ChatGPT

Comprehensive Architecture and Implementation Plan

1. Real-Time Updates & Interactive Data Visualization

Real-Time Data Pipeline: To support live updates (e.g. logs, health checks), the system will use an event-driven approach with WebSockets for low-latency streaming. Unlike Ajax polling – which is "far from real time" and burdens the server with frequent requests [1] – WebSockets provide a persistent full-duplex connection. This dramatically reduces overhead: for example, 10,000 clients polling every second consume ~66 Mbps, whereas 10,000 WebSocket messages use only ~156 Kbps [2]. By pushing log entries and health metrics over WebSockets, users can see new events immediately as they occur (similar to opening a log tail in the browser that "shows error messages as they are written to the log file" [3]). Historical logs will be buffered or stored (e.g. in a time-series database or append-only log) so the interface can retrieve past data on demand, while new data streams in live.

Health Checks & Port Monitoring: The platform will include background processes that periodically check service health (for example, pinging endpoints or attempting TCP connections to critical ports). Results of these health checks (up/down status, response times, port availability) are fed into the real-time pipeline. This allows the UI to display live status indicators (green/red lights, uptime counters, etc.) for each service or port. Enhanced features like alert thresholds can be included – for instance, highlight if a port has been unresponsive for >30 seconds. Historical uptime and error rates can be logged for trend analysis.

Dynamic Visualization (Vanilla JS + WebGL/WebGPU): The client interface will render data through high-performance graphics, using the Canvas API, WebGL, or the new WebGPU for GPU-accelerated visuals. We favor a lightweight, framework-free approach to keep performance high and avoid overhead – using raw WebGL/WebGPU shaders and canvas drawing calls directly. WebGPU, being the modern successor to WebGL, offers lower-level access to the GPU for complex visualizations. However, not all browsers fully support WebGPU yet, so our implementation will **support both**: when WebGPU is available, use it for enhanced graphics; otherwise seamlessly fall back to WebGL [4]. This dual-path approach ensures broad compatibility without sacrificing cutting-edge features. For example, a real-time "wormhole" simulation could use a WebGPU shader for better performance, but automatically degrade to a WebGL shader if needed – the project's demo already showcases a WebGPU wormhole that falls back to WebGL [4].

Interactive Controls & Data Models: To make the visualization interfaces truly interactive, we will provide UI controls for parameters and perspectives. Users might adjust time windows (to see last 5min, 1hr of data), toggle which metrics to display, or tweak shader parameters (color scales, particle size, etc.). These controls can be built with simple UI libraries or custom sliders/buttons in vanilla JS. The existing demos use tools like **dat.GUI** for live parameter tuning (e.g. adjusting physics constants in simulations) [5]; similarly, we can expose controls for log filtering (by level or source) and visual effects. The system will integrate multiple data sources (logs, metrics, health status) into a unified dashboard, possibly layering different visualizations. For instance, we could overlay real-time health status indicators on a canvas-based network map, or plot log frequencies over time on a WebGL graph. Each data "model" will have a visual representation that updates in real time. By using efficient data structures and only diffing changes, we keep the animation smooth even as data streams in rapidly.

Performance Considerations: Real-time rendering and updates can be performance-intensive, so careful resource allocation is key. We will utilize requestAnimationFrame loops for smooth canvas/WebGL animations, and offload heavy computations (like parsing large log files or computing statistics) to background web workers or the Node backend. The WebSocket messages will carry concise data (e.g. JSON snippets or binary if needed) to minimize payload size – an approach in line with information theory principles of reducing redundancy. (In fact, using WebSockets over HTTP polling already cuts out repetitive HTTP headers, reducing latency from ~150ms to ~50ms in Google's observations [2].) We will also cap update frequency for very fast signals: e.g. health pings might be done every few seconds rather than every millisecond, to avoid flooding the UI with more data than a human can observe.

Overall, this module will provide an **"all-in-one" live dashboard**: users can view real-time system behavior, inspect historical trends, and interact with rich visual models. The combination of vanilla JS and GPU-accelerated graphics will enable complex, smooth animations (for example, particle systems or timeline graphs with thousands of points) without external heavy libraries. By integrating these visualizations into the existing interface, operators get an immediate and engaging understanding of system status.

2. User Authentication & Role-Based Access Control (RBAC)

Custom Authentication Mechanism: We will implement user authentication from scratch to meet our specific needs (rather than relying on an out-of-the-box service). This gives full control over security and integration. The authentication flow will be built on proven principles: storing hashed passwords (using a strong hash like bcrypt), enforcing strong password policies, and using secure session tokens. The server (built on Express) will manage login sessions with HTTP-only cookies for web clients and JSON Web Tokens (JWTs) for API access. Indeed, our config already includes secrets for sessions and JWTs [6], indicating a token-based auth mechanism is planned. We'll use the session cookie for browser interactions (for convenience and automatic CSRF protection if using same-site cookies) and JWTs for any stateless interactions (e.g. an external service or mobile app calling the API). Each JWT will be signed and have an expiration (as configured by `JWT_EXPIRATION` in the env) [6], and we'll implement refresh tokens to allow long-lived sessions with periodic re-auth.

Multi-Factor & Key-Based Auth: To enhance security, the system will support optional two-factor authentication. One approach is **WebAuthn** (FIDO2), enabling passwordless or second-factor login with security keys or biometrics. Our environment variables (`RP_ID`, `WEBAUTHN_ORIGIN`) suggest WebAuthn integration [7], so we will likely use a library like *SimpleWebAuthn* to handle registration and authentication ceremonies. This allows users to register a device (platform authenticator or hardware key) and then login with a cryptographic challenge, which is phishing-resistant. For users who prefer traditional 2FA, we can also offer TOTP (time-based one-time passcodes via an authenticator app) or SMS codes, but WebAuthn is more secure. During login, after primary auth (password or passwordless), the server will prompt for the second factor and validate it before granting access. Additionally, **API key** auth will be provided for programmatic access: an admin can generate a long random API key tied to a service account or user role, and clients can pass this key in headers to authenticate. Internally, we'll treat API key usage as a special kind of token login (mapping the key to an identity with fixed privileges).

Single Sign-On (SSO) Support: For enterprise integration or convenience, the system will be designed to interface with SSO protocols (OAuth2/OIDC for Google, GitHub logins, or SAML for corporate SSO). While we are not using an external turn-key solution, we can leverage open standards so our implementation isn't literally from scratch in terms of protocol. For example, to allow "Login with Google", we'll implement an OAuth2 client flow: redirect users to Google's consent page and handle the callback to get the user's email/profile. Similarly, supporting SAML would involve parsing SAML

responses from an IdP. These integrations can be toggled via config. The key is that our auth system will be extensible – we maintain our own user database but can populate it based on SSO assertions. All authentication entries (whether local password or SSO) will funnel into unified session/JWT issuance logic.

Role-Based Access Control: After authentication, authorization is enforced through roles and permissions. We will define roles (e.g. *admin*, *editor*, *viewer*, etc.) and tie them to allowed actions. The design will follow the principle of **least privilege** [8] [9] – each role only has the minimum rights needed. During the design phase, we enumerate user types and resources, and "for every combination of user type and resource, determine what operations (read, write, delete, etc.), if any, the user can perform" [10]. For example, an **Admin** might have full read/write on all data and access to admin dashboards, while a **Viewer** can only read certain data and cannot make changes. Enforcement points will be placed in the application: route middleware in Express will check the user's role (from the session or JWT claims) against the action. By centralizing these checks (e.g. an authorization middleware that consults a policy map), we avoid mistakes like forgetting to protect an endpoint. All undefined access will be denied by default – any request not explicitly allowed by a rule will be rejected [11] (this "deny by default" approach ensures new features are secure until intentionally opened up).

Implementation Details: User accounts will be stored in a database (likely MongoDB given the project's use of it per README [12]). Passwords hashed with salt, and the DB will record roles and 2FA settings for each user. We'll build routes for registration, login, logout, and password reset. On login, in addition to issuing tokens, we'll log the event (for audit trail) and possibly notify the user (especially if 2FA is enabled or if an unrecognized device logs in). The system will include an **admin management UI** where an administrator can invite new users, assign roles, or revoke access. From a coding perspective, we may implement the auth logic in a dedicated module (for example, an `auth.js` route/controller and a `User` model). This module will also handle SSO callbacks and 2FA verifications.

By implementing auth in-house, we ensure it integrates seamlessly with our app's needs (e.g. custom roles, domain-specific permissions). At the same time, we adhere to security best practices (OWASP guidelines) to avoid common pitfalls. The result will be a robust authentication system with support for multi-user access control, built with custom logic but leveraging standard protocols and strong encryption where appropriate.

3. Full-Stack Next.js & Multi-Platform Workflow

Next.js SSR Frontend: The project adopts a **Next.js** application for the web front-end, providing Server-Side Rendering for fast initial load and SEO benefits. The Next.js "visualizer" station is our interactive UI (dashboard and visualizations). In development, it runs as a standalone dev server (e.g. on port 3000 with `npm run dev`). For production, we have two options: (a) run the Next.js server in SSR mode, or (b) do a static export and serve it via our Express backend. We'll use option (b) – exporting the Next app to static files and mounting it on Express – because it simplifies deployment and allows the Node backend (API + static files) to run as one service. The deployment guide confirms this approach: after running `npm run export`, we add a snippet in our Express `index.js` to serve the Next output directory [13]. Then starting the Node server will serve both the API and the Next.js frontend on the same host/port [14]. This integration means we don't need a separate server just for the UI – the Express app becomes an all-in-one API + Web UI server, which simplifies configuration (no separate domain or proxy needed for UI, avoiding CORS issues).

Mobile (React Native) & Desktop Clients: In addition to the web UI, we plan to target mobile (iOS/Android) and possibly a desktop or CLI interface. Rather than building completely independent apps for

each, we will **re-use as much code as possible** to maintain consistency and reduce effort. The repository already illustrates a strategy for this: it contains a React Native project and mentions that the Next.js app's components can be reused in React Native via React Native Web [15]. Specifically, we can either load the Next.js app in a WebView inside a native app (essentially embedding the PWA), or configure a monorepo such that components are shared between web and native. For example, one can import a Next.js component (e.g. a `<Planet/>` visualization component) directly into a React Native screen and render it natively [16]. We'll follow this pattern – structure the code so that core UI components are platform-agnostic React components, and use React Native wrappers for mobile-specific functionality. The mobile app will use the same backend (calling the Express API) so users on mobile have the same real-time data and visualization capabilities on their device.

For desktop, the "desktop" may not need a dedicated native app if the web app is fully featured. We can package the web app as a desktop application via Electron or a similar framework if offline access or OS integration is required. However, a simpler approach is to leverage the fact that Next.js can be a Progressive Web App – users can "install" it from their browser. In any case, our architecture will keep the heavy logic server-side and in shared libraries so the desktop is effectively another client rendering the same content.

Command-Line Interface (CLI): We'll also support a CLI for devops or power-users to interact with the system (for example, running maintenance tasks, triggering deployments, or querying logs from a terminal). This could be a Node.js script or small Express endpoints triggered by a curl script. In the current project, there are scripts like `scripts/astronomy.pipeline.js` and possibly others for data tasks. We can wrap such scripts in a user-friendly CLI (using Node packages like Commander or inquirer for interactions). This CLI will authenticate via an API key or admin credentials and then allow performing actions like exporting data, seeding databases, etc., directly from a terminal.

Deployment Workflow & Automation: A key aspect of this full-stack approach is a robust deployment pipeline. We will automate builds, tests, and deployment to ensure smooth updates. The project already uses GitHub Actions for continuous integration – every push runs tests and builds [17]. We will extend this to continuous deployment: for example, upon a new release or merge to main, automatically build the Docker container and deploy to our hosting environment. Containerization is central to our deployment strategy. We will maintain a Dockerfile that defines a container with Node (for Express API) and the static Next.js site. The docs outline how to build and push this combined image [18]. For instance, we can build an image `express-next:latest` containing both the API and Next output, and then deploy that to a cloud service. On Google Cloud Run, we simply deploy the container and it serves both the UI and API [19]. Similarly, on Azure, we push the image to ACR and update the Web App to use that image [20]. This "build once, run anywhere" approach simplifies maintenance: the same container can run locally, on a VM, or in Kubernetes.

We'll also set up **process management** for when running on VMs or bare metal. The documentation suggests using PM2 or systemd to keep the Node process alive [21]. For example, a systemd service file (as provided) can ensure the server restarts on failure and starts on boot [21]. We will provide these configurations so that if a user chooses to self-host on a Linux server, they can simply use the systemd unit or pm2 script to manage the service.

Continuous Support & Updates: To support ongoing development, our workflow will include staging environments and automated testing. We'll have a staging deployment (perhaps a separate Cloud Run service or just a branch that deploys to a test server) where we can preview changes, especially important for mobile (we might use something like Expo for OTA updates in the RN app, and TestFlight/Play Store betas for mobile testing). Maintenance tasks (like database migrations, cache clearing) will be scripted and included in the deployment process to reduce manual steps. Monitoring will be set up on

the production deployment – using the health check data we already collect – so if any service goes down or if the Node process crashes, alerts are sent out. Logging and analytics from both the client (Next.js has analytics plugins) and server will help us support the app post-deployment.

In summary, the full-stack workflow spans from development to deployment: we develop in a unified monorepo (back-end, front-end, mobile together), share code across platforms, and automate the build/test/deploy steps. This yields a consistent experience on web, mobile, and other interfaces, and a maintainable process for the engineering team to ship updates with confidence.

4. Implementation Options & Integration into Existing Architecture

Evaluating Implementation Choices: We will **enhance the system by exploring multiple implementation options for key components**, ensuring we pick the best of each or even support both where it makes sense. For instance, when building the visualization module, we considered using either WebGL or WebGPU for rendering. Rather than choosing one and excluding the other, we opted to implement *both* paths – leveraging WebGPU's advanced capabilities when available, and defaulting to WebGL for broader compatibility [4]. This dual implementation increases complexity slightly, but future-proofs the system and provides a fallback for older devices. We apply a similar philosophy to other decisions: for example, if deciding between an **Express REST API vs. GraphQL**, we might design the API layer to accommodate both (a REST interface for simplicity and a GraphQL endpoint for more flexible client queries). This way, different interfaces or client needs can be addressed without entirely separate backends.

Integration with Network & Deployment Scripts: The current network architecture appears to be managed by scripts and config files (possibly Bash scripts or systemd units that start the Node server, set environment variables, etc.). Integrating new components into this setup will require updating those configurations. For the Next.js frontend integration, as noted, we added the static serving to the Express app [13], so now the existing `index.js` process also serves the UI. We must ensure the script that was running the API (e.g., `node index.js` in a startup script) continues to work and now covers the UI as well. If there are bash scripts for setting up ports or proxies (for example, an Nginx config or `iptables` rules for port mapping), we'll adjust them to include the new service endpoints. In a Docker/Kubernetes environment, integration means updating the container build and deployment configs: our Dockerfile already handles both API and UI, so any Docker Compose or K8s manifest will be updated to expose the necessary port (say 8000) that serves both. Should the architecture involve multiple containers (database, etc.), we'll ensure the new container or combined container fits into that orchestration (updating Docker Compose yaml or Helm charts accordingly).

Because we now have **multiple interface options** (web, mobile, etc.), we also review CORS and networking. If mobile apps or other clients access the API, we might need to enable CORS for certain domains. If the network uses a reverse proxy or load balancer, it should route requests properly (e.g., route `/api/*` and `/` to the Node app). These are configuration details to get right during integration.

Shader Playground & Live Coding: To further enhance flexibility, we are introducing an in-browser **playground** for live-editing code and shaders. This is essentially a developer (and power user) feature that lets one tweak visualizations on the fly. The project already contains a Next.js based code editor playground with Monaco and live preview [22]. We will integrate this "playground" into the system, possibly as a protected route (for admins or developers) where they can modify shader code or visualization logic and see the results immediately. For example, if the real-time visualization has a

fragment shader controlling the color of data points, the playground could allow editing that shader source in the browser and updating the canvas in real time. This is immensely useful for rapid experimentation and fine-tuning visuals without a full redeploy. It could also serve educational purposes for users interested in the technology. The playground might connect to the running visualization context – e.g., sending the new shader code to the WebGL context to recompile – or run a parallel preview canvas so as not to disrupt the main dashboard. We'll ensure this tool doesn't compromise security (it will sanitize inputs and likely be disabled on production except for authorized users).

By incorporating the playground, we embrace a more interactive development operations culture. Instead of adjusting parameters blindly or editing config files, developers can use the playground to find the right settings live, then export those settings into the codebase once satisfied.

Both Approaches in Parallel: Where we decide to implement both options (as with WebGL/WebGPU), we typically structure the code with a clear interface so that components can swap implementations. For example, we might define an abstract `Renderer` interface that can have a WebGL implementation and a WebGPU implementation. At runtime, we detect the environment and instantiate the appropriate one. This adds some upfront work but keeps the design clean and open to extension (if a new graphics API emerges, we can add it). The same concept applies to other dual implementations: e.g., having both a REST and GraphQL API means we'll have a layer in the server that handles business logic, which both the REST controllers and GraphQL resolvers call into – so we don't duplicate logic, just expose two access methods.

Integrating with Bash/CLI Workflows: If the current system relies on bash scripts for tasks like deployment or monitoring, we will integrate our new components into those scripts or replace manual script steps with more automated solutions. For instance, if there was a `deploy.sh` that pulled code and started the server, we'd update it to build the Next app and then start the server. However, given our move to a more automated CI/CD, we might phase out some bash scripts in favor of Node-based scripts or CI pipeline steps (for better portability and error handling). Still, any essential network config (like bringing up interfaces, configuring firewalls, etc.) done via script will be documented and kept in sync with the new system's needs.

In summary, this phase ensures that all the new features (the advanced visualization engine, the playground, the multi-platform support) **fit seamlessly into the existing environment**. By implementing multiple options and integrating both, we avoid hard trade-offs and instead give the project versatility. The architecture remains consistent – everything ties back into the main Express server and deployment model – which preserves stability while extending functionality.

5. Resource Allocation, Design Principles & Theoretical Considerations

Building a system of this scope requires careful attention not just to coding and features, but to overarching design principles – from low-level efficiency to high-level architecture and even abstract "meta" considerations. Here we outline how we address allocation of resources, structural design, naming, and conceptual integrity, tying in some theoretical underpinnings:

- **Efficient Resource Utilization:** We allocate computational resources in a way that ensures the system remains responsive under load. This means using non-blocking async patterns in Node (leveraging its event loop strengths) and offloading heavy computations to worker threads or separate processes when necessary. Data stacking and sorting operations (for example,

accumulating log entries or sensor readings) will be optimized using appropriate algorithms and data structures – if we need to sort large datasets, we may use on-the-fly algorithms or streaming approaches to avoid high memory usage. Caching is employed to avoid recomputation: recently used data or visualizations can be stored in memory (with cache invalidation strategies in place). We will also monitor performance to prevent memory leaks or runaway processes. Techniques from information theory subtly guide us here: for instance, we aim to maximize the **signal-to-noise ratio** in our data handling – only transmit or compute what is necessary to convey the information. This is evident in the real-time stream compression (removing redundant header data in WebSockets, as discussed) and could extend to using binary formats or compression for any large data payloads.

- **Domain-Driven Design & Layer Separation:** We embrace a domain-driven design (DDD) approach to organize the codebase. This means our core business logic (the "domain layer") is kept separate from infrastructure and presentation. The benefit is a highly decoupled system where business rules aren't entangled with UI or database code [23] [24]. For example, the rules for when to raise an alert on a health check or how to calculate a risk score from metrics will reside in pure domain modules, which have no knowledge of **how** the data is displayed or stored. This separation into layers (e.g. Domain, Application, Infrastructure) follows the classic Onion/Hexagonal architecture principles. It ensures **high cohesion and low coupling**, making the system easier to maintain and extend. Changes in one layer (say, swapping the database or modifying UI framework) won't ripple into others as long as the interfaces contract remains the same. As one practitioner noted, *"Hexagonal architecture separates the Domain, Application, and Infrastructure layers, setting clear boundaries and preventing unnecessary complexity,"* which leads to easier refactoring [25] [26]. In practice, we will define clear module boundaries and use abstractions for dependencies (for example, define repository interfaces in the domain layer, implement them in the infrastructure layer for MongoDB). The outcome is a clean architecture that can evolve without collapsing under interdependency chaos.

- **Naming Conventions & Ubiquitous Language:** We impose consistent naming conventions across the project – not arbitrarily, but driven by the ubiquitous language of our domain. In DDD, a ubiquitous language means using the same terms in code that stakeholders use in conversation. We will collaborate with domain experts (or project owners) to establish clear terminology for components (e.g. if the system is about "missions" and "sensors", those terms should appear in class names, not generic ones). This makes the code more intuitive and self-documenting. For instance, if there's a process that monitors weather risks, we might have classes like `WeatherRiskMonitor` or events like `RiskAlertIssued` instead of generic names. As an example from a DDD context: *using real-life verbs and terms in code makes it easy for others to understand what's going on* [27]. We'll also maintain coding style conventions (camelCase for variables, PascalCase for classes, consistent file naming, etc.) to ensure the project is uniform. Tables, API endpoints, and messages will follow these conventions as well (for example, a database table or collection might be named `missions` or `users` clearly, and not something obtuse). Consistent naming reduces cognitive load and errors, as developers can predict where to find things and what they mean.

- **Type Safety and Correctness:** Although much of our stack is JavaScript, we plan to use TypeScript or thorough JSDoc annotations to introduce static typing to our code. Clear type definitions for data structures (such as the shape of a log message, or the schema of a user object) will help catch errors early and serve as live documentation. This ties into information theory in the sense that a well-defined type system encodes the "information" of our domain models explicitly, reducing ambiguity. For example, instead of passing around loose objects, we

define interfaces/types for a `HealthCheckResult` or `LogEntry` so their fields and meaning are unambiguous in the code.

- **Sorting and Data Organization:** When presenting data (logs, metrics) or running computations, efficient sorting and filtering mechanisms are key. We will choose appropriate algorithms (leveraging built-in sort which is efficient in V8, or specialized data structures like priority queues or indexed DB queries if needed). If real-time sorting is required (e.g. always showing the latest events or highest priority alerts on top), we might maintain data in sorted order as it comes in (using binary insertion or heaps) rather than sorting from scratch each time. This ensures responsiveness even as data volume grows.

- **Metaphysical Consistency (Conceptual Integrity):** This is a more abstract point – essentially making sure the system's design has a unifying vision or philosophy (its "metaphysics"). In practice, this means every part of the system should align with the overall purpose and model of the domain. We avoid ad-hoc hacks or features that don't fit the mental model of the system. If the core metaphor for our system is, say, a "mission control" (to use the project's terminology), then we design modules that fit that metaphor: components like dashboards, consoles, agents, etc., each with a clear role. We draw inspiration from real-world analogies (as metaphors are powerful in guiding design [28] [29]) – for instance, thinking of the data flow like a pipeline or the architecture like a layered defense. However, we remain cautious that metaphors serve clarity and not constrain us unnecessarily [30] [31]. Maintaining conceptual integrity also means following established **design patterns** consistently (such as MVC or observer patterns where appropriate) so the architecture feels cohesive.

- **Information Theory in Design:** We subtly apply information theory concepts to improve system efficiency and communication. This can manifest in how we handle logging and monitoring data – for example, compressing log streams or using entropy measures to detect anomaly (if a log deviates greatly from normal patterns, treat it as higher information content and possibly flag it). It also appears in how we ensure that each subsystem only gets the information it needs – for instance, the UI doesn't pull *all* historical data if the user only needs a summary, reducing information overload. We strive for a design where data is neither under-utilized nor over-transmitted: the right balance of data fidelity vs performance, much like encoding information with minimal redundancy.

In conclusion, by heeding these principles – efficient algorithms, clear layered architecture, consistent naming, type safety, and a strong conceptual model – we create a system that is not only powerful and feature-rich, but also **intelligible, maintainable, and conceptually sound**. This "sweet perspective" ensures that beneath the myriad features (real-time updates, cross-platform clients, fancy graphics), the system has a strong internal coherence. It marries practical engineering with a deeper understanding (from theory and proven practice), hopefully **impressing** the end users with both its functionality and its elegant design.

---

[1] [2] [3] Real time online activity monitor example with node.js and WebSocket
https://thoughtbot.com/blog/real-time-online-activity-monitor-example-with-node-js-and-websocket

[4] [12] [15] [16] [17] [22] README.md
https://github.com/spectra-gallery/arquolab-sandbox-models/blob/76a0e49ec42b9aa5eeff308559ba59ce665a0b53/README.md

5 unified-simulation.md
https://github.com/spectra-gallery/arquolab-sandbox-models/blob/76a0e49ec42b9aa5eeff308559ba59ce665a0b53/docs/unified-simulation.md

6 7 .env.example
https://github.com/spectra-gallery/arquolab-sandbox-models/blob/76a0e49ec42b9aa5eeff308559ba59ce665a0b53/.env.example

8 9 10 11 Authorization - OWASP Cheat Sheet Series
https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html

13 14 18 19 20 21 fullstack-deployment.md
https://github.com/spectra-gallery/arquolab-sandbox-models/blob/76a0e49ec42b9aa5eeff308559ba59ce665a0b53/docs/fullstack-deployment.md

23 24 25 26 27 Why we use Domain-Driven Design and Hexagonal Architecture
https://world.hey.com/bensinclair/why-we-use-domain-driven-design-and-hexagonal-architecture-904b26bb

28 29 30 31 Metaphors in Domain-Driven Design: A Double-Edged Sword | by Masoud Bahrami | Medium
https://masoudbahrami.medium.com/metaphors-in-domain-driven-design-a-double-edged-sword-cb628835862e