ChatGPT

Exosystem VRB & Spectra Deployment Bundle

System Setup and Networking Components

The **Virtual Routing Bridge (VRB)** serves as the central network hub connecting the **WAN uplink** (Internet) with the **LAN bridge** for internal devices and VMs. We configure Fedora 42 or Ubuntu Server with NetworkManager profiles to define:

- **WAN (Uplink)** – e.g. a static IPv4/IPv6 on the external interface (e.g. `eno8403`), representing the public gateway [1]. This interface is assigned a static IP (from your ISP or cloud provider) and default route.
- **LAN (Bridge)** – a bridge interface (e.g. `vbr1s0`) with a static IP on the internal subnet (e.g. `10.10.10.1/24`) [1]. The bridge groups the LAN NIC (e.g. `enp7s0f0`) and any virtual taps, allowing VMs or physical LAN devices to share the subnet. The VRB's bridge IP acts as the **default gateway** and DNS for LAN clients.
- **Bridge Port** – the physical LAN interface is enslaved to the bridge (no IP on the NIC itself). A `.nmconnection` profile (e.g. `30-port-enp7s0f0.nmconnection`) ensures the NIC attaches to `vbr1s0` on boot [2].

NetworkManager & Sysctl: On both Fedora and Ubuntu, NetworkManager is used as the renderer for consistency. We enable IP forwarding at the OS level so the VRB can route packets between interfaces. For example, a sysctl config `/etc/sysctl.d/50-enable-forwarding.conf` is installed with `net.ipv4.ip_forward=1` (and IPv6 forwarding if needed) [3]. This ensures the VRB can NAT traffic from LAN/VPN out to the WAN. The VRB's firewall (firewalld on Fedora, iptables on Ubuntu) is configured to **masquerade** outbound traffic on the WAN interface and allow incoming VPN UDP port (e.g. 32237) [4] [5].

dnsmasq DNS/DHCP: We deploy `dnsmasq` (managed via NetworkManager) to provide DNS and DHCP for both the LAN bridge and VPN networks. The VRB acts as an **authoritative DNS server** for the project's internal domains (e.g. `*.arquolab.io` on the LAN) and provides recursive resolution for external queries. For example, a dnsmasq config for the LAN might include:

```
# Authoritative DNS for internal domain
local=/arquolab.io/
domain=arquolab.io,10.10.10.0/24,local  # domain name and subnet
expand-hosts
localise-queries
```

This declares that any hostname under `arquolab.io` will resolve locally for clients on `10.10.10.0/24` [6]. Another config handles DHCP on that subnet, assigning IPs (e.g. `10.10.10.100-.200`) and options. Similarly, the WAN interface can have a dnsmasq config if using a public IP range. In our case, the VRB has a public /27 block; dnsmasq is set *non-authoritative* on `eno8403` to hand out a few public IPs to VMs while leaving upstream DHCP intact [7] [8]. The VRB's dnsmasq also forwards special TLDs: for example, a `server=/.spectra/127.0.0.1#5350` entry routes any `*.spectra` domain queries to a local resolver on port 5350 [9] (useful if `.spectra` is a

Handshake or custom TLD). All dnsmasq configs are placed in `/etc/NetworkManager/dnsmasq.d/` and NetworkManager is told to use dnsmasq for DNS [10].

WireGuard VPN: The bundle includes a WireGuard VPN (`wg0`) for secure remote access. The VRB is the VPN server (e.g. listening on UDP port 32237) with an internal VPN IP (e.g. `10.44.0.1/24`). Peers (developers, cloud nodes, or edge devices) get addresses in an overlay network (e.g. `10.44.99.x`). The VRB's WireGuard config (`/etc/wireguard/wg0.conf`) sets the interface address and brings up the LAN bridge on VPN start (PostUp) [11]. Peers are defined with allowed IPs for their /32. For instance:

```
[Interface]
Address = 10.44.0.1/24
ListenPort = 32237
PrivateKey = <server_private_key>
PostUp = nmcli connection up vbr1s0 || true  # ensure LAN bridge is up
...
[Peer] # Example peer
PublicKey = <peer1_pubkey>
AllowedIPs = 10.44.99.7/32
```

We provide scripts for **peer management and key rotation**. The `wireguard-add-peer.sh` helper generates a new keypair, appends a `[Peer]` entry to the server config, and instantly applies it with `wg set` [12]. It also produces a client config file (`<name>-wg0.conf`) with the correct settings, including the VRB's public key and endpoint, and the allowed routes. Notably, the client config pushes routes for both the VPN subnet and the LAN subnet (e.g. `AllowedIPs = 10.44.0.0/16, 10.10.10.0/24`), so VPN users can reach **both the overlay and internal LAN** networks securely [13]. The client config also sets the DNS to the VRB's LAN IP (10.10.10.1) so that internal hostnames (e.g. `myservice.arquolab.io`) resolve over the VPN [14]. A snippet of the generated client config is below:

```
[Interface]
PrivateKey = <peer_private_key>
Address = 10.44.99.10/32
DNS = 10.10.10.1

[Peer]
PublicKey = <server_pubkey>
Endpoint = <VRB_public_IP>:32237
AllowedIPs = 10.44.0.0/16, 10.10.10.0/24
PersistentKeepalive = 25
```

With WireGuard up, remote nodes can join as if they are on the LAN. For example, after connecting, a VM in the cloud could reach a database at `10.10.10.x`, or SSH into the VRB's LAN IP. The overlay uses a large `10.44.0.0/16` range to encompass various subnets (we segment VPN client IPs by function, e.g. `.99.x` for personal devices) [15]. The VRB itself can also use the VPN to reach remote peers (if needed) since forwarding between `wg0` and `vbr1s0` is enabled by our firewall (we mark `wg0` as internal zone) [5].

Summary: At this stage, the system setup yields a stable networking core: the **VRB** on Fedora/Ubuntu is running NetworkManager with a WAN uplink and a LAN bridge (`10.10.10.1/24`), `dnsmasq` providing DNS and DHCP on LAN (and any other managed networks), and **WireGuard** for remote connectivity. LAN devices get IPs and resolve internal domains via the VRB. Remote users can VPN in and appear on the network. All routing/NAT is handled by the VRB, reducing complexity. This design effectively **reduces entropy** in network configuration by centralizing key services on the VRB – a purposeful approach to impose order on what would otherwise be a chaotic mix of connections [16]. The VRB becomes a single source of truth for addressing and naming, optimizing network flows.

Web Portal & Application Layer

On top of the networking foundation sits the **Spectra web portal and application layer**, which is delivered through Node.js services and a custom front-end. The application layer is composed of multiple **Node.js microservices** (agents and APIs) plus a static web front-end that can function as a **Progressive Web App (PWA)**. Key components include:

- **Node.js Backend Services:** The backend is built on Node (with frameworks like Express or Hapi) to host RESTful APIs, realtime services, and agent daemons. In this project, several **agents** (such as *Uniphi*, *Kobalt*, etc.) run as Node processes to handle specific tasks (coordination, integrity checks, etc.) [17]. For instance, *Kobalt* (the system integrity agent) might expose health metrics, while *Uniphi* coordinates logic. These services form a **mesh of microservices** communicating over the LAN/VPN and updating the system state (e.g. via Redis pub/sub or MongoDB). The Node backend also implements the **domain logic layer** (resolving custom domains/TLDs) and acts as a registry for dynamic data [18] [19]. All APIs are typically containerized for consistency.

- **Static Front-End (PWA):** The user interface is delivered as a static bundle (HTML/CSS/JS) that can be served via any web server or CDN. The front-end is a **custom Progressive Web App** featuring advanced UI (e.g. a 3D dashboard with WebGL/D3 for visualizing the node mesh [20]). The project actually has *two* major front-end apps, reflecting two "personas" of the Spectra system: **Kobalt** and **Phoebii**. **Kobalt**'s front-end focuses on system structure and administration (the "structural / authority" facet), whereas **Phoebii** is a more creative, gallery-like interface for generative art content. In practice, these might be two modes or two related SPAs. For example, `kobalt.io` could host the main dashboard and API console (also used for authentication, e.g. WebAuthn) and `phoebii.art` could host the artistic showcase UI [21]. Both are built as installable PWAs for offline use and fast loading. They employ modern frameworks (the codebase mentions Next.js and Nuxt.js for SSR/SPA generation [22]) to pre-render content and then hydrate on the client, providing a smooth app-like experience. Features like service workers are used for caching; for example, static assets might be cached with a versioned name for long-term caching, while the service worker handles updating new content in the background to reduce network load.

- **Web Server (Apache/Nginx) Deployment:** We support deployment of the web portal on either Apache or Nginx as a reverse proxy serving the Node services and static files. Both approaches are documented:

- **NGINX:** As a high-performance option, Nginx can serve the PWA static files directly and proxy API requests to the Node backend. We configure an Nginx server block for the domain (e.g. `spectra.gallery` or `phoebii.art`). Static files (HTML, CSS, JS, images) are served from a local `dist/` directory or an S3 bucket (if using a CDN). Dynamic requests under `/api/` or specific paths are forwarded to the Node service running (for example) on `localhost:3000`.

We enable **fallback routing** so that any unknown route (especially for client-side routes in the SPA) returns the `index.html` – this lets the PWA handle 404s via its router. An example Nginx config snippet:

```
server {
    listen 80;
    server_name spectra.gallery;
    root /var/www/spectra;  # contains index.html and assets
    location / {
        try_files $uri $uri/ /index.html;
    }
    location /api/ {
        proxy_pass http://127.0.0.1:3000/;
        proxy_http_version 1.1;
        proxy_set_header Host $host;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
    }
    # ... (other settings like caching headers)
}
```

We also set caching headers for static assets (e.g. long `Cache-Control` for hashed files) and use **cache invalidation strategies**. One strategy is to embed version/hash in filenames (so new deploys naturally bypass old cache). Another is to send cache purge requests or use soft versioning (like appending `?v=X` to asset URLs on each release). For truly dynamic content, Nginx can be configured with `proxy_cache` and `proxy_cache_bypass` on deploy signals – but in our bundle, the focus is on frontend assets and API responses (the latter are typically not cached at the proxy, to ensure real-time data). We ensure the **web server reloads** gracefully on deployment (or uses blue-green deployment if containerized) to avoid downtime.

- **Apache:** The bundle includes an Apache virtual host example as well, to cater to environments where Apache is available. Apache (with mods like `proxy_http` and `rewrite`) can achieve similar setup. We use an `<VirtualHost *:80>` for the domain, set `DocumentRoot` to the PWA's `dist/`, and add rewrite rules for SPA routing:

```
DocumentRoot "/var/www/spectra"
<Directory "/var/www/spectra">
  Require all granted
  AllowOverride None
</Directory>
ProxyPassMatch "^/api/(.*)$" "http://127.0.0.1:3000/api/$1"
ProxyPassReverse "/api/" "http://127.0.0.1:3000/api/"
RewriteEngine On
# If the request isn't for a real file or directory, redirect to
index.html
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^ /index.html [L]
```

Apache handles static files and delegates `/api` to Node. We also configure appropriate **Expires headers** (using `mod_expires`) for static assets and possibly use `mod_cache_disk` for any server-side rendered pages if needed. Cache invalidation in Apache deployment might involve an `InvalidateCache` command or simply relying on new file names per release (which is our preferred approach with PWAs).

Both Apache and Nginx approaches aim to provide a robust **reverse proxy** in front of the Node services, enabling SSL termination, domain-based routing, and improved performance (serving static content directly). They also facilitate **zero-downtime reloads** – e.g., we can reload Nginx or use Apache's graceful restart during CI deployment so that new content is served atomically. In development or lighter setups, the Node.js app can also serve static files itself (via Express static middleware), but for production we prefer the web server or CDN for efficiency.

Finally, the web portal is designed with an eye on **offline-first and network optimization**. Because it's a PWA, once loaded it can function with reduced server interaction, caching frequently used data and thereby reducing network entropy (less unnecessary chatter). When online, it uses efficient protocols (HTTPS/2 or HTTP/3) and possibly WebSockets for realtime updates from the agents. In essence, the application layer leverages modern web optimizations to ensure that the network is utilized optimally and that the user experience remains smooth even under less-than-ideal network conditions.

DevOps & CI/CD Pipeline

The project employs a **DevOps pipeline** that automates testing, building, and deployment of both the VRB firmware/configuration and the application layers. We integrate **GitHub Actions** and **CircleCI** to cover different aspects of CI/CD:

- **GitHub Actions Pipelines:** The repository contains workflows that build and deploy the application containers and static assets on push. For example, a **"Cloud Deploy"** workflow runs on every push to the `main` branch [23]. It checks out the code, sets up Node.js, and runs tests/builds:

- Build steps – We install dependencies (`npm ci`) and run the front-end build (e.g. compiling the Next.js app to static files) [24]. For instance, the workflow navigates into `stations/next-visualizer` and runs `npm run build` to produce an optimized static bundle [25]. In parallel, any firmware build scripts (e.g. building a custom OS image or configuration archive for the VRB) could be invoked here, ensuring that **infrastructure code (NetworkManager profiles, etc.) is tested and packaged**.

- Containerization – The workflow then builds a Docker image for the application services [26]. This image might include the Node.js backend and the static front-end (for example, serving the static files via an Express app or bundling them in a lightweight HTTP server). We tag the image (often using `latest` or the commit SHA). If the VRB firmware needs containerization (for instance, building a custom router OS image), that too can be handled via Docker or specialized tools (like Packer). Our GitHub Actions config uses artifacts to pass the built image to subsequent jobs [27].

- Multi-platform Deploy – After building, separate jobs deploy to different targets. For example, one job deploys to **Google Cloud Run** and one to **Azure Web App** (as mentioned in the workflow) [28] [29]. The Google Cloud deploy job downloads the image artifact, pushes it to Google Container Registry, and then triggers a Cloud Run deployment [30] [31]. The Azure job

similarly pushes to Azure Container Registry and deploys to an Azure Web App container instance [32] [33]. These demonstrate branch-based or environment-based triggers: e.g., pushing to `main` auto-deploys to production cloud, whereas pushing to a `dev` branch might deploy to a staging environment (configurable via branch filters in the `on:` clause). Secure credentials (GCP service accounts, Azure registry logins, etc.) are managed via **GitHub Secrets**, so no keys are exposed in the repo [34] [35]. This pipeline ensures that any commit to the main codebase is quickly tested and propagated to the live environment in an automated, consistent manner.

- Static content deployment – In addition to container-based deploys, we have workflows or scripts for static content. For example, a separate job or Action might upload the static PWA files to a CDN or FTP host. Indeed, the repository's `web-01` outpost includes a script to upload built files via FTP [36]. We integrate this in CI (as hinted by a `static-ftp.yml` workflow [37]), so that updating a marketing site or documentation portal can be as simple as pushing a new commit.

- **CircleCI Pipelines:** While GitHub Actions covers a lot, CircleCI is also configured (in our scenario) for certain tasks – often for building **firmware and low-level components** in a controlled environment, or running more complex integration tests. For instance, we might have a CircleCI config that builds the VRB's custom image (if using an immutable infrastructure approach). This could involve:

- **Firmware Build Automation:** Using a CircleCI job to assemble a bootable image or run Ansible scripts. Because the VRB setup (NetworkManager profiles, sysctl, etc.) is scriptable (as seen in `apply-fedora.sh` and `apply-ubuntu.sh`), we could create a Docker image that mimics a fresh OS, run the apply script, then snapshot the filesystem as an image artifact. This artifact could be published (for example, an AMI in AWS or a QCOW2 for OpenStack) for deploying new VRB instances. The pipeline would ensure any change to those configs is tested (possibly by booting it in a VM in CI and running a quick health check).

- **Testing & Linting:** Both CI systems run unit and integration tests for the Node services (`npm test`) [38] and any front-end tests. They also run linters, and security scans. A test stage might instantiate a minimal WireGuard peer (using `wg` in a container) to verify connectivity, or spin up the Node services in Docker Compose to run API integration tests on localhost.

- **Branch-Based Deployment Triggers:** We use git branch conventions to manage environments. The CI/CD pipeline is set so that commits on development branches trigger test builds, but only certain branches trigger deployment. For example, `main` -> production, `develop` -> a staging server, etc. In the GitHub Actions example, the workflow is triggered on push to `main` by design [39]. In CircleCI's config (e.g., `.circleci/config.yml` if present), we might use filters or contexts: e.g., jobs will only deploy when `CIRCLE_BRANCH == 'release'` or when a tag is pushed. This prevents unintended deployments and allows feature branches to be tested in isolation.

- **Security & Secrets:** Both CI systems emphasize secure handling of keys and secrets. We never store sensitive credentials in the code. Instead, we rely on:

- **GitHub Secrets / CircleCI Contexts:** API keys, SSH keys (for server access), cloud credentials, etc., are stored in the CI platform's secret store. For example, the Cloud Run deploy uses `${{ secrets.GCP_SERVICE_KEY }}` and other secrets for authentication [40]. Similarly, an Infomaniak API token (for DNS updates) or Let's Encrypt credentials could be stored and referenced during deployment to automate certificate issuance.

- **Docker Secrets:** If our deployment uses Docker Swarm or similar, runtime secrets (like database passwords) are injected via Docker secrets or environment variables in the cloud environment, not baked into images. The build pipeline might pull these from a vault if needed for testing, but generally we use dummy placeholders in tests and real values in production via the orchestration platform.

- **Continuous Delivery of Web & Firmware:** The outcome is that any change to the **web portal** (UI or backend) triggers an automated build of new Docker images and static files, runs tests, and if successful, deploys to the hosting environment. Likewise, changes to the **infrastructure code** (network configs, etc.) can trigger rebuilding a VRB image or applying changes via infrastructure-as-code (for instance, using GitOps: the VRB's WatcherAgent could poll a repo for DNS zone changes [41]). In this project, the WatcherAgent bridges GitOps, meaning if, say, we update a DNS zone file in the repo, agents will synchronize it across the mesh [41]. This is facilitated by CI ensuring the changes are valid and then perhaps committing to a config repo that VRB nodes pull from.

By combining GitHub Actions and CircleCI, we cover both **application deployment** and **infrastructure deployment**. The philosophy is to automate everything: building the **firmware** (or base system configuration) and the **application stack** with minimal manual steps. Each pipeline's success reduces entropy in the system – by consistently enforcing tests and standards, we avoid configuration drift or "unknown states." The result is a highly reproducible deployment process where every component, from a DNS config to a UI bundle, is built and released in a controlled, traceable manner.

Distributed Redundancy & Quorum Architecture

For high availability and resilience, the system is designed as a **distributed meta-cluster** of nodes that share responsibilities and maintain quorum. This means multiple VRB nodes and service instances can run in tandem, tolerating failures without downtime:

- **Meta-Cluster of Fedora Nodes:** We deploy the VRB stack on more than one node (for example, three Fedora Server nodes forming a cluster). Each node has the same VRB configuration (bridge, dnsmasq, WireGuard, etc.), but only one may act as the active gateway at a time. We use a **quorum-based redundancy** approach: decisions like "who is primary router" or "which service instance is leader" are determined by consensus among nodes. For instance, we might run a lightweight **etcd or Consul** cluster across the 3 nodes, which stores the cluster state and uses Raft consensus – requiring a majority (quorum) for decisions. This way, if one node goes down, the remaining two can still agree on state and continue serving.

- **High Availability (HA) Failover:** At the network level, we implement failover using something like **VRRP (Virtual Router Redundancy Protocol)**. A tool such as **Keepalived** can run on the VRB nodes to manage a **virtual IP** address for the gateway. The virtual IP (for example, the address used by `ecosys.mu` internally or the floating public IP) will be bound to whichever node is elected master. If that node fails health checks, another node automatically takes over the IP. Keepalived uses heartbeats over the LAN – if the primary VRB doesn't advertise within a threshold, a backup node assumes the virtual IP. This provides seamless failover at the IP level: LAN devices and VPN clients still reach "the VRB" at the same IP, unaware of the node change. The health checks for failover can be as simple as ping/VRRP or as advanced as script checks (e.g., ensuring dnsmasq and other services are running on the primary). This ensures **no single point of failure** for network routing.

- **Service Discovery and Load Balancing:** For the application layer, we need to ensure multiple instances can work together. We incorporate **service discovery** so that microservices find each other across nodes. This could be achieved via Consul DNS or a shared etcd registry. For example, if Node agent *Uniphi* runs on all 3 nodes but *WatcherAgent* runs only on one, services can register their presence and clients (or proxies) discover an available instance via DNS (`agent.service.consul`) or via a built-in coordination. We also deploy a **load balancer or API gateway** at the edge of the cluster. This could be an Nginx instance running on each node (with keepalived ensuring only the master's Nginx handles external traffic), or a cloud load balancer in front of all nodes. The API gateway routes incoming requests to the appropriate service instance (possibly using round-robin or more intelligent routing). For static content, we leverage a **CDN or edge cache** so that user requests might be served by a nearby edge node rather than hitting the origin cluster every time [42]. For instance, using a Content Delivery Network (like Cloudflare or AWS CloudFront) in front of `spectra.gallery` can cache images and static files globally, reducing latency and load on the origin. Dynamic API calls still reach the cluster, but those are load-balanced.

- **Quorum and Consistency:** With a quorum design, certain actions require consensus. For example, consider the internal DNS zone updates (the project's "self-sovereign namespace" logic [18]) – any change (like registering a new `.phi` domain or updating a record) could be propagated only after at least 2 of 3 nodes validate it. Agents like *ValidationAgent* and *SyncAgent* ensure consistency: *ValidationAgent* runs on each node to perform safety checks on data (including monitoring entropy or anomalies) [43], and *SyncAgent* distributes zone and state changes peer-to-peer. If one node has a new piece of data, it shares it and others confirm. This quorum approach prevents split-brain scenarios and helps **reduce entropy in the network's knowledge** – the system actively converges on a single agreed state across nodes, rather than diverging.

- **Health Checks and Self-Healing:** Each node runs health check routines. These can be system-level (CPU, memory, service processes) and application-level (e.g., is the Node API responding, is the database reachable?). If a service on one node fails, a monitoring agent (or external orchestrator like Kubernetes, if we used it) would restart it or route around it. If an entire node fails, as discussed, failover takes place for its responsibilities. The cluster may also be configured to automatically **re-provision** a node in some cases (for example, using Ansible or cloud-init scripts triggered by a monitoring system). However, given this is not a full Kubernetes setup but a more bespoke HA cluster, much of the self-healing is handled by simplified means like systemd service restarts and the aforementioned keepalived for node failover.

- **Edge Deployment:** The project also considers **public API gateway / CDN edge deployment**. This means parts of the system are deployed on the cloud edge to accelerate global access. For instance, we might deploy read-only instances of certain microservices or caches on a CDN or serverless platforms near users. The GitHub Actions pipeline shows deployments to Cloud Run and Azure; these could represent edge instances of our services, acting as a globally distributed layer [31] [44]. Static assets definitely go to the CDN, but even dynamic interactions can be sped up via geo DNS or anycast IPs pointing to the nearest active cluster node. The **Spectra Gallery** being an art/content platform benefits from fast content delivery, so images and media are likely served from an S3 + CloudFront or a similar setup, with the origin cluster as backup.

In summary, the distributed design ensures **high availability** (through multi-node redundancy and automatic failover) and **scalability** (through load balancing and optional cloud edge offloading). It embodies the project's theme of **entropy reduction** by actively synchronizing state (preventing config

drift or split data) and applying consensus – a network optimized for reliability and consistency. Any single node's chaos (failure) is absorbed by the order imposed by the quorum rules of the collective.

Domain Authority and Certification

The project spans multiple domains under the Spectra ecosystem, each chosen for semantic significance and configured for proper DNS resolution and certification. We establish a clear **domain authority structure** with mappings to our infrastructure:

- **Domain Mapping (Spectra Ecosystem):** The domains in use reflect different facets of the project, following a semantic layering [45] [46]. For example:
- **ecosys.mu** – Mapped to the core internal ecosystem. The name "ecosys" (ecosystem) with ".mu" (the Greek letter μ) symbolizes a microcosm experimental environment [47]. We use `ecosys.mu` as an internal domain (and hostname) for the VRB cluster itself. For instance, the VRB's LAN IP (10.10.10.1) might resolve to `core.ecosys.mu`, and the VRB's WireGuard endpoint might be `vpn.ecosys.mu` (resolving to the static public IP). This domain is primarily used inside the network and via VPN, representing the **micro-scale lab environment**.
- **exosys.xyz** – Points to external-facing services of the exosystem. "Exosys" suggests an external system, and the ".xyz" TLD implies openness to any coordinate or experiment [48]. We use `exosys.xyz` as a public entry point to the network. For example, the primary public API or portal might be accessible at `exosys.xyz`. This could be an A record mapping to the VRB's public IP (so that hitting `exosys.xyz` in a browser goes to the VRB or load balancer). The choice of .xyz also underlines the project's experimental nature. In practice, `exosys.xyz` could direct to the same server as ecosys.mu but via the WAN interface – useful for testing external connectivity and for services intended for external users.
- **arquolab.io** – The main lab domain for the project framework [49]. "Arquo Lab" hints at an architecture laboratory, and .io (commonly used in tech) reinforces input/output and tech focus. We assign `arquolab.io` as the domain for the web portal (UI and API) in production. For instance, the PWA might actually live at `gallery.arquolab.io` or `app.arquolab.io`, and internal services might use subdomains of arquolab.io. In our dnsmasq configs, we set `arquolab.io` as a local domain on the LAN [6] so that hostnames under it (e.g. `db.arquolab.io`) resolve to internal IPs. Public DNS for arquolab.io will have records pointing to the cluster's public endpoints (with possibly different subdomains for different services or environments).
- **spectra.gallery** – The flagship domain and namesake of the project [50]. We use this as the primary public web front-end for showcasing the spectrum of art and data. It's likely aliased or CNAME'd to something like `gallery.arquolab.io`, or vice versa. The `.gallery` TLD makes its purpose clear: it's the front-facing art exhibit and interactive portal. We ensure that `spectra.gallery` is served via our web servers (Apache/Nginx as discussed) with a friendly URL for users. This domain will be configured in DNS to point to the CDN or load balancer that fronts the cluster. In the internal hierarchy, spectra.gallery is the **public content layer**, while the underlying APIs might still use `.io` or `.xyz` domains.

Additionally, the project references other domains like **kobalt.io** and **phoebii.art** for specific roles [51] [21]. For instance, `kobalt.io` could be used as an official API root and documentation site (public-facing, representing the structural core), and `phoebii.art` as a showcase for generative art (creative front-end). In fact, the team explicitly planned that *kobalt.io* be the WebAuthn origin and API root, and *phoebii.art* host the creative front-end [21]. We have reserved these names and mapped them appropriately: DNS A records for `kobalt.io` and `phoebii.art` both resolve to our infrastructure, but in the application, each domain may serve a different front-end or content set (ensuring a clean

separation of concerns and audience). This multi-domain setup follows a **"semantic stratum"** strategy – each domain's TLD hints at its purpose (e.g. `.io` for tech, `.art` for creative, `.org` for authority) [45]. Such a structured yet poetic domain constellation matches the Spectra Gallery's narrative, using domain names to mirror the system's architecture and the golden-ratio metaphor of **$x^2 - x - 1$ layers** [52] [53].

- **DNS Configuration:** All these domains need proper DNS records. For internal usage, the VRB's dnsmasq handles `.io` and custom TLDs. For public DNS (e.g. managed via Infomaniak or another registrar's interface), we create the necessary A/AAAA records:
- `arquolab.io` – point to the static public IP of the VRB (or the load balancer IP). If using IPv6, add AAAA record to the VRB's v6. In our config snippet, we saw an example public subnet (144.2.68.224/27) with the VRB's IP .227 [54]. We'd set `arquolab.io A 144.2.68.227`.
- `spectra.gallery` – CNAME to `arquolab.io` or another A record to the same IP (depending on whether we serve it from the same server). It could also point to a CDN. For instance, if using CloudFront, `spectra.gallery` CNAME to the CloudFront distribution domain, and origin is our server.
- `exosys.xyz` – also an A record to the same IP, unless we split it to a different server. But likely it's the same cluster, so it's just an alias domain for external access.
- `ecosys.mu` – if we want this accessible externally (perhaps not, if it's mainly internal), we could still publish an A record for completeness. However, we might limit it to internal resolution only (only our dnsmasq knows it).
- Subdomains (like `api.kobalt.io`, `cdn.arquolab.io`, etc.) – configured as needed to point to specific targets (API gateway, storage bucket, etc.).

We also ensure **reverse DNS** for the IP if needed (especially if running mail services or just cleanliness – e.g., PTR for 144.2.68.227 to something like `vrb.arquolab.io`).

- **Certification Authority & HTTPS:** All public domains will use HTTPS. We have two main approaches:
- **Let's Encrypt (ACME) Integration:** For publicly reachable domains (`*.io`, `*.xyz`, `*.gallery`, `*.art`), we use Let's Encrypt certificates. The deployment bundle can include an ACME client (certbot or acme.sh) and either run it directly on the VRB or via a CI step. Since Infomaniak is the registrar for some of these domains, we might leverage their API for DNS-01 challenges to get wildcards. For example, using the Infomaniak API token (stored as a secret) with an ACME client to create a `_acme-challenge.spectra.gallery` TXT record, thus obtaining a wildcard cert for `*.spectra.gallery` without downtime. This process can be automated in our GitHub Actions pipeline (a step to request/renew certs and deploy them to the server or container). Alternatively, if the servers themselves run ACME clients, we open port 80 for HTTP-01 or use DNS-01 for wildcard. We consider a **staging environment** for ACME on first runs to avoid rate limits, then switch to production CA. The bundle's documentation includes instructions for linking Let's Encrypt: e.g., install certbot, and a systemd timer for renewal.
- **Private CA for Internal Domains:** For domains that are internal (like perhaps `ecosys.mu` if used only internally, or any `.spectra` handshake domain if not recognized by public CAs), we set up a **self-signed Certificate Authority** within Spectra. This could be done by generating a root CA certificate (perhaps branded as Spectra Authority) and distributing its trust to clients (VPN clients can be given the CA cert to install). Then we issue our own certificates for internal services (like an `ecosys.mu` wildcard or even device certificates for IoT nodes). The semantic-authority docs suggest Kobalt's role as an open-standard steward / CA root [55] – indeed *kobalt.org* or *kobalt.cert* might be theoretical CA domains. But pragmatically, we can use our self-signed CA to secure `.mu` or internal hostnames. This ensures even internal web interfaces are HTTPS (though you must trust the CA in your browser or device).

In practice, we likely implement both: Let's Encrypt for anything public (with auto-renewal), and a mini internal PKI for anything that LE can't cover. If Infomaniak (our DNS host) offers an integrated solution (some registrars provide free LE certs or their own CA), we could also use that for convenience – e.g., Infomaniak might allow one-click issue of a wildcard certificate that we can download and install.

- **Static IP and PTR setup:** We have a static IP (or IPs) for the VRB cluster as mentioned. We coordinate with the VPS provider or ISP to ensure these IPs are assigned to our servers (Fedora/Ubuntu picks it up via the NM profile). For multiple nodes, either each has its own IP and DNS (and we use round-robin DNS or a load balancer that points to all), or we use a floating IP that can move between nodes. Many cloud providers allow an IP to be moved to a backup server on failover. If on bare-metal, that's where VRRP (Keepalived) is essential as described. We document how to update DNS records when IPs change (though ideally static means it won't, but if migrating between hosts, a DNS update might be needed). It's good to lower DNS TTL during transitions.

All these domain configurations are tied back to the **Spectra Gallery's semantic clustering** of names. Our documentation explicitly references how each domain corresponds to a layer or component of the architecture, reinforcing a human-meaningful structure. For example, we note that: - *"kobalt.io → WebAuthn origin + API root; phoebii.art → creative front-end; umowt.io → CI/CD daemon; umowt.org → open spec and root certs"* [53] .

This mapping shows a deliberate design: the domains themselves communicate the role of each endpoint (growth/public layer, middleware, authority layer) [45] . By following this scheme, we reduce cognitive entropy – there's an inherent logic to the domain names which mirrors the system's architecture, making management easier. The **Spectra domain authority structure** is essentially a framework where every service is given a domain that fits into a narrative hierarchy (public interface, bridging services, authority/governance) [52] . This not only benefits technical organization but also branding and clarity to users.

In summary, our bundle covers obtaining and configuring all relevant domains, setting up DNS records to map them to the infrastructure, and securing them with appropriate certificates. The end result is that whether a user visits `spectra.gallery` or an API calls `api.kobalt.io`, they get a secure, trusted connection to the Spectra ecosystem. The combination of an internal CA (for closed networks) and Let's Encrypt/Infomaniak (for public) ensures trust is established without manual cert management overhead.

SSH Configuration and Documentation

For operators and developers, convenient and secure SSH access to the various hosts is set up via a structured `.ssh/config`. We provide a template that defines host aliases for the Exosystem environment:

- **Host Aliases:** The template includes entries like `exosystem`, `exo-lan`, and `exo-public`. These correspond to the different network pathways to reach the VRB or cluster:
- `exosystem` – the primary alias for the system. In practice this could be a smart alias that tries the best route available. For example, it might point to the **WireGuard VPN address** of the VRB (say 10.44.0.1) since that's reachable when you're on the VPN. This would be used by developers who are connected via VPN to manage the system. If VPN is active, `ssh exosystem` logs you into the VRB directly over the secure tunnel.

- `exo-lan` – an alias for the LAN access. This typically maps to the VRB's LAN IP (10.10.10.1). It's used when you are on-site or connected to the LAN (e.g. your laptop is plugged into the VRB's LAN switch or on the same Wi-Fi). It ensures you reach the device over the internal network. For example:

```
Host exo-lan
    HostName 10.10.10.1
    User spectAdmin
    IdentityFile ~/.ssh/id_rsa_exosystem
```

(Using the appropriate user and key.)

- `exo-public` – an alias for public internet access to the VRB. This would use the VRB's public DNS (or IP) and typically is only used if VPN isn't up. For instance:

```
Host exo-public
    HostName exosys.xyz
    User spectAdmin
    Port 22
    IdentityFile ~/.ssh/id_rsa_exosystem
```

Here `exosys.xyz` would resolve to the public IP. We might also specify additional options like `ProxyJump` if direct SSH is disabled and only VPN or another bastion is allowed. (E.g., we could force SSH to go through a bastion host, but in our case the VRB itself is the entry point.)

- **SSH Key Management:** The bundle recommends using an SSH key pair (ed25519 or RSA) for authentication, with the public key added to the VRB nodes' `authorized_keys`. The `.ssh/config` entries ensure the correct IdentityFile is used. We encourage creating a dedicated key for this infrastructure (for auditing). The config can also set `IdentitiesOnly yes` and maybe restrict which keys to try. Because our VRB might be running Fedora Core or similar, the default user could be `fedora` or a created user (we indicated `spectAdmin` in examples). Documentation specifies the username if not default.

- **Host Verification:** Since some domains (like those on .mu or .xyz) will have valid certificates via Let's Encrypt, we can also consider using SSH certificates or at least be mindful of host key verification. We include the initial host keys fingerprint in documentation so users can avoid man-in-the-middle issues on first connect. If the internal CA is in play, we might even sign SSH host certificates with it for an extra layer (this is advanced, optional).

- **Usage and Troubleshooting:** The documentation section for SSH guides the user:

- Copy the provided template to `~/.ssh/config`.
- Replace any placeholder values (if we left `<USER>` or paths).
- Ensure your SSH agent is running and the key is loaded.
- To connect, use `ssh exosystem` (for general use when VPN is connected), or `ssh exo-public` (if outside without VPN), etc. The idea is to abstract the actual IP/hostname – the operator can just think "I need to get into the exosystem" and the config chooses the best route.

The `.ssh/config` also helps with **tunneling and testing**. For instance, if we need to test a service that is only open on the LAN, we can use SSH's port forwarding via these aliases. E.g., `ssh -L 8080:10.10.10.100:80 exosystem` could allow us to view a dev web service on an internal VM. The config can include convenience tunnels or a `ProxyJump` through the VRB for reaching deeper internal nodes (if our cluster had separate internal IPs not directly reachable, though in our design all nodes are on VPN and LAN so they are reachable).

- **Manual Verification Steps:** We provide a checklist of manual steps for verifying the setup after deployment:
- **Network Bridge Test:** Connect a laptop or another device to the LAN port (or to the LAN Wi-Fi if applicable). Ensure it obtains an IP via DHCP in the 10.10.10.0/24 range. Ping the VRB's LAN IP (`10.10.10.1`). Then ping an external site (e.g. `8.8.8.8` or `google.com`) to confirm that the VRB's NAT is functioning and DNS queries are being resolved by the VRB. We expect internet access to work for LAN clients. If not, check firewall masquerade and forwarding settings.
- **DNS Resolution Test:** From a LAN client (or via SSH on the VRB), test that `arquolab.io` resolves to 10.10.10.1 internally [6]. Also test an external resolution: e.g. from outside, `arquolab.io` should resolve to the public IP. If a custom TLD like `.spectra` is used, test that on the VRB (e.g. `dig test.spectra @127.0.0.1#5350`). Likewise, test `ecosys.mu` and `exosys.xyz` – internally and externally if applicable. This ensures our dnsmasq and external DNS configs are correct.
- **VPN Connectivity Test:** From an external machine, generate a WireGuard peer config (using the script or manual wg tool). Bring up the tunnel and verify you can ping the VRB's VPN IP (e.g. 10.44.0.1). Then try to ping a LAN address through the VPN (e.g. 10.10.10.1 or another device). Also test SSH: `ssh exosystem` (which goes over VPN) should log you in. This confirms the WireGuard and routing rules (AllowedIPs, etc.) are correct. If ping fails to LAN, check that the VRB's firewall internal zone includes wg0 (as we did) [56] and that AllowedIPs on client include the LAN subnet.
- **Web Portal Test:** Access the web UI via a browser. Test both HTTP and HTTPS. E.g., go to `https://spectra.gallery` – it should load the PWA (if DNS is propagated and certificate is installed). If using a self-signed cert for internal domain, test that as well (with the CA installed in the browser to avoid warnings). Navigate through the app, ensure the API calls succeed (the Node backend is reachable). Possibly test offline mode (install the PWA, turn off network, see if content is available).
- **Failover Simulation:** If you have a multi-node cluster setup, test failover. For example, if using keepalived, shut down the primary node's keepalived or unplug its network – within a few seconds, the secondary should take over the virtual IP. You should still be able to ping the gateway IP and access services (maybe a brief dropout during transition). Also test that if the primary comes back, either it stays secondary or properly reclaims primary (depending on priority settings). Similarly, stop one instance of a service and see if the load balancer still finds the others.
- **CI/CD Dry Run:** Push a benign change (like edit a README) to a test branch to ensure the GitHub Actions pipeline triggers and passes. This tests that all secrets are set and the workflows are not erroring. In a staging environment, you might do a full deploy from CI to validate the process end-to-end.

All these steps are documented so that after deployment, an engineer can systematically verify each part of the bundle. The documentation encourages using the `.ssh/config` shortcuts for all SSH interactions (to avoid mistakes like hitting the wrong IP or using the wrong key). It also notes common troubleshooting tips: e.g., *"If you can SSH via* `exo-public` *but not* `exosystem`*, ensure your WireGuard is up and the VPN IP is correct"* or *"If LAN devices get IP but no internet, check that IP forwarding is enabled (use* `sysctl -q net.ipv4.ip_forward` *on VRB)"*.

By following the guide and using the provided config templates, admins can **confidently manage the system**. They spend less time figuring out addresses or commands and more time on actual development, which aligns with our theme of reducing friction (entropy) in system operations.

System Architecture Map

To tie everything together, below is an overview of the system's architecture layers and how data flows from the end-user to the core services:

Physical Network & Access: At the base is the physical layer: - **WAN Uplink:** A static internet connection (fiber or VPS network) providing a public IP (e.g. `144.2.68.227`). This connects to the VRB and carries incoming/outgoing traffic to the internet.
- **LAN Network:** A private network (10.10.10.0/24) for internal communications. This might include physical IoT devices, user laptops on local Wi-Fi, and virtual machines on the host. The LAN is bridged through the VRB for Layer-2 connectivity among VMs and the host NIC [1].
- **WireGuard VPN:** An overlay network (10.44.0.0/16) that remote nodes use to join the LAN virtually [15]. It traverses the WAN (UDP traffic on the public IP) but puts clients logically inside the LAN with addresses like 10.44.99.x.

Virtual Routing Bridge (VRB): The VRB (one or multiple nodes) sits at the intersection of these networks: - It hosts the **bridge interface (br-ext)** that has an IP on the LAN (10.10.10.1) and possibly also holds the public IP (if bridging to VMs that use it, or via alias) [57]. - It runs **dnsmasq** for DNS and DHCP on LAN and any other networks (serving `.arquolab.io` internally, etc.) [6]. - It terminates **WireGuard VPN** on interface wg0 (10.44.0.1) allowing VPN peers to route into LAN [13]. - It performs **NAT** for LAN and VPN clients to access the internet via the WAN. - It also runs a local **firewall** segregating zones: WAN (untrusted, only allow necessary ports like 22, 443, 32237) vs internal (LAN, VPN) [5]. - Optionally, VRB nodes run **Keepalived** to manage a virtual IP if multiple VRBs (ensuring one active gateway).

From a **network flow perspective**: when a user from the internet accesses `spectra.gallery`, the DNS resolves to our public IP. The request hits the VRB's WAN, gets forwarded to the web service (which might be on the VRB itself or a VM on the bridge). When a developer SSHes to `exosys.xyz`, it lands on the VRB (or possibly on a jump host container on it). When a remote sensor device connects via WireGuard, its traffic is decrypted on VRB and goes into the bridge to reach, say, a database server on 10.10.10.5.

Service & Application Layer: Above the network, we have the services: - **Reverse Proxy / Web Server:** Running on the VRB (or on each cluster node), this is Nginx or Apache as described. It listens on ports 80/443 on the host. It serves static files (PWA assets) and proxies API requests to backend services. It also handles TLS encryption using the certificates we set up. - **Node.js Microservices:** The backend services can run directly on the VRB nodes or on separate VM/containers (connected to the bridge or launched via Docker Compose [58]). Key services: - *Uniphi* (logic coordinator), *Kobalt* (integrity & mesh binding), *Phil Mo-Tsu* (philosophical consensus engine), *ScribeAgent* (logging and zone tracking), *SyncAgent* (P2P sync), *ValidationAgent* (safety/entropy checks) [17], etc. Each may expose an API or interact internally. - A **database** (MongoDB for state, Redis for pub/sub) also lives here [59], typically on the VRB or on a dedicated VM. They are accessed via the bridge network by the Node services. - These services collectively form the **Spectra Backend**, which the PWA communicates with (e.g., via REST or WebSockets). They also handle internal tasks (like syncing domain records or running AI models for generative art).

Data & Storage Layer: The project also integrates **decentralized storage and identity** components: - **IPFS (InterPlanetary File System):** Used for storing content (like large media or generative art assets) and ensuring they can be distributed. An IPFS node might run on the VRB or a nearby machine, pinning the project's assets [42]. If a piece of content is requested, the PWA might retrieve it via an IPFS gateway or directly if the node is exposed. - **ENS/HNS (Ethereum Name Service / Handshake):** These are used for naming integration [42]. For example, the project might register a Handshake TLD like `.spectra` and resolve it internally (as we saw in dnsmasq config for `.spectra` forwarding [9]). ENS could be used for identity of users or assets (tying into wallets or NFTs for art). - **Persistent Storage:** Traditional databases (Mongo, and possibly InfluxDB or others for telemetry) hold the state that must be consistent cluster-wide. If running in a cluster, MongoDB might be a replica set across the nodes (which itself uses a quorum for consistency). Alternatively, a lightweight distributed DB or even Git as a backing store (with GitOps) could be used for certain config (like zone files).

Client Layer: Finally, the end-users and clients interact at the top: - **Web Clients (Browsers/PWA):** A user with a browser or the installed Spectra PWA accesses the system via the domain URLs. The PWA provides a HUD, canvas, and controls to visualize the mesh, view generative art, and perform actions [60]. It communicates with the backend via secure APIs. Thanks to service workers, it can show cached content instantly and fetch updates behind the scenes. - **API Clients:** Could be other services or command-line tools (the project may have a CLI tool `spectra-cli` for maintenance [61]). They use the public API (with auth) to query or command the system (e.g., adding a new agent or uploading a new artwork dataset). - **Internal Dashboards:** Possibly, there are admin dashboards (maybe Kobalt's interface) available only over VPN or on LAN, giving deeper control (logs, system metrics). These would be accessed via `ecosys.mu` or a secured port. - **Edge Consumers:** In the decentralized spirit, other nodes (even user-run nodes) might connect. For example, if community members run their own node that joins the Spectra network (via WireGuard and authentication), they might contribute compute or storage and appear in the dashboard mesh (subject to consensus rules).

All these layers are tied together with a design philosophy: **minimize entropy, maximize clarity and performance**. By using a structured domain scheme, automated deployments, and robust networking, the system maintains order. The *ValidationAgent* continuously monitors "entropy metrics" – effectively keeping an eye on any disorder in the system's state [43] – and the architecture itself (with feedback loops, health checks) strives to self-correct, much like a thermostat. In metaphorical terms, the Spectra architecture "blends thermodynamic, informatic, and philosophical systems in a continuous feedback structure" [16]. In practice, this means it's always balancing loads, synchronizing data, and aligning with its semantic design.

Below is a summary table of the key layers and components for clarity:

Layer	Components & Domains	Function
Physical Network	WAN uplink (ISP, static IP) – exosys.xyz LAN bridge (10.10.10.0/24) – ecosys.mu WireGuard VPN (10.44.0.0/16)	Provides connectivity. WAN connects to Internet; LAN for internal communication; VPN for remote secure access into LAN.
VRB (Network Core)	Fedora/Ubuntu VRB nodes (vbr1s0 bridge, dnsmasq, wg0, firewalld/ iptables) Virtual IP via Keepalived (if HA)	Routes and manages network traffic. DHCP/DNS for *.arquolab.io LAN zone [6]. Terminates VPN. NATs LAN/VPN to WAN. Maintains network services even if one node fails (in HA setup).

Layer	Components & Domains	Function
Proxy & Gateway	Nginx/Apache web server – spectra.gallery, kobalt.io, phoebii.art	Terminates HTTPS and serves static content. Routes requests to appropriate backend services (reverse proxy). Implements caching and routing rules (e.g. SPA fallback).
Application Services	Node.js microservices (Uniphi, Kobalt, etc.) – api.kobalt.io Databases (MongoDB, Redis) – db.arquolab.io	Implements business logic and data storage. Agents coordinate domain logic, mesh state, and content generation [17]. The database layer ensures persistent state across restarts (and possibly across cluster via replication).
Decentralized Layer	IPFS nodes, ENS/HNS integration, CDN edges (Cloud Run, etc.)	Distributed storage and naming. IPFS provides content delivery; ENS/HNS link blockchain identities or alt domains to content. CDN/edge deployments bring content closer to users globally [31].
Client & Access	PWA front-ends (Spectra Gallery UI) – spectra.gallery, phoebii.art CLI tools (spectra-cli) and external APIs consumers	User interaction layer. The PWA gives real-time visualization (HUD, 3D canvas) and control, working offline when needed. CLI and external clients interact via the API for automation or integration. Users authenticate and then explore the "spectrum" of data and art through this layer.

Each part of this bundle – from networking scripts to CI pipelines – is designed to work in concert. By following this comprehensive setup, the **Exosystem VRB & Spectra Portal** operates as a coherent, secure, and efficient whole. It not only meets technical requirements (connectivity, deployment, redundancy) but also reflects the project's conceptual underpinnings (entropy reduction, semantic coherence, and the unification of diverse elements under a singular architecture).

With this deployment bundle, maintainers can launch a Fedora 42 or Ubuntu server and transform it into the nerve center of a decentralized Spectra Gallery node, confident that all pieces from low-level network config to high-level domain strategy are in place and documented. **(Sources: Configuration scripts and docs from Spectra repository [2] [13] [21])**

---

[1] [15] [57] GitHub
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/docs/uniphi/xbox-p2p-cloud-gaming.md

[2] [3] [5] [10] [56] GitHub
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/uniphilabs/ark/exosysmu/scripts/apply-fedora.sh

[4] GitHub
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/uniphilabs/ark/exosysmu/scripts/apply-ubuntu.sh

6 7 8 54 GitHub
https://github.com/Exosysh-Lob-Ueht/spectra-ecosystem/blob/caf6855f5a2469311ba38d46a5e85eea786ded11/configs/dnsmasq.d/03-DHCP-arquolab-io.conf

9 GitHub
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/uniphi/domain-extension-bundle-1/configs/30-DNS-spectra.conf

11 GitHub
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/uniphilabs/ark/exosysmu/configs/wireguard/wg0.conf

12 13 14 GitHub
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/uniphilabs/ark/exosysmu/scripts/wireguard-add-peer.sh

16 17 18 19 20 41 42 43 58 59 60 61 GitHub
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/docs/Spectra_Architecture_Overview.md

21 45 46 51 52 53 55 GitHub
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/afloat-buffer/phid.md

22 README.md
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/outposts/web-01/README.md

23 24 25 26 27 28 29 30 31 32 33 34 35 39 40 44 cloud.yml
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/.github/workflows/cloud.yml

36 37 workflow.md
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/outposts/web-01/workflow.md

38 GitHub
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/docs/FAQ.md

47 48 49 50 GitHub
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/arketyp/semantic-authority.md