolver that supports Handshake or configures their system to use our DNS server for it. We could run a public recursive resolver on our server that listens on a standard port. Or instruct advanced users to use NextDNS with HNS enabled, etc. Since our goal is to be "public," we should consider running an open resolver (with rate-limiting) that serves our TLD (and maybe forwards others normally). However, open resolvers can be misused for amplification attacks; a safer approach: encourage use of a specific configured resolver or browser plugin. This is more of a community outreach issue. We'll note these options in docs (e.g., "to access our sites on .spectra, you can use Fingertip (Handshake resolver) or the Bob Extension in Chrome, etc., or point your DNS to 144.2.68.227 (which will resolve .spectra in addition to normal domains)"). Because our project is likely targeting tech-savvy users, adopting a browser plugin or adding a DNS might be acceptable.

By following this workflow, we ensure a **"legit basis, integration and validation, devops"** for the new domain extension across all layers: from low-level network (DHCP/DNS) up to high-level application logic (Node.js API, web content). Each component plays a role in the overall *interoperable mesh*. The Fedora and Ubuntu servers remain the core of the network services (the **authority within the LAN**), while the blockchain and external DNS make up the **authority to the outside world**. Together, they uphold a unified namespace.

Conclusion and Future Outlook

We have designed and outlined a comprehensive solution to create and integrate a custom top-level domain into the Spectra/ArquoLab architecture. By leveraging Handshake's decentralized DNS, we **mimic standard DNS practices** (delegation, NS records, etc.) while gaining freedom from central control [9] [58]. We adhered to compatibility and internet policies – our implementation still uses DNS protocols (so standard browsers can work via a compatible resolver) and we considered security via DNSSEC/DS records to align with best practices [39]. At the same time, we pushed the envelope by using a *"public registry on a blockchain ledger"* where appropriate [11], and tying it into a *decentralized CDN (IPFS via Fleek)* for content delivery [25].

This strategy reflects the project's philosophical stance: a **unified philosophy (Uniphil)** bridging Western and Eastern thought [59] in concept translates to a unified network bridging traditional and decentralized paradigms in implementation. Our domain extension `.spectra` becomes a metaphorical *"spectra of perspectives"* – it's accessible via Web2 (traditional DNS) and Web3 (Handshake/IPFS) alike, embodying the spectrum of old and new technology.

Moving forward, as we deploy this, we'll gather metrics and feedback. We should remain agile to adjust (for example, if a browser begins to natively support Handshake, we can drop instructions for plugins). We'll also monitor the Handshake community's developments – ensuring our solution stays updated with the latest best practices (or migration if ever needed).

The outcome will be a fully automated bundle that any team member can run to set up the environment, plus documentation (like this) describing the architecture and reasoning. By following this plan, we empower the Spectra project with its own domain realm, reinforcing its identity and technical sovereignty in an interconnected, multi-domain world.

Sources:

- Spectra/ArquoLab internal docs and configs for network architecture [4] [26] [33].
- DNS decentralization references (Handshake protocol) [9] [60] [11].
- IPFS blog on Handshake and Fleek integration [10] [25].
- Spectra domain semantics and project ethos [1] [18].

---

1 2 14 18 53 59 semantic-authority.md
https://github.com/spectra-gallery/arquolab-sandbox-models/blob/0a1860ba765b9483f06ef4f0089f054806ac1037/arketyp/semantic-authority.md

3 4 WEB-01-network.md
https://github.com/spectra-gallery/arquolab-sandbox-models/blob/0a1860ba765b9483f06ef4f0089f054806ac1037/docs/WEB-01-network.md

5 10 12 23 25 29 30 Decentralizing the Internet's Root | IPFS Blog & News
https://blog.ipfs.tech/decentralizing-the-internet-s-root/

6 Understanding ICANN's Draft Applicant Guidebook: What's New for ...
https://www.cscdbs.com/blog/understanding-icanns-draft-applicant-guidebook-whats-new-for-round-two/

7 8 9 15 16 17 19 20 21 39 58 60 Hosting Handshake domains with DNSimple - DNSimple Blog
https://blog.dnsimple.com/2022/04/introducing-handshake-domain-support/

11 13 Namecheap Handshake TLDs - Domains - Namecheap.com
https://www.namecheap.com/support/knowledgebase/article.aspx/10484/2278/namecheap-handshake-tlds/

22 Quickly Deploying dWebsites with Handshake & Fleek - Medium
https://medium.com/blockchannel/quickly-deploying-dwebsites-with-handshake-fleek-84f81650345f

24 Using Fleek IPFS Hosting For Your Handshake HNS Domain Name
https://skyinclude.com/fleek/

26 27 37 01-DNS-arquolab-io-net-lan.conf
https://github.com/spectra-gallery/arquolab-sandbox-models/blob/0a1860ba765b9483f06ef4f0089f054806ac1037/uniphilabs/ark/exosysmu/configs/dnsmasq.d/01-DNS-arquolab-io-net-lan.conf

28 32 33 40 41 02-DHCP-arquolab-io-net-lan.conf
https://github.com/spectra-gallery/arquolab-sandbox-models/blob/0a1860ba765b9483f06ef4f0089f054806ac1037/uniphilabs/ark/exosysmu/configs/dnsmasq.d/02-DHCP-arquolab-io-net-lan.conf

31 34 35 36 03-DHCP-arquolab-io.conf
https://github.com/spectra-gallery/arquolab-sandbox-models/blob/0a1860ba765b9483f06ef4f0089f054806ac1037/uniphilabs/ark/exosysmu/configs/dnsmasq.d/03-DHCP-arquolab-io.conf

38 44 47 48 54 apply-ubuntu.sh
https://github.com/spectra-gallery/arquolab-sandbox-models/blob/0a1860ba765b9483f06ef4f0089f054806ac1037/uniphilabs/ark/exosysmu/scripts/apply-ubuntu.sh

42 43 45 46 apply-fedora.sh
https://github.com/spectra-gallery/arquolab-sandbox-models/blob/0a1860ba765b9483f06ef4f0089f054806ac1037/uniphilabs/ark/exosysmu/scripts/apply-fedora.sh

49 50 51 52 index.ts
https://github.com/spectra-gallery/arquolab-sandbox-models/blob/0a1860ba765b9483f06ef4f0089f054806ac1037/uniphilabs/ark/interwined-cluster/apps/server/src/index.ts

55 56 vrb-validate.sh
https://github.com/spectra-gallery/arquolab-sandbox-models/blob/0a1860ba765b9483f06ef4f0089f054806ac1037/uniphilabs/ark/exosysmu/scripts/vrb-validate.sh

57 wireguard-add-peer.sh
https://github.com/spectra-gallery/arquolab-sandbox-models/blob/0a1860ba765b9483f06ef4f0089f054806ac1037/uniphilabs/ark/exosysmu/scripts/wireguard-add-peer.sh