ChatGPT

Scope 3 Implementation Plan: Northbound API & Scenario Configuration

This phase implements the **Northbound API (NBI)** for the simulator, enabling external clients to define and query scenarios via gRPC. The goal is to support **create/read/update/delete (CRUD)** operations on all scenario entities (Platforms, NetworkNodes, NetworkInterfaces, Links, ServiceRequests) using Aalyria Spacetime–compatible protobuf schemas [1] [2]. The design builds on the Scope 1/2 architecture (shared KnowledgeBase, entity data models, time controller, connectivity module) and follows the same principles of modularity, extensibility, and use of protobuf/gRPC [3] [4]. All NBI messages will use Aalyria's published schemas (e.g. *network_element.proto*, *service_request.proto*, *network_link.proto*) to ensure full compatibility [5]. Users (or scripts) can thus load a constellation and traffic scenario without manipulating internal data structures directly [1].

Supported Entities and Data Models

Scope 3 exposes the following key entity types via the NBI:

- **PlatformDefinition** (physical assets: satellites, ground stations, etc.): Holds an ID, name/type, and motion source (TLE, static coordinates, or scripted trajectory) [6] [7]. The API allows creating/updating a Platform with its initial position and motion parameters. We include fields for payloads (e.g. bent-pipe transponder configs from *bent_pipe.proto* [7]) but initially stub detailed payload logic (e.g. no active channelization limits) [8] [9]. Upon CreatePlatform, the engine will instantiate a Platform in the KnowledgeBase and start propagating its orbit if a TLE is given (using the existing Scope 1 orbit module).

- **NetworkNode** (logical network node): Represents a device or subnet endpoint, with a unique `node_id`, optional human-readable name/type, optional link to a PlatformDefinition, and node-level settings. The NBI CreateNode call accepts the node properties and (optionally) an inline list of its interfaces [10]. By default we follow Spacetime's older model: interfaces are embedded in the node definition (via `node_interface` list) [10]. Alternatively, an API could allow later additions via separate calls (see below). If the CreateNode request includes a `platform_id`, the simulator will link the node to the existing Platform (attaching it so its position is inherited). If no platform is specified, the node remains standalone (some scenarios use pure "network" nodes without location) [11] [10]. Each NetworkNode can include fields like routing or SDN agent config (as per Spacetime's `RouterConfig` or `SdnAgent` messages), but these may be stubbed or placeholders in Scope 3.

- **NetworkInterface**: Describes a specific interface (wired or wireless) on a NetworkNode. The interface has its own ID, type (wireless/wired), and parameters (transceiver model ID, frequency band, power, etc.). In practice, interfaces are normally created inline with `CreateNode` (embedded in `node_interface`), but we also expose separate interface endpoints for completeness. If not provided in CreateNode, the NBI will allow **CreateInterface(node_id, InterfaceDefinition)** to add an interface to an existing node. The interface data model follows Scope 2's design (e.g. enums for medium type, attached parameters) [4] [11]. Each created interface is linked internally to its parent node in the KnowledgeBase.

- **NetworkLink**: A static connectivity resource linking two interfaces. Users can call **CreateLink** with two interface IDs (and optionally a link ID or name) to declare an always-available link (typically representing fixed terrestrial fiber or a permanent point-to-point link) [12] [13]. This creates a `NetworkLink` object in the state that joins the two interfaces (treated as a bidirectional or two directional links internally). In Scope 3, these links are considered "always up" in the connectivity graph (unless we later simulate impairments). Dynamic wireless links (satellite-to-ground or ISL) are handled by the connectivity engine in Scope 2 and do not require CreateLink calls; they arise from the geometry of node positions [14] [15].

- **ServiceRequest**: Represents a traffic demand (flow) to be provisioned. The NBI **CreateRequest** RPC takes a ServiceRequest message including source and destination (`src_node_id`, `dst_node_id`), flow requirements (desired and minimum bandwidth, maximum latency), priority, and flags like `is_disruption_tolerant` (DTN enabled) and time validity [16] [17]. We focus initially on explicit node-to-node requests (region-based source/dest may be accepted but not fully implemented) [16] [17]. The created ServiceRequest is stored and later fed into the planning/scheduling logic (scope 4+); Scope 3 only registers the demand without doing actual routing. We also provide **Get/ListServiceRequests** and **DeleteRequest** so that clients can see all requests and delete them. Each ServiceRequest record includes status fields (e.g. `is_provisioned_now` and provisioned time intervals) which will be updated by later scheduling algorithms [18] [16]. For debugging and API completeness, a read-only **ListIntents** endpoint may be offered to view internal Intent objects (LinkIntents, PathIntents) created during planning [19] [20].

All these entities will be stored in the simulator's central KnowledgeBase (an in-memory store) keyed by their IDs [21] [22]. Cross-references are validated on create: e.g. creating a Node with a `platform_id` will check that the platform exists, and creating a Link checks both interface IDs exist [22] [4]. Duplicate-ID errors or missing references should result in appropriate gRPC errors (ALREADY_EXISTS or NOT_FOUND) rather than silent failure [23] [22]. The internal data models for these entities mirror the Spacetime API models, ensuring an eventual one-to-one mapping between NBI messages and stored objects [4] [5].

NBI gRPC Services and Endpoints

We will define a set of gRPC services to handle the NBI operations. Each major entity type has a service with the standard CRUD methods. For example:

Service	RPC	Request	Response
PlatformService	CreatePlatform	`CreatePlatformRequest` (contains `PlatformDefinition`)	`CreatePlatformResponse` (PlatformDefinition)
	GetPlatform	`GetPlatformRequest` (`platform_id`)	`PlatformDefinition`
	ListPlatforms	`ListPlatformsRequest` (filter criteria)	`ListPlatformsResponse` (list of PlatformDefinition)
	UpdatePlatform	`UpdatePlatformRequest` (PlatformDefinition)	`PlatformDefinition` (updated)

Service	RPC	Request	Response
	DeletePlatform	`DeletePlatformRequest` (`platform_id`)	`DeletePlatformResponse` (empty)
NodeService	CreateNode	`CreateNodeRequest` (NetworkNode proto)	`CreateNodeResponse` (NetworkNode)
	GetNode	`GetNodeRequest` (`node_id`)	`NetworkNode`
	ListNodes	`ListNodesRequest`	`ListNodesResponse` (list of NetworkNode)
	UpdateNode	`UpdateNodeRequest` (NetworkNode)	`NetworkNode` (updated)
	DeleteNode	`DeleteNodeRequest` (`node_id`)	`DeleteNodeResponse` (empty)
InterfaceService	CreateInterface	`CreateInterfaceRequest` (includes `node_id`, InterfaceDefinition)	`CreateInterfaceResponse`
	GetInterface	`GetInterfaceRequest` (`interface_id`)	`NetworkInterface`
	ListInterfaces	`ListInterfacesRequest` (`node_id` filter)	`ListInterfacesResponse` (list of NetworkInterface)
	UpdateInterface	`UpdateInterfaceRequest` (InterfaceDefinition)	`NetworkInterface` (updated)
	DeleteInterface	`DeleteInterfaceRequest` (`interface_id`)	`DeleteInterfaceResponse`
LinkService	CreateLink	`CreateLinkRequest` (NetworkLink proto)	`CreateLinkResponse`
	GetLink	`GetLinkRequest` (`link_id`)	`NetworkLink`
	ListLinks	`ListLinksRequest`	`ListLinksResponse` (list of NetworkLink)
	DeleteLink	`DeleteLinkRequest` (`link_id`)	`DeleteLinkResponse` (empty)
RequestService	CreateRequest	`CreateRequestRequest` (ServiceRequest proto)	`CreateRequestResponse`
	GetRequest	`GetRequestRequest` (`request_id`)	`ServiceRequest`
	ListRequests	`ListRequestsRequest`	`ListRequestsResponse` (list of ServiceRequest)

Service	RPC	Request	Response
	UpdateRequest	`UpdateRequestRequest` (ServiceRequest)	`ServiceRequest` (updated)
	DeleteRequest	`DeleteRequestRequest` (`request_id`)	`DeleteRequestResponse` (empty)

Each RPC uses the Aalyria protobuf messages for requests and responses. For example, `CreateNodeRequest` carries a `NetworkNode` message (as defined in *network_element.proto* [10]), so clients can supply the full node spec (with embedded `node_interface` entries). We will use these well-defined schemas as the data contracts (e.g. `NetworkInterface`, `NetworkLink`, `ServiceRequest` from Spacetime's API) [5]. The Create RPCs return the created object (populated with any server-assigned IDs or defaults) so clients have confirmation. List RPCs return all entities (or filtered by parent ID) so scripts can retrieve state [22]. Delete RPCs return success or an error if the entity was not found. Update RPCs allow modifying properties (e.g. changing a node's name, or altering a request's bandwidth). All methods will validate inputs and return gRPC status errors for bad requests (e.g. `INVALID_ARGUMENT` if a required field is missing, `ALREADY_EXISTS` on duplicate ID, `NOT_FOUND` if a referenced ID doesn't exist).

Internally, each service method invokes the KnowledgeBase or corresponding module. For example, CreatePlatform calls `KB.AddPlatform(p)` (from Scope 1) [23], and any motion data in `p` is fed to the orbit propagator. CreateNode calls `KB.AddNetworkNode(n)` and, if `n.platform_id` is set, also links it to that Platform (updating both records). CreateInterface updates the NetworkNode's interface list. CreateLink constructs a new link object and updates the connectivity graph (Scope 2 module) with a static, always-available edge. CreateRequest adds the request to an in-memory queue or map for the scheduling engine (future scope). After each create/update/delete, the KnowledgeBase sends appropriate events or logs (optionally via an observer pattern) so that other modules (connectivity, planning) can update their state [4] [22].

Data Store and Entity Mapping

All entities are kept in an in-memory **KnowledgeBase** (KB) or lightweight database. The KB stores separate maps (dictionaries) keyed by ID for Platforms, Nodes, Interfaces, Links, and Requests [21] [22]. We enforce data consistency on each operation: for example, when a NetworkNode is added, `KB.AddNetworkNode` checks that no other node has the same ID and that any referenced platform exists [23] [22]. If the operation is valid, the KB inserts the new object and updates any cross-references (e.g. a Node is appended to its parent Platform's list). Deleting an entity cleans up or unlinks it (e.g. removing a node detaches its interfaces and deletes any links attached to those interfaces). All cross-entity relationships from Scope 2 are preserved: interfaces link to nodes, nodes link to platforms, links join interfaces, and requests refer to source/destination nodes [22] [4]. We also support reverse queries by index (e.g. given a node ID, find all its interfaces or links).

The KB provides lookup methods (`GetPlatform(id)`, `GetNode(id)`, etc.) and listing methods (`ListPlatforms()`, `ListNodes()`, etc.) which back the gRPC Get/List RPCs [21] [22]. Internally, the same protobuf message types are used to represent stored data, so returning them over the wire is straightforward. For example, storing a `PlatformDefinition` proto in memory means that `GetPlatform` can simply serialize it to the client as-is. This one-to-one mapping (protos = internal data objects) simplifies implementation and guarantees schema compatibility [4] [5].

Error Handling and Validation

Each NBI call performs strict validation and returns clear errors. Common rules include:

- **Duplicate ID**: If a Create RPC is called with an ID that already exists, return `ALREADY_EXISTS`. (Scope 1's KB design similarly "returns error if ID conflict" [23].)
- **Missing Reference**: If a Create or Update references another entity that does not exist (e.g. a Node's `platform_id`, or a Link's interface IDs), return `NOT_FOUND`.
- **Invalid Data**: If required fields are missing or values out of range, return `INVALID_ARGUMENT`.
- **Idempotency**: If a client retries Create with identical data after a failure, we should either allow it if no conflict occurred, or consistently error if it did succeed. We design the services to be idempotent where possible.
- **Transactionality**: For multi-step creations (e.g. creating a Node with interfaces), either create all parts or none. We can use a short-lived transaction in the KB or code to roll back on error.

All errors and validation failures are logged at WARN or ERROR level with context (entity type, ID, failing field). The system should never panic on bad input but always return an error status. For instance, trying to delete a nonexistent Platform should simply reply "not found" rather than crash. By adhering to gRPC status codes, we make it easy for clients to programmatically detect and handle errors. Our error design follows the general pattern outlined in Scope 1 (fail early on bad configs) and ensures visibility via logs [23] [22].

API Compatibility and Extensibility

To ensure compatibility with Aalyria Spacetime, we use **Aalyria's published protobuf definitions wherever possible** [5]. All field names and types in our NBI messages match the Spacetime API (e.g. `NetworkNode` message, `PlatformDefinition`, `ServiceRequest`, etc.). This means existing Spacetime clients or scenario textprotos can often be used as inputs with minimal change. We stick to the stable subset of the API: for example, we initially use the "embedded interfaces in Node" model since it matches older Spacetime usage [10] [5]. If Spacetime evolves (e.g. an NMTS graph-based API), we plan to adapt later, but we will encapsulate our usage behind clear service interfaces so as not to break clients.

The design is **extensible**: adding a new entity type (e.g. a future "GroundNetwork" resource) would involve defining a new proto message and service without altering the core engine. Because the KB and core simulation logic treat entities generically (through IDs and interfaces), new features like additional interface types, new link impairments, or extended request parameters can be added by extending the proto schemas and service handlers. We version our APIs carefully: deprecated fields will be marked so that old clients remain compatible, and new optional fields (e.g. extra QoS flags) will not break older service logic.

For instance, if Aalyria later adds a "scheduler affinity" field to ServiceRequest, our `ServiceRequest` message (from *service_request.proto*) will accept it as an optional field. We document such extensibility, and our service handlers ignore unknown fields (the protobuf runtime skips unrecognized fields), preventing failures when clients send newer schemas.

Performance and Concurrency

Although primarily a configuration API, the NBI must handle large scenarios and many calls efficiently. We implement the services in a concurrent-safe way. The KnowledgeBase uses thread-safe data

structures (e.g. concurrent maps or mutex-protected maps) so that multiple gRPC handlers can operate in parallel without conflict. For example, one client creating platforms while another lists nodes should not block each other unnecessarily. We favor read-write locks or lock-free techniques: reads (Get/List) can occur concurrently, while writes (Create/Update/Delete) acquire locks on affected entities. Because all state is in-memory, RPC responses are low-latency. Using Go (as recommended for concurrency [3]) or a similar performant language with goroutines means we can serve hundreds of requests per second if needed.

For very large constellations (thousands of satellites/interfaces), we ensure lookups remain fast (O(1) map access). Bulk creates (e.g. loading 1000 satellites) could be optimized by batch APIs or by streaming RPCs if required. We also consider supporting asynchronous operations: for example, if CreatePlatform takes time to download a TLE or propagate orbits, we might provide an async variant or simply return quickly with the initial state while propagation continues in background.

Throughput metrics (RPCs per second, latencies) are tracked (see Observability below) so we can identify any bottlenecks. In extreme cases, we could shard the KB by region or use a database, but initial design uses a single in-memory store for simplicity.

Testing and Validation

We adopt a rigorous test-driven approach. For each NBI method, we write unit tests and integration tests covering:

- **Unit Tests (Service Logic)**: Test each RPC handler in isolation. Example cases: creating a Platform with valid data, retrieving it, and verifying the returned fields match; creating a NetworkNode with valid and invalid `platform_id`; adding an interface and checking it appears in `ListInterfaces`; creating a Link with nonexistent interface IDs and expecting an error; creating a ServiceRequest and verifying it is stored. We also test edge/error cases (duplicate IDs, bad arguments, etc.) to confirm correct error codes.

- **Integration Tests (Scenario Load)**: Use the actual protobuf messages or textproto files to load complete scenarios. For example, write a script that calls CreatePlatform for several satellites, then CreateNode+interfaces to attach ground stations, then CreateLinks for terrestrial fibers, and finally CreateRequests. After setup, query `ListPlatforms`, `ListNodes`, etc., and compare with expected data. We also run the simulation clock for a few ticks to ensure orbits update (from Scope 1) and that dynamic links appear (from Scope 2).

- **Interoperability Tests**: Where possible, feed existing Spacetime configuration files into our NBI (after minor conversion) to see if we correctly import them. For instance, the "Building a Scenario" tutorial example textproto could be replayed via gRPC calls. This verifies API compatibility [5].

All tests are automated (e.g. using Go's `testing` package or pytest), and we include them in CI. We ensure coverage of both "happy paths" and error conditions. As in Scope 2, we also write some end-to-end scenario tests that mimic realistic use (multiple satellites and requests) to validate overall behavior [24] [25].

Observability and Instrumentation

Observability is built in at the NBI layer as per our architecture guidelines [26]. Specifically:

- **Logging**: Every API call is logged with its method name, parameters (ID values), and result (success or error). We use structured logging (e.g. JSON logs) so tools can filter by entity or request type. For example, a log entry might record "CreateNode succeeded: node_id=GS1 linked to platform SAT2". Errors are logged at WARN/ERROR with details. We avoid logging sensitive or overly large data (e.g. not printing full configuration dump on every call).

- **Metrics**: We instrument counters and histograms (via Prometheus) for key events: number of Create/Get/Update/Delete calls per method, success vs error counts, and latencies of RPC handlers. For instance, a Prometheus counter might track "CreatePlatform RPC calls" and labels for status (`OK`, `AlreadyExists`, etc.). This helps detect API abuse or performance regressions.

- **Tracing**: For complex scenarios, we support distributed tracing (e.g. OpenTelemetry). Each incoming gRPC request can be given a trace/span; internal service calls (to KB or other modules) propagate this context. This allows end-to-end latency analysis when, for example, a high-level client issues multiple RPCs to set up a scenario.

- **Dashboard/Alerts**: We recommend setting up a simple dashboard of metrics (using Grafana) to monitor things like "platforms in KB", "nodes in KB", RPC error rates, CPU/memory usage. Alerts can be configured (e.g. if error rate spikes above threshold, or if memory usage grows unexpectedly) to catch issues early.

These guidelines follow our design principle of high observability [26]. By logging each NBI action and tracking metrics, developers and users can troubleshoot scenario configuration issues (e.g. why a platform wasn't created or why a request failed) quickly.

Summary of Deliverables

In summary, Scope 3 will deliver:

- **NBI gRPC Services** for all major entities: Platform, NetworkNode, NetworkInterface, NetworkLink, ServiceRequest (and read-only Intents). Each service supports Create/Get/List/Update/Delete operations, aligned with Aalyria's protobuf API [2] [5].
- **Data Store Integration**: The central KnowledgeBase is extended to persist these entities and maintain cross-links (node↔platform, interface↔node, link↔interfaces, service↔nodes) [22] [4].
- **API Handlers** that map NBI calls to internal actions: e.g. adding a platform starts the orbit propagation, creating a node attaches it to a platform, creating a link updates the connectivity graph, and storing requests for future scheduling.
- **Validation and Error Handling**: All API inputs are validated (with gRPC error codes for duplicates or invalid references) [23] [22]. This follows the pattern from previous scopes (duplicate IDs cause errors, missing fields are flagged) [23] [2].
- **Compatibility Design**: Messages use the Spacetime API definitions, enabling reuse of existing config files and easing migration [5]. The design allows future API evolution (field additions) without breaking clients.

- **Performance and Concurrency**: Services are implemented in Go (or an equivalent performant language) to leverage goroutines and concurrency [3]. The system can scale to large scenarios by efficient in-memory lookups and non-blocking request handling.
- **Testing Suite**: Extensive unit and integration tests (test-driven development style) will verify each NBI operation and full scenario workflows [24] [25].
- **Observability**: Structured logging, Prometheus metrics, and optional tracing are included for all API operations and key internal events [26].

These deliverables build on the Scope 1 and 2 foundations (core data models, time loop, connectivity) and fulfill the requirements outlined for Scope 3 (NBI and scenario configuration) [1] [2]. The next phases will use this configuration interface to drive the network planning and control logic, but by the end of Scope 3 the simulator will be fully controllable via a gRPC NBI and will accurately reflect any scenario defined through that API.

Sources: Architectural guidelines and scope definitions from the Aalyria Spacetime-compatible simulator documentation [1] [5] [3] [2] were used to derive the above implementation plan.

---

[1] [5] [6] [8] [11] [12] [16] [19] [22] Roadmap for Spacetime-Compatible Constellation Simulator Development.pdf
file://file_00000000b34c71fa97ee0e5193f4087f

[2] [7] [9] [10] [13] [17] [18] [20] Requirements for an Aalyria Spacetime-Compatible Constellation Simulator.pdf
file://file_00000000c4f872078fb77b08397250db

[3] [26] Architecture for a Spacetime-Compatible Constellation Simulator.pdf
file://file_0000000015d07207bf6ce808d3222899

[4] [14] [15] [24] [25] Scope 2 Implementation Plan_ Network Interfaces & Connectivity Evaluation.pdf
file://file_00000000275c7207990b34fd717fca6e

[21] [23] Scope 1 Implementation Plan_ Core Entities & Orbital Dynamics.pdf
file://file_00000000bf207207a0927617bff10cd3