ChatGPT

Architecture for a Spacetime-Compatible Constellation Simulator

Overview & Goals: This backend is designed as a **modular, scalable constellation simulation engine** that mirrors Aalyria **Spacetime's** core concepts and APIs. It will allow users to define complex satellite network scenarios (LEO/MEO/GEO satellites, ground stations, links, traffic demands) and then simulate the **time-varying network state** – updating satellite positions, computing connectivity (contact windows, link quality), orchestrating scheduled beam pointing and routing updates, and providing telemetry outputs. The system implements both Spacetime's **Northbound API (NBI)** for scenario definitions and its **Southbound Interface (SBI)** for control-plane scheduling and telemetry, using Aalyria's open gRPC/protobuf definitions for full compatibility [1] [2]. By adhering closely to Spacetime's API and leveraging open standards (e.g. TLE orbital data, CCSDS DTN protocols), the simulator can serve as an open-source complement to Spacetime – suitable for research, development, and integration testing in the satellite networking field.

Design Principles & Tech Stack: The architecture follows best practices from high-reliability simulation backends (NASA, Google, Amazon) – emphasizing **clean separation of concerns**, extensibility, and observability. We propose using a **modern typed language** like **Go** for the core implementation, due to its efficient concurrency model and first-class gRPC support (notably, Aalyria's own codebase is predominantly Go [3]). Go's lightweight threads (goroutines) and channels are well-suited for handling simulation events and distributed components. The choice of language and frameworks also ensures performance for large constellations and facilitates future contributions. Key frameworks include **gRPC** (for API services), a proven **SGP4** orbital propagator library (for computing satellite orbits from TLEs), and possibly an event messaging system (if needed for decoupling, e.g. **NATS** or Kafka for streaming telemetry). We favor a **microservices-friendly design** – each major function can run as an independent service or as modules in a single process ("modular monolith") for simplicity initially. Containerization (Docker) and orchestration (Kubernetes) can be used in production to scale out components (e.g. running multiple simulation engine instances or agent services). All data interchange uses well-defined **protobuf schemas** (the Aalyria API messages or compatible structures) to serve as clear **data contracts** between modules and to clients [4]. This ensures that scenario definitions or config in textproto/JSON can be accepted with minimal translation, and it improves testability (modules can be fed recorded protobuf messages for unit tests). The system will be **back-end only** (no UI in core), but it's designed to feed external visualization or control UIs via APIs or streaming telemetry. Emphasis is placed on **observability**: structured logging, metric instrumentation (e.g. Prometheus client for Go) and possibly distributed tracing (OpenTelemetry) are built-in, so the running simulation can be monitored and debugged easily. The design also supports both a real-time live mode and an accelerated/offline mode, described later.

High-Level Architecture: The simulator is organized into loosely-coupled **modules/components** with a central data store, as depicted below. Each component has a single responsibility and communicates with others via well-defined interfaces or message streams, allowing independent development and future replacement/upgrades of modules (e.g. swapping out the orbit propagator or adding a new routing algorithm). Key components include: the **Simulation Engine** (time and physics model), a **Network Topology & Connectivity service**, a **Planning/Scheduling engine** for network orchestration, the **API layer (NBI/SBI)**, and supporting services for data management and integration.

High-level architecture of the Spacetime-compatible constellation simulator backend. The API layer (top) handles external gRPC calls from users. The Simulation Core (middle) maintains system state and runs the time-stepped physics (orbit propagation) and link evaluations. The Planning & Orchestration modules compute scheduling and routing decisions, which are sent as time-tagged commands to simulated node Agents via the SBI stream. Arrows denote data or control flow: e.g., scenario definitions flow in from NBI, orbit data flows into the propagator, link updates flow into the connectivity graph, and telemetry/metrics flow out.

Core Simulation Engine (Time & Orbit Propagation)

At the heart is a **time-stepped simulation engine** that advances the state of all platforms and links. A **Time Controller** component governs the simulation clock, supporting multiple run modes: real-time (wall-clock synced) or accelerated discrete time steps for faster-than-real-time batch simulation. In real-time mode, the engine can ingest live data and produce live outputs; in accelerated mode, it can simulate hours or days of network activity in minutes by advancing the sim clock in steps or via event triggers. The Time Controller triggers periodic updates (e.g. each simulation tick or on each event) to drive motion and connectivity calculations.

Orbital Propagation: The engine includes an **Orbit Propagation module** responsible for updating the position and orientation of every space platform. This module ingests standard orbit definitions (primarily **Two-Line Element sets, TLEs**) and uses an SGP4 propagator (or similar) to compute each satellite's ECEF coordinates at the current sim time [5] [6]. Using open libraries (e.g. **Orekit**, *sgp4* C++/Python implementations, or a Go port), the simulator periodically propagates each orbit to reflect real-time position [5]. This allows accurate dynamic positioning of LEO/MEO/GEO satellites, including updates from live TLE feeds – for example, the engine can accept new TLE data at runtime (via an update API or automated fetch) and seamlessly adjust the trajectory modeling. Non-orbital platforms (e.g. fixed ground stations, airborne relays) are handled with alternative motion models: the module supports scripted trajectories or live data injection (for instance, reading live ADS-B for aircraft or simply holding fixed coordinates for ground sites) [7]. All positions are maintained in an Earth-Centered, Earth-Fixed frame for consistency [8]. The Orbit Propagation module publishes position updates into a central **Knowledge Base** (the system's state store) and/or emits events when significant position changes occur (e.g. a satellite comes into view of a ground station). This design ensures other components (connectivity, planning) can subscribe to position updates without being tightly integrated. In summary, the Simulation Engine provides a **consistent time base** and continuously updated platform positions, forming the foundation upon which connectivity and networking logic operate.

Network Topology & Connectivity Service

On top of the moving platform layer, the simulator maintains a live **network topology model** and evaluates connectivity in real time. Each physical platform may host one or more **Network Nodes** (logical network endpoints), each with interfaces representing transceivers or links [9] [10]. The **Knowledge Base** holds a representation of all Platforms and NetworkNodes (with their properties like IDs, types, etc.), as well as all defined interfaces (wireless or wired) and their configurations. This essentially acts as an in-memory **"Spacetime Knowledge Base"** of entities [11] [12]. A **Connectivity/ Link Evaluator** service runs either continuously or at intervals to determine which links are possible at a given time and what their status/quality is. Using the latest platform positions from the Orbit module, it performs **geometric line-of-sight checks** for each pair of compatible interfaces: e.g. for a satellite-ground pair, it checks if the satellite is above the horizon (elevation > min elevation) of the ground station; for a satellite-satellite pair, it checks that the direct line between them is not blocked by Earth (no Earth occlusion) [13] [14]. If line-of-sight exists, the link is marked as **potentially available**. The

connectivity service can also enforce range limits or field-of-view constraints if antenna models specify them (initially, a simple max range or assume omni coverage as a default) [15] [16].

Link Budget & Quality: When a potential link is in view, the simulator estimates its signal quality using a basic **link budget calculation**. For example, applying the **Friis transmission formula** given the distance and frequency band to estimate received power [17], then comparing against receiver sensitivity thresholds to decide if the link can carry data (up, down, or marginal) [18]. In early versions, we assume clear weather and no interference, so if geometry allows a link, it is considered "up" at some nominal bitrate [19] [20]. This yields an approximate **capacity or SNR** value that could be used later for more refined simulation. The connectivity service updates the **link state** in the Knowledge Base – effectively producing an **adjacency matrix** or graph of the network that evolves over time [21]. For performance, this can be optimized by pre-computing **contact windows** for predictable links: e.g. using orbit predicts to calculate time intervals when each satellite-ground pair will have access [22]. These contact windows can be stored and referenced, rather than recomputing geometry every tick, which is important for large constellations.

Topology Management: The system supports both **wireless and wired links**. **Wired interfaces** (e.g. an inter-ground fiber or an inter-satellite laser link that is continuously connected) can be modeled as always-on links with fixed latency and capacity, unless explicitly impaired [23] [24]. Users can also define static NetworkLink entities via the NBI to represent known fixed links (the simulator will include those in the graph) [25] [26]. The connectivity module honors any configured impairments or failures: via API, a link or interface can be toggled down (e.g. simulate hardware failure), and the link evaluator will mark it unusable during that period [27] [28]. Initially, we assume each interface can support **one beam/link at a time** (single connection per radio) [29], deferring complex multi-beam or interference scenarios to future enhancements. We also keep transceiver models simple – e.g. treat antennas as isotropic or with a fixed gain if no pattern given [30]. These assumptions simplify early development while leaving room to incorporate more advanced RF modeling later (contributions could add detailed antenna patterns, adaptive modulation, interference calculations, etc., given the module's pluggable design [31]).

In summary, the connectivity service maintains an up-to-date **Network Graph** of nodes and links. This graph, updated each simulation tick or upon movement events, is the basis for higher-level planning (routing and scheduling). The modular **ConnectivityService** component can be unit-tested independently (e.g. feed it static positions and verify it returns correct LOS link statuses), and can be scaled – in a large scenario, this calculation could be partitioned or parallelized if needed. The clear API (e.g. `CheckLinks()` function or internal event on position-change) ensures the **network layer** is well-separated from the orbital physics underneath.

Data Model & Knowledge Base

All simulation entities and state are stored in a central **Knowledge Base** – this can be an in-memory database or set of data classes that act like a repository of the "world state." It contains tables/collections for Platforms, NetworkNodes, Interfaces, Links, and higher-level constructs like ServiceRequests (demands) or scheduled tasks. Each entity is identified by unique IDs, following the same identifiers as in the Spacetime API (for easy cross-reference). For example, each PlatformDefinition (physical asset) has an ID and attributes (name, type tag, orbit parameters, etc.) [32] [33]; each NetworkNode (network endpoint) has a node_id and attributes (type, associated platform if any, etc.) [34] [10]; interfaces have interface IDs and belong to nodes, and so on. We will **reuse Aalyria's protobuf message structures** where possible for these definitions [4] [35] – for instance, the internal representation of a Platform might be directly an instance of `PlatformDefinition` proto, or a lightly adapted struct with the same fields. This ensures **API compatibility** and eases loading of scenario data

from existing Spacetime config files (textproto or JSON) with minimal translation [36] [37]. The Knowledge Base enforces referential integrity: e.g. you cannot create an interface referring to a non-existent node; linking a node to a platform automatically associates that platform's position with the node. It also tracks dynamic state: current position of each platform (updated by Orbit module), current status of each link/interface (updated by Connectivity module), as well as scheduled tasks and active beams (updated by the Scheduler/agent). For performance and simplicity, an in-memory store (with appropriate locking or using concurrent maps in Go) is sufficient initially [38] [39]. This could later be backed by a database if persistence or multi-process access is needed, but in early phases the simulation runs in-memory (with maybe periodic dumps to disk for checkpointing). The Knowledge Base exposes thread-safe getters/setters or pub-sub notifications so that other components can either query the latest state or subscribe to changes (e.g. the scheduler could subscribe to "link up/down" events).

Entity CRUD via API: The Northbound API layer (described next) operates mainly by creating/updating these Knowledge Base entries. For example, a "Create Platform" API call will result in a new PlatformDefinition object in the store and trigger the orbit propagator to start tracking it [40] [41]. Likewise, deleting a node or link via API removes it from the Knowledge Base and all associated simulation tracking. By centralizing state here, we make sure all modules (physics, connectivity, planning) refer to a single source of truth.

Northbound API Layer (Scenario Definition & Control)

The **API Façade** is the external interface to the simulator. It exposes gRPC services matching Aalyria Spacetime's NBI, so that clients can programmatically control the simulation scenario. All major entity types have Create/Update/Delete and Query methods [42] [11]. For example: clients can create PlatformDefinitions (satellites, ground stations, etc.), NetworkNodes and their Interfaces, and define connectivity or traffic demands – all through API calls rather than hardcoding anything in the simulation. This design follows the principle that the simulation **state is entirely driven by API inputs**, making it easy to integrate with orchestrators or scenario generators. Key API endpoints include:

- **Platform Management:** CreatePlatform (with initial orbit or position data), UpdatePlatform, DeletePlatform, Get/List Platforms [43] [44]. Creating a Platform with a TLE or orbital parameters will register it with the orbit propagator (so it starts updating) [40] [41]. Non-orbital platforms can be created with static locations or even with an external "motion source" (though initially just static/scripted).
- **NetworkNode & Interface Management:** CreateNode (with properties like node type and optional list of interfaces), UpdateNode, DeleteNode [45] [46]. If a Platform ID is associated, the node will automatically track that platform's movement [47] [48]. We allow adding interfaces to a node either as part of CreateNode or via separate CreateInterface calls [49] [50]. Interfaces include fields like type (wired/wireless), a reference to a **TransceiverModel** (frequency band, antenna gain, etc.), data rate for wired links, etc. [51]. (The specific transceiver model definitions would follow Spacetime's proto for WirelessDevice, allowing us to represent different antenna types and bands [52]). The API also supports defining fixed links (CreateLink) for any always-on connections like fiber or pre-deployed laser links [25] [53].
- **Service Requests (Traffic Demands):** The NBI will include an API to register **ServiceRequests**, representing an intent or demand for connectivity between two nodes (source and dest) with certain QoS parameters (bandwidth requirement, latency tolerance, priority, disruption tolerance) [54] [55]. This is akin to a "flow request" that the simulation will try to satisfy via scheduling/routing in later phases. The ServiceRequest is stored in the Knowledge Base and its status (provisioned or not, current throughput, etc.) can be queried [56] [57]. In initial scopes,

creating a ServiceRequest does not immediately allocate anything – it just records the demand. The orchestration logic (Scope 5) will later pick it up and compute a schedule/route for it.

- **Query Operations:** For every entity type, the API provides Get and List methods to retrieve the current state [58] [59]. For example, a client can list all NetworkNodes and see their properties (including any dynamic fields like current position if included, or current routes). This is critical for verifying and observing simulation state via the API. It effectively makes the simulation's Knowledge Base remotely accessible, which is useful for building external dashboards or for automated tests to assert that the simulator's state matches expectations.

Under the hood, the **API layer** will be implemented as one or more gRPC services. We can group related calls (e.g., a **ModelService** for CRUD operations on all entities, analogous to Aalyria's `ModelService` in their open API). The service implementation in our backend will simply translate these API calls into manipulations of the Knowledge Base and trigger any necessary re-calculations. For example, after creating a new satellite platform via NBI, the service will call into the Orbit Propagation module to start propagation, and notify the Connectivity service that a new node exists (so it can include it in link calculations). By mirroring the **protobuf definitions** of Spacetime's API, we ensure compatibility – potentially even allowing us to use Aalyria's existing client libraries (in Go or Python) to interact with our simulator with minimal changes [4].

Authentication & Access Control: For now, since this is a local simulator, we might not implement full auth (or use a simple token), but we design the API with the assumption that authentication (e.g. JWT in metadata) can be added in the gRPC interceptors, to remain true to production-like deployment [60].

Batch Configuration: In addition to live gRPC calls, we plan to support loading a scenario from a file (e.g. a textproto scenario definition used by Spacetime). This would simply be a matter of parsing the file into the same protobuf messages and calling the same gRPC handlers internally. This allows users to bootstrap complex scenarios in one go.

The NBI is the main **integration point for external tools** and also enforces that our design is not hardwired to any specific scenario – everything is data-driven via API, which aligns with Spacetime's philosophy of an SDN controller for space. By completing the NBI implementation (Scope 3 of the roadmap), a solo developer can fully configure a scenario via API without touching internal code [61].

Scheduling & Southbound (SBI) Interface Simulation

A core feature that elevates the simulator from static modeling to an active network orchestration platform is the implementation of a **Scheduling Engine** and **Southbound Interface (SBI)**. In Spacetime, the SBI is how the central controller sends time-tagged commands to distributed **agents** (on satellites, ground nodes, etc.) and receives telemetry. We replicate this by simulating an **embedded agent** for each node, and a **controller-side scheduler** that sends it commands via a gRPC stream. The outcome is a closed-loop control-plane simulation: the controller (simulator) schedules network configuration changes (like "enable this link at time T" or "update this routing table"), and the agent applies them in the simulated hardware at the right time, then reports status/telemetry back [62] [63].

Embedded Agent Model: We will instantiate a **Simulated Agent** for each NetworkNode (or each Platform that hosts a node). This could be implemented as a Goroutine (or thread) that represents the logic running *on that node*. The agent listens for incoming **SBI commands** (beam configurations, routing updates, etc.) and also generates telemetry reports. In our design, agents are internal objects, but we treat them as if they were separate processes – communicating with the controller via the same gRPC interfaces an external agent would use. (In fact, one could choose to run actual agent code from Aalyria's open-source Go agent inside the simulator process, but it may be simpler to implement a

lightweight simulated agent.) Each agent maintains a **schedule queue** of future actions for that node: e.g. "at time T, turn on beam to GS1; at time T+Δ, turn it off" etc. [64] [65]. The agent watches the simulation clock (which it can query via Time Controller or get tick events) and when the time matches a scheduled action, it executes it – meaning it updates the node's state in the Knowledge Base (e.g. marks an interface as active pointing to a target, or adds a route entry to the node's routing table). Execution may also trigger changes in the connectivity graph (e.g. turning on a beam actually makes a previously potential link become an active link with capacity).

Scheduling Service (CDPI): The controller side has a **Scheduler/Orchestrator** module which decides *what* actions to schedule *when*. It implements the **Control Data-Plane Interface (CDPI)** gRPC stream (`ReceiveRequests`) as the server. When the simulation starts, each agent (acting as a gRPC client) "connects" to the scheduler stream, identifying itself by its agent/node ID [66] [67]. The scheduler then sends down any pending configuration tasks for that agent. We follow the Spacetime message protocol: the controller sends a sequence of **CreateEntryRequest** messages, each containing a time-tagged command (oneof: UpdateBeam, DeleteBeam, SetRoute, DeleteRoute, etc.) plus a unique request ID [68] [69]. The agent, upon receiving these, will insert them into its local schedule. At the designated time, the agent executes the command and then sends back a Response (acknowledgment with status) for that request ID [70] [70]. This ack allows the controller to know the action was applied (in simulation it's almost always successful unless we choose to simulate a failure). After execution, the link state or routing state in the sim is thereby changed. For example, an **UpdateBeam** command carries a Beam configuration (target interface, freq, power, etc.) and a start time – the agent will activate that beam at the given time, which in our sim means marking the link between the specified interfaces as "up" (assuming geometry allowed) [71] [72]. A corresponding **DeleteBeam** at time T2 would turn it off, freeing that interface [73] [74]. Similarly, **SetRoute** at time T might add a route entry in a node's routing table (we maintain a simple per-node route table in the Knowledge Base) [74]. The scheduling stream also supports some housekeeping messages like **FinalizeRequest** (controller hint to agent to drop all tasks before a cutoff time, used for cleanup) and **Reset** (agent->controller to indicate it restarted and cleared its schedule) [75] [76]. We implement these for completeness: e.g. our agent on Reset will inform the controller and the controller will resend the current desired config if needed [76] [77]. The net effect is that at simulation runtime, we have a faithful emulation of the **SBI mechanism**: even though controller and agents are in one program, they interact via the defined gRPC streams and messages. This not only ensures we match Spacetime's behavior, but also allows the possibility of plugging in a real external agent later (e.g. if someone wanted to test a hardware-in-the-loop where a real device runs an agent, it could connect to our scheduler service).

Basic Scheduling Logic: In the initial implementation (Scope 4), the scheduler's logic will be rudimentary – essentially demonstrating the mechanism. For example, we might schedule **beam on/off tasks** for every feasible link by default: as soon as a satellite comes into view of a station, schedule an UpdateBeam to activate that link immediately (or with some small lead time) and a DeleteBeam for when it goes out of view [78] [79]. This would actively turn links on and off in sync with geometry, achieving a basic emulation of link handovers. Similarly, we could schedule static routes when links are up (though initial routing can be static or single-hop). The key is that by implementing the scheduling pipeline (controller->agent->action->telemetry), the network becomes **dynamic** – links appear and disappear as scheduled, and later a smarter algorithm can decide *which* links to activate for optimal network performance. Essentially, Scope 4 covers the mechanism (the "how"), while Scope 5 will cover the policy (the "what/when").

Telemetry Service: Alongside scheduling, the SBI includes a **Telemetry API** where agents report status and metrics to the controller [80] [81]. We implement a TelemetryService (gRPC) that the agents will call periodically with metrics. Each agent can periodically send an **ExportMetrics** message containing data like interface status (up/down, utilization), link quality, possibly buffer occupancy, etc. [82] [83]. Initially,

we focus on basic metrics: e.g. each agent reports the status of each interface (is it active/up and with whom) and perhaps counters of how much data was “theoretically” sent if a flow is going through (we can simulate simple byte counters for active flows) [84] [85]. This telemetry feed serves two purposes: (1) It allows the simulated controller to monitor the network and make decisions (in a real system, the controller might alter scheduling based on telemetry; we can incorporate simple feedback loops if needed), and (2) It provides a stream of live data that we can expose to users or UIs. For example, an external visualization could subscribe to telemetry to display current link statuses or throughput on each link. In the simulator, since we have ground-truth knowledge already, telemetry is somewhat redundant; but we implement it to remain true to the API and to allow testing of external components that expect Spacetime-like telemetry.

Data Flow and Consistency: The SBI interactions are carefully orchestrated with the sim state. The **Scheduler/Orchestrator** reads the global state (from Knowledge Base and from higher-level planning inputs) to decide what to schedule. When an agent executes an action at time T, it updates the state (which triggers connectivity or routing changes). Telemetry then reports those changes back. Because everything runs in one process, we ensure ordering: e.g. to simulate real conditions, the agent might update the state and then the telemetry is sent slightly after, so the controller can verify the action took effect. We may introduce artificial control-plane latency if needed (the Spacetime model allows configuring latency per agent [86] [87], but we will initially assume near-zero latency).

In summary, the Scheduling/SBI subsystem makes the simulator **interactive and adaptive**. It lays the groundwork for advanced features: once we have this pipeline, we can implement complex scheduling algorithms (deciding optimal link schedules under constraints) and advanced routing, knowing we have a way to enact those decisions in the simulated network.

Network Planning & Routing (Advanced Orchestration Logic)

With the core simulation (positions, basic connectivity) and the scheduling mechanism in place, the next layer is the **brain of the network**: modules that automatically plan how to use the time-varying connectivity to satisfy user demands. This includes route computation, scheduling optimization, and resource management. We design this as a higher-level **Orchestration component** that consumes inputs (the current or predicted network topology from Connectivity service, plus the list of ServiceRequests and any policies) and produces outputs (schedules and configuration tasks handed to the Scheduler component).

Routing Engine: A **Routing module** will be responsible for finding end-to-end paths through the dynamic network for each ServiceRequest (flow). Initially, we implement a simple routing strategy: e.g. treat the instantaneous connectivity graph at a given time or over a time window, and run a shortest path algorithm (Dijkstra or BFS) to find a viable path from source to destination [88] [89]. This can be extended to handle time-varying connectivity: for disruption-tolerant flows (DTN), the route might be a **time-sequenced path** – data can hop to an intermediate node and wait until the next link comes up (store-and-forward) [90] [91]. For continuous (non-DTN) flows, the routing engine might have to compute a path that is available at one moment and hand over to an alternate path when topology changes, or simply mark the flow as down during outages [92] [93]. In early implementation, we could choose a straightforward approach: pick a single path for each flow (e.g. source -> sat -> dest) and if it’s not continuously available, so be it (the flow goes down intermittently). As we improve, we can implement more elaborate routing that changes over time or pre-computes multiple contingency paths [92] [94]. The routing engine likely works closely with the connectivity service – possibly using the contact window computations to know when certain links will be up or down, thereby planning routes that leverage future links if DTN is allowed.

Scheduling Optimization: The **Scheduling/Planning engine** (which is part of the Orchestrator) takes the routes and demands and decides **who gets to use which link when**. When multiple flows contend for the same link or a satellite has limited beams, this module must allocate resources over time. Key aspects include enforcing **interface beam limits** (e.g. a satellite with one antenna can only connect to one ground station at a time, so if two flows want it, we must time-slice or prioritize) [95] [96]. The scheduler will consider **flow priorities**: higher priority service requests get first use of scarce link time or capacity [97]. We might implement a simple greedy algorithm: iterate through requests in priority order and schedule as many as possible, then lower priority ones get whatever capacity remains. For each flow, using the route info, schedule beam on/off events for each hop. For example, if a satellite needs to downlink to GroundStation A from 12:00 to 12:05 to serve flow X, schedule that beam on at 12:00 and off at 12:05, and similarly ensure the ground station's antenna is free in that interval for that link. If another flow needs the same satellite at 12:00, and it cannot use it concurrently, the scheduler might queue it for 12:05 or route it differently. **Conflict resolution** policies will be applied here (round-robin, preemption of low-priority, etc., as configurable). The output of this planning stage is a set of **scheduled tasks** (beam activations, beam deactivations, route installations) with timestamps – exactly what the Scheduler component needs to send to agents. These tasks are then fed into the SBI scheduling pipeline described above (the Orchestrator module effectively programs the Scheduler/ Orchestrator component with these tasks to distribute).

We will also incorporate other constraints into scheduling decisions: for example, ensure that a node's total transmit power budget isn't exceeded if two links overlap in time – if a satellite can transmit 100W total and we scheduled two simultaneous beams that would require 120W, the scheduler should avoid that (either by not overlapping them or by knowing the power per link) [98] [99]. Similarly, if a node has limited data storage (for DTN), the scheduler should ensure not to indefinitely queue more data than storage can hold [100] [101] (e.g. if a link is down for too long, some data may be dropped or we mark the flow as unmet). These considerations ensure the simulation reflects real-world limits.

By the end of this planning stage (which corresponds to Scope 5 in the roadmap), the simulator will be capable of autonomously managing the network to a large extent: **multi-hop routing, store-and-forward buffering, priority-aware scheduling, and recovery from outages** [102] [103]. All these algorithms are encapsulated in the Planning module, which we keep separate from the lower-level mechanics. This means different research algorithms or external controllers could be plugged in. For example, one could replace the default scheduler with an AI-based optimizer without touching the orbit or link code.

Extensibility for Advanced Modules: The architecture anticipates future modules like **DTN protocol simulation**, where actual bundles are generated and forwarded (to test routing decisions in detail), or **federation** with other networks, where allow_partner_resources flags on flows might allow using "external" assets [104] [105]. While full federation (connecting two simulation instances or a sim+real network) is beyond initial scope, our design keeps it in mind: e.g., the API has a flag for federation which we simply store now [106], and we avoid assumptions that all nodes are in one simulation. If federation were implemented, the scheduler might forward requests to another orchestrator via an API instead of handling everything internally. We also plan for **hardware-in-the-loop**: because our SBI uses real gRPC, one could run an actual Spacetime agent on hardware and have our controller treat it like any other agent – allowing a real modem or antenna to be "in the loop" with the rest simulated. The modular architecture (especially using gRPC interfaces and clearly defined data models) makes such integration possible without core changes.

Modes of Operation: Real-Time vs. Offline Simulation

A strength of this backend design is flexibility in how time is handled, enabling both live interactive use and faster-than-real-time analysis. In **Real-Time Mode**, the simulation can align its time progression with wall-clock time and ingest live data streams. For instance, it could subscribe to live TLE updates or even live telemetry from on-orbit satellites to correct orbits, and it can output telemetry on-the-fly to live dashboards. The time controller would run in a loop sleeping as needed to keep simulation time in sync with real time. This mode allows the simulator to act as a "digital twin" of an active constellation, where it mirrors and predicts network state slightly ahead of actual time.

In **Offline/Accelerated Mode**, the simulation runs as a batch job: the time controller will fast-forward the simulation clock, only limited by computational resources. This is useful for running what-if scenarios or long-duration network schedule planning. In this mode, external real-time feeds are not needed (the simulator might load a static TLE for the day and simulate the whole day's contacts). Outputs can be saved to log files or databases for post-analysis instead of streamed live. We may allow the user to configure a time scale factor or to run "to completion" (e.g. simulate 24 hours then stop). The architecture supports this because all time-dependent logic goes through the central Time Controller and schedule queues. For example, the agents' scheduled tasks execute based on simulation time, and the time controller can jump in discrete increments from one event to the next (discrete-event simulation). We could optimize by using an **event-driven scheduler**: rather than fixed 1-second ticks, the simulation can compute the next interesting event (next contact start/end, next scheduled beam action, etc.) and jump to that time, which is efficient for long idle periods. This is an optional optimization if needed for scale.

Regardless of mode, the **determinism** of the simulation is important: given the same initial state and inputs, the sequence of events and outputs should be reproducible. We'll design with that in mind (e.g. avoid non-deterministic concurrency where it affects outcomes, or provide a simulation seed for any pseudo-random processes). This is critical for testing and for trust in the simulator's predictions.

Finally, **time management** includes the ability to **pause, resume, or step** the simulation via API. We could provide an admin API to pause the time progression (e.g. to inspect state or synchronize with an external event) and to manually advance a single tick or to a specific timestamp. This kind of control is useful in debugging and in integration testing scenarios.

Observability, Testing, and DevOps Considerations

To be "world-class", a backend must be operable and maintainable. We embed **observability** into the design from the start. Each module will produce logging at appropriate levels – e.g., the Orbit module might log when it loads a new TLE or if propagation fails, the Connectivity service can log link events (up/down) with timestamps, the Scheduler can log every command sent or any schedule conflict detected, etc. These logs can be structured (JSON logs or similar) for easy filtering. We also include **metrics** collection: for instance, number of active links, number of scheduled tasks, message processing latency, etc., which can be scraped by Prometheus or monitored in dashboards. This allows monitoring the health and performance of the simulation (useful especially if running a long simulation as a service).

We plan extensive **testing** at multiple levels. Unit tests will target each module (physics, link calc, scheduling) with controlled inputs. Integration tests will use the public API – for example, a test might load a small scenario via NBI and verify that after simulation run, the expected contact windows or routes were established. Because the system adheres to protobuf APIs, we can even use recorded

scenarios from Spacetime (if available) as test cases, ensuring compatibility. The modular design (and usage of dependency injection where applicable) makes it feasible to swap out real components with mocks in tests (e.g., a fake OrbitPropagator that returns predetermined positions to test the link calculator). The development workflow will use modern DevOps tools: continuous integration to run tests and perhaps even simulate a small network on each commit.

Scalability for Community Contributions: As a solo developer builds the initial system, care is taken to establish clear module boundaries and documentation so others can join. For example, the code repository might have separate folders for `simulation_core/` vs `network_planning/` vs `api/` services, each with its own README and proto definitions. New contributors could work on a new propagation model (e.g. add a higher-fidelity orbital perturbation module) by extending `simulation_core/orbit/` without affecting APIs. We also plan to contribute an open dataset of example hardware models (similar to Aalyria's `contrib` directory [107]) so that adding a new antenna type or satellite configuration is straightforward via data files, not code changes. The use of standard interfaces (gRPC, pub-sub events) for inter-module communication means a contributor could even replace an entire module by a microservice in another language if desired (for instance, a C++ high-performance contact calculator could plug in, communicating over a socket or RPC to the main sim). This flexibility and modularity ensure the project can grow into a community-driven platform.

Operational Deployment: While the simulator can run on a single machine for development, a production-grade deployment might split components: e.g., run the API service in one container, the core simulation engine in another, and perhaps agents each in separate threads or containers (though for 1000s of satellites, one container running all agent goroutines is fine). If very large scale, one could even partition the constellation into regions handled by different instances that sync at boundaries (this edges into federation). The design does not assume centralized single-process only; it uses central state now for simplicity, but could evolve to a distributed state (with a database or distributed memory) if needed.

Finally, we note that visualization or UI is **out of scope** for the backend, but the backend makes it easy to build one. For example, the system could output a live stream of events (contact events, link up/down, etc.) to a message bus or even just via the telemetry API, which a separate UI process can subscribe to. In initial version, we rely on textual logs or simple metrics as "outputs" for visualization [108]. But designing the backend with an **API-first approach** means any external tool (web dashboard, CLI, etc.) can be added later without needing changes in the simulation logic.

Conclusion

This architecture leverages open standards and Spacetime's proven model to create a **comprehensive constellation simulator**. By replicating Spacetime's key APIs and data models, the system can accept the same inputs and produce similar orchestration behavior, from satellite motion to scheduled network configurations. The design is modular and robust: each concern (orbits, connectivity, scheduling, routing, data management) is handled in a separate component with clear interfaces, allowing the solution to scale in complexity and performance. Following the roadmap, initial development builds the core physics and data model (Scopes 1–3), then adds the control-plane feedback loop (Scope 4) and advanced optimization (Scope 5), all while keeping the codebase organized for collaboration. The result will be a **world-class backend** capable of simulating dynamic space networks with high fidelity, real-time control, and extensibility – essentially an open-source Spacetime twin [109]. Such a platform can drive research and innovation in space-based networking, providing a foundation for modules like DTN routing, federation with other networks, or even hardware-in-the-loop testing as the community grows. With solid architecture and best practices, this simulator is poised to become a reference system for

satellite network digital twins, combining realism (SGP4 orbits, real link budgets) with the flexibility of software-defined networking principles.

Sources: Derived from the provided requirements and roadmap for an Aalyria Spacetime-compatible simulator, which in turn reference Aalyria's public API definitions and documentation for accuracy. Key design points – from dynamic TLE-based orbit updates [5] to SBI scheduling streams [62] [63] and DTN store-and-forward handling – have been incorporated to ensure alignment with Spacetime's concepts. By building on these foundations and following the phased roadmap, the architecture meets current requirements and remains adaptable for future enhancements in realism and scale.

---

[1] [2] [5] [6] [7] [8] [9] [10] [32] [33] [34] [42] [70] [76] [77] [78] [79] [80] [81] [82] [83] [84] [85] [86] [87] [98] [99] [100] [101] Requirements for an Aalyria Spacetime-Compatible Constellation Simulator.pdf
file://file_00000000ff487206b4408a1a7ea6ff08

[3] [107] GitHub - aalyria/api: The API for Spacetime, a temporospatial software defined networking platform.
https://github.com/aalyria/api

[4] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [35] [36] [37] [38] [39] [40] [41] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [61] [62] [63] [64] [65] [66] [67] [68] [69] [71] [72] [73] [74] [75] [88] [89] [90] [91] [92] [93] [94] [95] [96] [97] [102] [103] [104] [105] [106] Roadmap for Spacetime-Compatible Constellation Simulator Development.pdf
file://file_000000007cf8720698f3ec5360c7ab2f

[60] Authentication – Spacetime Knowledge Base - Aalyria
https://docs.spacetime.aalyria.com/api/authentication/

[108] [109] Requirements for an Aalyria Spacetime-Compatible Constellation Simulator.pdf
file://file_0000000012b87206951eb78067248b9a