ChatGPT

Scope 1 Implementation Plan: Core Entities & Orbital Dynamics

Module Breakdown

In **Scope 1**, we implement the foundational modules of the simulation – focusing on core entities (platforms and nodes) and orbital motion. The architecture will consist of loosely-coupled components with clear responsibilities and interfaces [1]. A central **Knowledge Base** will store all state, and each component will interact through well-defined APIs or events. Below is a breakdown of the key modules:

Platform & NetworkNode Data Models

Responsibilities: Define data structures for physical platforms and logical network nodes, mirroring Aalyria Spacetime's core entities. These classes hold all static properties and dynamic state (position, etc.) of assets in the constellation [2]. They primarily serve as containers for data, with minimal logic beyond accessors.

Public Interfaces: These are plain data classes (or structs in Go) with public fields (or getter/setters) for each attribute. They will be used throughout the simulation and via APIs (e.g. NBI) as the primary data contracts [3] [4]. Key interfaces include: - Constructors or factory functions to create a PlatformDefinition or NetworkNode from config (e.g. from proto or JSON inputs). - Methods to link a NetworkNode to a Platform (e.g. `node.AttachToPlatform(platformID)`). - (Optionally) helper methods for convenience (e.g. `node.IsMobile()` based on type, etc.).

Dependencies: No external dependencies beyond basic types – these models align with Aalyria's protobuf definitions [3]. They may depend on common types like coordinates or time (for position/orbit data). They do not themselves perform computation – other modules (e.g. orbit propagator) will update their fields. This separation ensures they remain simple data containers (following the Entity definitions from Spacetime).

Knowledge Base (State Store)

Responsibilities: Maintain an in-memory store of all entities (platforms and nodes), providing thread-safe access and update mechanisms [4]. The Knowledge Base (KB) acts as the source of truth for simulation state at any time, analogous to Spacetime's knowledge base of entities [5] [6]. It handles: - **Storage:** Keep dictionaries/mappings of PlatformDefinition and NetworkNode objects keyed by their IDs (unique identifiers). - **Lookup & Listing:** Provide APIs to retrieve a platform or node by ID, or list all of them (or by type/category). - **Updates:** Expose methods to add new entities (e.g. when a scenario is initialized) and to update fields (especially coordinates each tick). - **Linking:** Manage associations between nodes and platforms. For example, if a node is attached to a platform, store that relationship so the node's position can be derived from its platform [7]. - **Observability:** Optionally, notify observers of changes. The KB can implement an observer pattern or pub-sub mechanism: e.g. when a platform's position is updated, it can emit an event or invoke callbacks for any listeners (such as a telemetry logger or a connectivity module in later scopes) [8].

Public Interfaces: - `AddPlatform(p *PlatformDefinition) error`: Insert a new platform (returns error if ID conflict). - `AddNetworkNode(n *NetworkNode) error`: Insert a new node (error if conflict). If `n` references a platform, the KB can validate that the platform exists and store the linkage. - `GetPlatform(id string) *PlatformDefinition`: Fetch a platform by ID (or return nil if not found). - `GetNetworkNode(id string) *NetworkNode`: Fetch a node by ID. - `ListPlatforms() []PlatformDefinition` and `ListNetworkNodes() []NetworkNode`: Return copies or snapshots of all entities for external inspection (to avoid direct modification). - `UpdatePlatformPosition(id string, coord Motion)`: Update a platform's coordinates in a thread-safe manner, and also update any derived data (e.g. notify linked node or trigger observer events). - `Subscribe(func(Event)) (unsubscribe func())`: Allow registration of listeners for state changes (e.g. a callback for "platform moved" events or general entity updates).

Dependencies: The KB may use synchronization primitives (mutex/RWMutex) from the Go runtime for thread safety. It does not depend on external databases in Scope 1 (data can be transient in-memory). All interactions with KB are via its API or events; modules like the time controller or orbit propagator will call these methods to read/update state.

Simulation Time Controller

Responsibilities: Govern the advancement of simulation time and orchestrate the periodic update loop [9]. The Time Controller handles **tick generation** (the simulation "heartbeat") in either real-time or accelerated modes. It provides control over starting, pausing, and stopping the simulation, as well as adjusting the speed (time scale).

Key responsibilities: - **Time Keeping:** Maintain the current simulation time (e.g. as a `time.Time` or simulation timestamp) and a simulation epoch start. In real-time mode, this might track wall-clock alignment; in accelerated mode, it calculates time progression faster than wall clock. - **Tick Scheduling:** Emit ticks at a fixed step or variable step. For example, a default could be 1-second simulation time per tick. It may use a goroutine with a `time.Ticker` for real-time mode (tick every 1 second) and a tight loop or short sleeps for accelerated mode. - **Mode Management:** Support at least two modes – **real-time (1x)** where simulation time progresses in sync with wall clock, and **accelerated** where the simulation runs as fast as possible or at a specified faster rate [9]. In accelerated mode, the controller might increment time in discrete steps (e.g. 1-second or 1-minute steps) without waiting for real time to pass. - **Pause/Resume:** Provide a way to pause the simulation (freeze the simulation clock and suspend tick events) and resume it. Pausing could be implemented by suspending the ticker/loop (e.g. not advancing time or sending tick signals until resumed). The controller should maintain state (running/paused) and ensure a clean resume (no lost time accumulation). - **Tick Broadcast:** Notify other components on each tick. The Time Controller can maintain a list of registered **tick listeners** (e.g. the Orbit Propagation module, future link calculators, etc.). On every tick, it calls each listener or broadcasts an event that a new time step is available. This decouples the time loop from specific update logic.

Public Interfaces: - `Start(realTime bool, step time.Duration, scale float64)`: Begin the simulation loop. Parameters might specify mode: e.g. `realTime=true` for wall-clock sync, or `scale` for acceleration factor, etc. A default step (like 1s) can be used for tick granularity. - `Pause() error` / `Resume() error`: Pause or continue the simulation. On pause, the controller might simply set an internal flag and stop emitting ticks; on resume, it recalculates the next tick time. - `Stop()`: Gracefully stop the simulation loop (terminate the goroutine). - `Now() time.Time`: Get the current simulation time (could also be accessed via global state or passed in tick events). - `AddTickListener(func(simTime time.Time))`: Register a callback to be invoked each tick

(returns an unsubscribe handle). - Possibly `SetRate(scale float64)`: Dynamically adjust the speed (not strictly required for Scope 1 but a consideration).

Dependencies: Uses Go's time utilities and concurrency (goroutines, channels or locks) to schedule ticks. Integrates with the Orbit Propagation module (which will register a tick listener) and the Knowledge Base (which may be updated on each tick). The design should avoid tight coupling – e.g. the controller doesn't directly update orbits, it merely signals others to do so, consistent with an event-driven design [8]. This modular approach allows replacing the time mechanism or adjusting tick frequency without altering other modules.

Orbit Propagation & Motion Models

Responsibilities: Update the position (and orientation, if applicable) of every platform each tick, according to its motion model [10]. This module integrates an **SGP4 orbital propagator** for satellites and provides a strategy interface for different motion types. It ensures all platform coordinates in the Knowledge Base are refreshed based on the current simulation time.

This module can be structured around a **PlatformMotion** interface (strategy pattern) that defines how to compute the platform's position:

```
type MotionModel interface {
    UpdatePosition(simTime time.Time, platform *PlatformDefinition)
}
```

Each PlatformDefinition is associated with one MotionModel implementation, chosen based on the platform's configuration (orbital, static, etc.). The Orbit Propagation module manages these models and coordinates their use on each tick.

Key motion model strategies: - **Static Position Model:** For fixed platforms (e.g. ground stations), this model leaves the position unchanged every tick. The PlatformDefinition's coordinates are set once (e.g. from a known lat/long or ECEF coordinate) and the UpdatePosition simply does nothing (or ensures it remains constant). Orientation can also remain constant (e.g. a ground station might have a fixed orientation pointing upward). - **Orbital (SGP4) Model:** For satellites with TLE data, this model uses an SGP4 library to compute the ECEF position at the given simulation time [11]. On each tick, it calculates the satellite's position vector (X, Y, Z in Earth-Centered Earth-Fixed frame) from the TLE orbital parameters and current time. We will integrate a proven SGP4 implementation (via a Go library or binding to Orekit/Python if necessary) [12]. The model will likely store precomputed state from the TLE (e.g. mean motion, orbital elements) in an internal struct when initialized, so that each UpdatePosition call is efficient. Orientation for satellites can be left as a default (e.g. align with velocity or NADIR pointing) – for Scope 1, we will include an orientation field but not actively simulate attitude changes (just set a sensible default or identity rotation). - **Scripted / External Motion Model:** For moving but non-orbital objects (e.g. aircraft, drones, or hypothetical moving user terminals), this model updates position based on an external source or predefined trajectory [13]. In Scope 1, we can support a simple scripted trajectory: for example, a list of waypoints or a function of time. Alternatively, this model can be a stub that always returns the last externally provided position. For now, we will allow injection of position data via an API or test hook – e.g. an external thread can call `KnowledgeBase.UpdatePlatformPosition()` for a platform, and the external motion model's UpdatePosition might simply accept those updates (or read from a shared data structure). Essentially, if

a platform's `motion_source` is marked as external or unknown, we either treat it as static or allow manual updates. This sets the stage for later integration of live data (like ADS-B for aircraft) [13].

The Orbit Propagation module itself acts as a **tick listener** on the Time Controller. On each tick event, it will: 1. Iterate over all PlatformDefinitions in the Knowledge Base (or maintain its own list of platform IDs). 2. For each platform, call the associated MotionModel's `UpdatePosition(currentSimTime, platform)`. 3. Write the updated coordinates back to the PlatformDefinition (and thus into the Knowledge Base). This might be done via a `KnowledgeBase.UpdatePlatformPosition()` call to ensure thread safety and to trigger any observers [8]. 4. (Optional) If certain significant events occur (e.g. satellite passes over a ground station), the module could emit events. In Scope 1, we will defer event logic; instead, we ensure positions are correct, and we rely on future connectivity logic to consume them.

SGP4 Integration Strategy: We will use a well-tested SGP4 implementation to calculate orbital positions [14] [12]. Options include: - Using a Go library (if available) for SGP4/TLE propagation. - Wrapping a C/C++ or Python implementation via cgo or gRPC if needed (since accuracy is crucial). - As an interim step, for testing, we might use the Python `sgp4` library (via cgo or by generating an ephemeris table) – but ultimately, a direct Go integration is preferred for performance [15].

The integration will parse the TLE strings (two-line element set defining the orbit) for each satellite platform at initialization. Each PlatformDefinition with orbital motion will store either the TLE lines or the parsed orbital elements. The MotionModel for that platform will use these to initialize an SGP4 propagator instance. On each tick, we compute the satellite's position at simulation time. We must convert the resulting coordinates into ECEF frame [11] (SGP4 typically yields position in TEME or an inertial frame; we will convert to ECEF so that all platforms share a common frame). The result is set into the Platform's `coordinates` (Motion) field.

Public Interfaces: This module might not be heavily exposed publicly (it works behind the scenes), but for completeness: - `RegisterPlatformMotion(platformID, model)`: Associate a motion model with a platform (called during scenario setup). - `UpdateAllPositions(currentTime)`: Method called on each tick to update every platform. - For testing or external updates, possibly `SetTLE(platformID, tle1, tle2)` to update a platform's orbit data at runtime (so new TLEs can be ingested and the MotionModel for that satellite updated accordingly) [16]. - Utility functions like coordinate conversions (e.g. ECEF <-> geodetic) as needed for output or verification.

Dependencies: Depends on the Time Controller for tick events and on the Knowledge Base for reading initial states and writing updates. Also depends on the SGP4 library and potentially a small math/geo utility (for coordinate transforms, vector calculations). This module should be modular so that new motion models can be added easily (e.g. if we later incorporate more complex orbital mechanics or launch trajectories) [17].

Data Models and Enums (Scope 1 Entities)

Scope 1 defines two primary entity types: **PlatformDefinition** (physical platform) and **NetworkNode** (logical node). These correspond closely to Aalyria's proto messages and will include fields aligned with those definitions. We also incorporate relevant enums and sub-structures (like motion sources, routing settings) to ensure compatibility and future expansion. The tables below enumerate the required fields, their types, and their purpose/rationale:

PlatformDefinition Data Model

Field	Type	Description / Rationale
id	string	Unique platform identifier (primary key). Not explicitly in Aalyria's `PlatformDefinition` proto message, but required in our simulation to uniquely reference the platform [4]. This ID will be used to link nodes to platforms and to query the platform in the KB. In Spacetime's NBI, the `Entity.id` plus an EntityType designates a platform [18] [19]. Our simulation can generate or use provided IDs.
name	string (optional)	Human-readable name for the platform [20] [21]. Used for logging, UI, or debugging. Example: `"Satellite-A"` or `"GroundStation-1"`.
type	string (optional)	A type label or category tag for the platform [20] [22]. This can describe the role or kind of platform (e.g. `"SATELLITE"`, `"GROUND_STATION"`, `"AIRCRAFT"`). In Aalyria's proto, `type` is a freeform string often used similarly to category (for UI or logic tags) [20]. We will use it to classify platforms, which can trigger domain-specific behavior (e.g. a `"SATELLITE"` might require an orbital motion model, a `"GROUND_STATION"` remains fixed) [23].
category_tag	string (optional)	An additional tag or category classifier [24]. Could be used for grouping or styling in a UI (not heavily used in core logic for now). We will accept and store it for completeness.
coordinates	**Motion** (structure)	The platform's position and orientation state in ECEF frame [25]. This can include fields like ECEF position (x, y, z in meters) and orientation (e.g. quaternion or Euler angles relative to ECEF axes). In Aalyria's `PlatformDefinition`, this is a `Motion` message which may contain position and possibly velocity/orientation [25]. In our implementation, we can define a `Motion` struct (with fields like `X, Y, Z float64` for coordinates and perhaps `Quaternion` for orientation). The **orientation** portion will initially be kept default (e.g. identity orientation, or pointing to Earth's center for satellites) since we are not simulating attitude in Scope 1. The coordinates will be updated every tick for moving platforms [11].

Field	Type	Description / Rationale
motion_source	enum `MotionSource`	Indicates how the platform's motion is determined [26]. Values align with Aalyria's proto: e.g. `UNKNOWN_SOURCE (0)` for manual or unspecified, `SPACETRACK_ORG (1)` for automatic TLE-based updates, `FLIGHTRADAR_24 (2)` for aircraft via ADS-B feed [27]. In our simulator, this field is mostly informational in Scope 1 (we won't automatically fetch external data in this phase), but we will set it accordingly. If `motion_source == SPACETRACK_ORG`, it implies we expect to use TLE propagation for this platform (which we do). If an aircraft, we might set `FLIGHTRADAR_24` to indicate a live feed could be used later [27].
motion_ref_id	string (optional)	Reference to a `MotionDefinition` ID if an external motion profile is used [28]. Aalyria uses this to decouple motion data from the platform (advanced usage). For Scope 1, we do not implement separate motion definitions; if provided, we can log or ignore it. Our simulation will primarily use the embedded `coordinates` or TLE. We'll store this field to remain schema-compatible, but if it's set, we will treat it as "external motion provided" and likely not use the static coordinates (per Aalyria semantics [28]). Possibly we interpret it as a signal that an external source will drive the position, so our Orbit module could skip updating this platform unless externally triggered.
norad_id	uint32 (optional)	The NORAD catalog ID of the satellite [29]. If `motion_source` indicates TLE updates, this ID identifies which satellite to fetch updates for. In our simulator, we don't auto-fetch TLEs in Scope 1, but we store norad_id for completeness and potential verification (we could use it to cross-check the TLE corresponds to this NORAD ID).
transceiver_model	list of TransceiverModel	(Stub for Scope 1) List of radio transceiver hardware models on this platform [30]. We will include this field to accept scenario data (if provided), but we do not simulate any transceiver behavior in Scope 1. It's a placeholder for later use in link budget calculations. We can keep it as a list of structs (with fields like frequency bands, gain, etc., as defined in Aalyria API). For now, these will just be stored and potentially logged.

Field	Type	Description / Rationale
bent_pipe_payloads	list of BentPipePayload	(Stub for Scope 1) Any bent-pipe repeater payloads on the platform [31]. Similar to transceiver models, we will store this info if provided, but not actively used yet. This could represent, for example, a satellite's relay payload. No processing logic in this phase.
adsb_transponder	AdsbTransponder (optional)	(Aircraft-specific, stub) If the platform is an aircraft, this may contain ADS-B information (like 24-bit address, flight number) [32]. We will include the structure to be schema-complete. In Scope 1 it's not used unless we integrate live ADS-B later. For now, likely not present in typical LEO/GEO scenarios.

Rationale: The PlatformDefinition model encapsulates all physical attributes of assets like satellites or ground stations [33] [2]. We include fields even if they are not fully utilized in Scope 1 to align with the Aalyria API and make sure the data model is ready for later scopes (e.g. transceiver details for link simulation in Scope 2+). Many of these fields will be accepted and stored *as placeholders*, per the Scope 1 simplifications: e.g. payloads, power, storage are recorded but not yet enforced [34] [35]. The critical fields for Scope 1 operation are the ID, name/type, and coordinates (with motion parameters) [4], which enable the simulation to track each platform's position over time.

NetworkNode Data Model

Field	Type	Description / Rationale
id (node_id)	string	Unique identifier for the network node [36] [37]. This is analogous to `node_id` in Aalyria's NetworkNode proto. It distinguishes the node in the Knowledge Base and is used to refer to the node in APIs or when linking to interfaces, etc. In our design, node IDs might be distinct from platform IDs (they exist in a separate namespace since a platform and node can share a name but represent different layers).
name	string (optional)	Human-friendly name of the node [38], e.g. `"SatA_CommsSubsystem"` or `"Gateway01 Router"`. Just for display/logging.
type	string (optional)	Node type label or role, similar to platform type [39]. We use this to classify nodes (e.g. `"SATELLITE_NODE"`, `"GROUND_STATION_NODE"`, `"USER_TERMINAL_NODE"`, `"INFRASTRUCTURE_NODE"`). This is not an enum in the proto, just a string tag. Scenario definitions often use such tags for clarity [40]. Our simulator can use this field to decide default behaviors: e.g. a `"SATELLITE"` node might automatically expect an associated moving platform, whereas a `"GROUND_STATION"` node might default to static platform. It's mainly metadata.

Field	Type	Description / Rationale
category_tag	string (optional)	Another categorical tag for the node [41]. Like platform's category, it may be used for grouping or visual cues. We include it for completeness, though it's not crucial in simulation logic.
platform_ref	string (optional reference)	*Not an explicit proto field, but added in our model:* The ID of the PlatformDefinition that this node is attached to (if any). In Spacetime, a NetworkNode can optionally be associated with a platform (physical host) [42] [7]. Since the proto doesn't include a direct field for platform ID, our Knowledge Base will establish the link. We add a reference in our Node object for convenience (so `node.platform_ref = platformID`). If set, this implies the node's position is inherited from that platform's coordinates. If a node has no associated platform (e.g. a cloud/ground router not tied to a location), `platform_ref` may be empty and the node is considered non-spatial in Scope 1.
routing_config	RoutingConfiguration (optional)	Networking routing parameters [43] [44]. We include a sub-structure similar to the proto: it may have fields like `router_id` (an IP routing identifier or ASN), `node_sid` (Segment ID for segment routing), etc. These are **placeholders** in Scope 1 – we will parse/store them if provided but not perform any routing logic [45]. The structure might also hold an `ek_route_fn` (external routing function reference) as in the proto. For now, this info is just kept for future use (routing comes in later scopes).
subnet	list of string (CIDRs)	List of IP subnets or prefixes served by this node [46]. If the node acts as a gateway, it might announce these subnets. We store these for completeness; no behavior is attached in Scope 1. They could be used in later networking logic or just documentation of what networks the node connects.
interfaces	list of NetworkInterface	The network interfaces attached to this node [47]. Each **NetworkInterface** would be a structure modeling a port or transceiver: fields include `interface_id` (unique per node), `name` (label), `ip_address` (if any), `ethernet_address`, and possibly mode flags etc [48] [49]. In Scope 1, we will define the data structure and accept interface definitions from scenario input, but we do **not** simulate any traffic or link behavior yet. The presence of interfaces is mainly to prepare for Scope 2 where links between interfaces will form connectivity. We ensure each interface has an ID and can be referenced, and that for now they remain static entries.

Field	Type	Description / Rationale
agent	SdnAgent (optional)	SDN agent configuration for this node [50] [51]. If the node runs an Aalyria SDN agent, this submessage defines its control-plane parameters. We include fields from the proto: e.g. `type` (an enum for the CDPI protocol, like `AIRFLOW` or others [52]), an `ek_sdn_agent` (reference ID if an external agent manager exists), and a map `maximum_control_plane_latency` which maps priority levels to durations [53]. In our simulation, all nodes we control will be considered "SDN-enabled" by default (meaning the controller can send them commands) [54]. We will **note** these fields but assume negligible latency unless configured otherwise (one of the simplifications) [55]. Essentially, we store the map but do not delay any commands in Scope 1 – any control actions are instantaneous (we just log if needed).
power_budget	list of SignalPowerBudget	The node's transmit power constraints [56]. Each entry can specify an Interval (time window, or "infinite" if constant) and an available power in Watts [57]. This allows modeling limited power over time. In Scope 1, we will accept and store any defined power budgets but **not enforce them** [58]. If the simulation were to schedule transmissions, it would ensure not to exceed this, but since we have no transmission model yet, we simply retain this info. If multiple budgets are given, presumably only one is active for the current time – but we won't need to resolve that now. We might log a warning if a scenario is defined such that a node tries to use more power than budget (though without a link simulation, that situation won't really arise).
storage	Storage (optional)	Onboard storage capacity for DTN (Delay/Disruption Tolerant Networking) [59] [60]. This structure has at least one field: `available_bytes` (int64) indicating how many bytes can be stored for store-and-forward operation [61]. If this value is non-negative, it implies the node is DTN-capable (even 0 or 1 byte means DTN support, by proto definition) [62]. We include this to mark nodes that can buffer data. In Scope 1, we will track the capacity (and possibly current usage if we simulate any data flows, which we do not yet) but not simulate DTN protocols. We simply mark DTN support if `available_bytes >= 0` and ensure not to "overfill" in any hypothetical usage (which again, is stubbed out now) [63] [64].

Rationale: The NetworkNode model aggregates networking attributes of a device or endpoint in the constellation [33] [65]. We closely mirror Aalyria's fields to allow seamless scenario definition via their API

or protos. Many fields serve as placeholders in this foundational phase – e.g. routing IDs, interface details, power budgets – because actual routing, link and DTN logic will be added in later scopes [34] [35]. For now, the critical part of a NetworkNode in simulation is its **association with a PlatformDefinition** (if it has one) and its identity. By linking a node to a platform, the node inherits the platform's position in space. This means when a platform moves, any attached node is considered to move with it (e.g. a satellite's node moves as the satellite orbits). If a NetworkNode lacks a platform (e.g. a "Point of Presence" node that represents a remote network with no physical location [42]), it will simply have no positional data – and we treat it as a non-moving, abstract node (it could be assumed at a fixed location or omitted from spatial calculations).

Enumerations and Constants: In addition to the fields above, our implementation will define certain enums and constants for clarity: - `MotionSource` enum for PlatformDefinition (values: `UNKNOWN_SOURCE, SPACETRACK_ORG, FLIGHTRADAR_24, ...`) [27]. - `CdpiProtocol` enum for SdnAgent (values: at least `UNKNOWN, AIRFLOW` as per proto) [52], though we won't act on these beyond storing them. - `NetworkInterface.Mode` enum (with values like `PROMISCUOUS, NON_PROMISCUOUS`) [66] – relevant when we simulate packet reception, but in Scope 1 it's simply stored if provided. - Possible constants for node and platform types (we might define Go `const` or string lists for known types like `"SATELLITE"`, `"GROUND_STATION"` to avoid typos when comparing types).

By structuring data models this way, we ensure **schema alignment** with Aalyria's APIs, making it possible to ingest scenario definitions directly from their protobuf/JSON formats with minimal translation [3]. The data classes can be populated from parsed proto messages (e.g. using Aalyria's definitions in `api-main`), and likewise we could easily produce telemetry or state outputs in the same format. This fidelity to the source schema is intentional to ensure compatibility and to set the stage for implementing the NBI/SBI gRPC interfaces on top of our simulator.

Simulation Time Controller Design

The **Simulation Time Controller** provides a unified mechanism to advance and control simulated time. It supports both real-time progression and accelerated (or stepped) time, as well as pause/resume semantics [67] [9]. Below is the design for this controller:

- **Tick Mechanism:** We use a tick-based loop where each tick represents a fixed simulation time increment (default 1 second of simulation time per tick). In real-time mode, the controller uses a wall-clock timer to emit ticks in real seconds. In accelerated mode, it runs ticks as fast as possible (or on a scaled schedule). For example, if accelerating 10×, each real second corresponds to 10 simulation seconds, so ticks might still emit every real second but the simulation time jumps by 10s each tick. Alternatively, in a *fully offline* mode, we might not wait at all – simply increment time in a tight loop to cover long durations quickly.

- **Real-Time vs Accelerated Mode:** These modes are configured at start. In **real-time mode**, the controller aligns the simulation clock with actual time progression (e.g., using `time.Sleep` or ticker to ensure each sim-second aligns to one wall second) [9] [68]. This mode is useful when the simulation is connected to live systems or for demonstration. In **accelerated mode**, the controller disregards wall clock (aside from perhaps yielding to avoid locking the CPU) and continuously issues ticks until a target simulation end-time or scenario duration is reached. We can also allow an intermediate mode: e.g. *scaled real-time* where you run at 2× or 5× speed, etc., by shortening the sleep between ticks.

- **Pause/Resume Implementation:** The controller maintains an internal state flag. On `Pause()`, it can suspend the tick loop – if implemented with a ticker channel, we can stop or ignore ticks; if a manual loop, we can break out or wait on a condition. Pausing should freeze the simulation's advancement: the current simulation time is remembered, and no further tick updates occur until resumed. On `Resume()`, the loop continues from the last simulation time. We must ensure that when resuming real-time mode, we realign if necessary (the simulation time might *lag* wall time if paused; upon resume, it could either skip ahead to current wall time or simply continue from where it left off – the latter is simpler to implement and avoids time discontinuities in simulation). For accelerated mode, pause/resume is straightforward since wall time is not tied to sim time.

- **Synchronization and Threading:** The time controller will likely run in its own goroutine. It sends tick events (perhaps the current simulation timestamp) to other components. We can implement this with channels (a channel of `time.Time` or a custom TickEvent struct) or with a subscriber list that the controller iterates. Using channels has the advantage of backpressure (if consumers are slow) but could complicate multiple consumers. A simple design: maintain a list of subscriber callbacks and call them sequentially each tick. Given Go's goroutines are lightweight, an alternative is to have the time controller broadcast on a channel and each subscriber runs its own goroutine reading from that channel (via fan-out pattern). For Scope 1, a simpler approach is fine: lock a subscriber list, and on each tick do `for each subscriber: call subscriber(simTime)`.

- **Time Representation:** Internally, simulation time can be represented as an absolute timestamp (e.g. Go's `time.Time`) or as an offset (e.g. seconds since simulation start). It's often convenient to pick a start epoch (possibly the real current time when simulation begins, or a user-provided start time). We will represent simulation time as a real timestamp to easily interface with orbital propagators (which expect epoch-based times for TLE propagation). For example, if a TLE's epoch is in UTC, providing an actual datetime to SGP4 is natural. We may allow the scenario to specify a start time (e.g. "start the sim at 2025-01-01T00:00:00Z") or default to now. The controller will then initialize `currentSimTime = startTime`. Each tick, it increments this by the tick size (e.g. 1 second). This means if accelerated, the `time.Time` will jump faster than wall time. (We should be careful to use UTC and a monotonic source for delta calculations, to avoid timezone or NTP issues.)

- **Controlling Tick Duration:** The tick duration itself could be configurable. While 1-second steps are a good default for many networking scenarios, some use cases might require finer granularity (e.g. if we needed sub-second event timing) or larger steps (for long-term orbit simulation with less detail). We can allow the step to be set (as `step` in Start). For example, a 10-second step would update positions every 10 sim-seconds – which might be acceptable for slower dynamics like GEO satellites. The time controller doesn't inherently know what step is "safe"; it's up to the user/config to choose one that balances simulation accuracy and performance.

- **Event Hook for Each Tick:** After updating the simulation clock, the Time Controller triggers the tick event. We can define that the **Orbit Propagation module** is subscribed to these events. The ordering of updates each tick in Scope 1 is not too critical since only orbits are being updated. In later scopes, ordering might matter (e.g. update positions, then update links, then routing, etc.). For now, the controller could simply notify all subscribers in the order they were added. We'll ensure the Knowledge Base updates from the orbit module are effectively atomic per tick (likely using locking inside KB) so that when tick processing is done, the state is consistent.

- **Termination Condition:** If the simulation is meant to run for a certain duration or until a certain simulation end-time, the controller should check each tick whether the goal is reached (e.g. run for 24 simulated hours then stop). It could also support an explicit `StopTime` configuration or just rely on an external stop call. In interactive use, we expect `Stop()` to be called by the user or test when they want to halt the sim.

Pause/Resume Semantics: Pausing the simulation freezes the simulation time. No tick events will be sent while paused, meaning no platform positions update either. From an external perspective, the whole system appears halted. Resume will continue from the same sim time it paused at (unless configured to catch up). We do **not** auto-adjust for missed real time on resume because that could introduce a large jump; instead, we treat simulation time as a controlled timeline. If real-time syncing is important, an alternate strategy is needed (like when resuming, jump sim time forward by the wall-clock gap to realign). We will document this: e.g. "when paused, simulation time stands still".

Example: If we start at sim time 00:00:00 and run in accelerated mode, after 10 ticks the sim time is 00:00:10. If we then pause for 5 real seconds and resume, the next tick will advance to 00:00:11 (not 00:00:15). In real-time mode, if we pause and resume, we might accept that simulation falls behind real time by the pause duration.

In summary, the Time Controller ensures a **consistent time base** for the simulation and drives periodic updates [69]. Its design emphasizes flexibility (real vs accelerated) and control (pause/resume) so that one can, for instance, fast-forward through a scenario or hold the state at a critical moment for inspection. This component will be implemented first, as other modules (orbit, etc.) depend on it to progress.

Orbit Propagation System

The **Orbit Propagation system** is responsible for computing platform positions over time. It consists of the **SGP4 integration** for satellites and the generalized motion model abstraction that can handle static or externally updated positions. Key aspects of the design:

SGP4 Integration Strategy

We will integrate an SGP4 library to propagate satellite orbits from TLE data [12] [11]. The strategy is to create a thin wrapper around the chosen SGP4 implementation to fit our interface: - At scenario initialization, for each orbital platform we parse the TLE (Two-Line Element set) strings to get an SGP4 "satellite" object. - We store this object (or relevant orbital parameters) in the platform's MotionModel (Orbital model). - On each tick, when `UpdatePosition(simTime)` is called for that model, we compute the position at the given simulation time.

The time passed to SGP4 needs to be in the proper format (usually minutes or days since the TLE epoch). We will likely use the `time.Time` difference between simTime and the epoch of the TLE to get the elapsed minutes. The SGP4 library then returns position (and velocity, if needed) in an Earth-centered inertial frame (TEME). We convert that to ECEF: - We compute the Earth's rotation angle (Greenwich sidereal time) at simTime and rotate the inertial coordinates to ECEF. The details of this conversion can use known formulas (or possibly the library provides ECEF coordinates directly). - Alternatively, use a higher-level library (like Orekit or others) that handles these frames under the hood to directly output lat/long or ECEF.

We will validate SGP4 outputs with known cases (e.g. ISS TLE position at a given time) during testing to ensure correctness. The integration will be encapsulated such that if we later switch libraries (say from a Python binding to a pure Go one), the rest of the code doesn't change – only the MotionModel implementation does.

Performance considerations: SGP4 is very fast per computation (microseconds to milliseconds), so even for large constellations (hundreds of satellites) computing each tick is feasible. We should be mindful if using cgo or external calls to not create excessive overhead per tick (maybe batch updates if needed or reuse objects). For now, a straightforward per-satellite call per tick is fine.

Platform Motion Abstraction (Strategy Pattern)

As mentioned, each platform has a MotionModel. We define concrete strategies: - `StaticMotionModel` – holds a fixed coordinate. Its UpdatePosition simply might ensure the platform's coordinates remain the same. (No-op or maybe just set coordinates to initial in case something overwrote it – but likely no change). - `OrbitalSGP4MotionModel` – holds the parsed TLE and SGP4 context. UpdatePosition computes new coords via SGP4 and updates the platform. This model may also handle if the TLE is updated: e.g. if new orbital elements are fed (perhaps via a future API call), we update the internal SGP4 model. The model could also optionally compute an approximate orientation for the satellite – not required, but we could set the platform's orientation to, say, have Z-axis pointing to nadir (Earth) as a default for completeness. - `ExternalMotionModel` (or **ScriptedMotionModel**) – could hold a trajectory function or dataset. For instance, it may have an array of `(time, position)` waypoints or a function like `pos = f(simTime)`. UpdatePosition finds the appropriate position for the current time. If times between waypoints, it could interpolate or simply hold last known. Another variant is a **LiveMotionModel** that doesn't compute anything but expects an external process to update the platform's coordinates asynchronously (perhaps by calling KB). In that case, UpdatePosition might do nothing – it trusts that the coordinates field is being updated externally. We should support at least a simple version: for example, a *scripted linear motion* – e.g. an aircraft moving along a straight line at constant speed. We can implement that easily to demonstrate non-orbital moving platforms.

The strategy pattern here allows adding new motion types (e.g. a future high-fidelity orbital integrator, or a ground vehicle driving on a map) without modifying the core simulation loop – just by introducing new MotionModel classes and assigning them to platforms [17].

We will provide a factory that, given the platform's data, returns the appropriate MotionModel:

```
func NewMotionModel(p *PlatformDefinition) MotionModel {
    if p.motion_ref_id != "" {
        // Could indicate external motion control; return an
ExternalMotionModel
    }
    switch p.motion_source {
    case SPACETRACK_ORG:
        return new OrbitalSGP4Motion(p.tleData)
    case FLIGHTRADAR_24:
        return new ExternalMotionModel("adsb")
    default:
        // If coordinates are given but no specific source, likely static
        if p.coordinates != nil {
```

```
            return new StaticMotionModel(p.coordinates)
        }
        return new StaticMotionModel(defaultCoordinate)
    }
}
```

This is a rough idea: for now, basically, if a TLE is present (or motion_source indicates orbit), we use Orbital model; if motion_source or type suggests a plane (`FLIGHTRADAR_24`), we might set up a placeholder External model; otherwise default to Static for ground. We interpret `motion_ref_id` as a hint that an external process will update the motion (meaning we assign an External model or something that doesn't propagate on its own).

All MotionModel implementations will have access to the platform object so they can set the platform's `coordinates` (and possibly orientation). Alternatively, they could return a new coordinate and the Orbit module then sets it in the KB. We'll likely do in-place update with proper locking via KB API.

Supporting Different Position Sources

- **Static Platforms:** e.g. a ground station at a fixed latitude/longitude. We will likely input such positions as lat/long/alt in scenario definitions. We need to convert to ECEF coordinates for internal use (since our simulation uses ECEF universally [70] [71]). We'll implement a utility to convert geodetic (lat, lon, altitude) to ECEF (x,y,z) using WGS-84 Earth model. This is done at initialization. After that, the static platform's coordinates (x,y,z) don't change. We may store the original lat/long for convenience or output, but ECEF is authoritative internally.
- **Orbital Platforms:** require initial TLE data. The scenario might directly provide the TLE lines for a satellite. If using Aalyria's API, a PlatformDefinition might not embed the TLE text but could reference a motion source (like SPACETRACK) and norad_id. In our open-source context, we expect the user can supply TLEs. We'll design a way to input them (perhaps in textproto scenario format, include the TLE string or a reference). For Scope 1, it's simplest to allow directly setting the TLE in the config. Our PlatformDefinition class can thus have a `tle_lines [2]string` or similar field (not in proto, but for our internal use) which the simulation uses to initialize the SGP4 model.
- **External/Scripted:** For example, if we have an aircraft, we might not have a TLE but maybe a list of positions over time or a simplistic kinematic model. We can support a basic pattern such as constant velocity motion: if provided a starting point and velocity vector, update position = previous + v*dt each tick. Another example: a circular trajectory for a drone around a point. These can be custom-coded for demonstration. But since the requirements explicitly mention using live ADS-B or predefined trajectories eventually [13], we ensure our design can incorporate that. In Scope 1, perhaps we include a dummy "circular path" model to show extensibility (for testing). Or we simply stub external models by keeping them static unless externally changed.

All positions will be maintained in ECEF coordinates in the PlatformDefinition, which simplifies line-of-sight calculations later and avoids dealing with moving reference frames [72]. Orientation will also be stored in a consistent reference (e.g. as quaternion relative to ECEF axes). If needed, we can provide utility to get a platform's lat/long at any time by converting ECEF to geodetic (for logging or output).

The Orbit Propagation system publishes or pushes the updated positions into the Knowledge Base. After each tick's updates, the Knowledge Base holds the latest coordinates for every platform (and thus implicitly for every attached node). This up-to-date spatial info will later be used by connectivity computations. In Scope 1, we don't have link computation, but we can at least demonstrate retrieving positions (for telemetry output or simple checks like distance between two satellites).

Orientation Handling: While not a focus in Scope 1, we store orientation in the PlatformDefinition's Motion to not lose that detail. For satellites, one might assume they maintain some orientation (e.g. nadir-pointing or some inertial attitude). We will not simulate dynamic attitude changes. If needed, we can set an initial orientation (like identity or a default yaw/pitch/roll). Ground stations might have orientation relevant if they have antennas (e.g. pointing up). We could set ground station orientation to point towards sky (like align local Z-axis with local vertical). But these are cosmetic at this stage; there are no pointing computations yet. We just ensure the field is there for future use (like when modeling antenna gain patterns in a direction, etc.).

Testing the Orbit Module: We will incorporate known test cases such as verifying that a satellite at epoch time is at the expected position (which for TLE is often 0 error by definition). Another test: simulate a full orbit period and ensure the position roughly repeats (for a closed orbit). This will be discussed more in the Test Strategy, but designing the orbit system to be testable means isolating the SGP4 computation (maybe allow dependency injection of a fake propagator for unit tests) and being able to run it for known scenarios.

In conclusion, the Orbit Propagation system ensures each platform's **real-time position is reflected** in the simulation knowledge base [42] [7]. It abstracts the motion logic so that adding a new type of moving object or changing how we propagate orbits doesn't impact the rest of the system. By using strategies and a central update loop triggered by time ticks, we fulfill the requirement of periodically propagating orbits and updating coordinates as time progresses [73].

Knowledge Base (KB) Design and Concurrency

The **Knowledge Base** is the in-memory data store that holds all simulation entities and their latest state. It plays a critical role in Scope 1 by keeping platform positions and node info updated and readily accessible [4]. The design focuses on thread-safe access, efficient reads for possibly frequent queries, and support for change observation.

Data Storage Model

We will implement the KB likely as a singleton struct (or package-level object) for simplicity. Internal data structures include: - `platforms map[string]*PlatformDefinition` - `nodes map[string]*NetworkNode`

These maps allow O(1) lookup by ID. The values are pointers so that updates to a PlatformDefinition (like its coordinates) are reflected wherever that object is referenced. We must be careful with concurrency though – if multiple goroutines access these, we need locking when mutating or possibly when reading mutable fields.

We also maintain linking info: - Each NetworkNode will have a `platform_ref` as described. We may also want a reverse map: `platformToNodes map[string][]*NetworkNode` to quickly find all nodes on a given platform. This is useful if, for instance, when a platform moves we want to update all attached nodes. However, in our current design, a node doesn't have its own position field independent of the platform (we consider node position = platform position if linked). So we might not need to separately update nodes; we can derive on the fly. But for completeness, having that mapping could help if we later need to iterate nodes by platform. - Alternatively, we skip a reverse index and just iterate all nodes when needed (since number of nodes likely in same order of magnitude as platforms, this is fine).

We will also incorporate any necessary indexing or filtering mechanisms for listing, if needed (e.g. get all satellites vs all ground stations – can be done by filtering on the type field, not by separate storage).

No external database is used in Scope 1, but we keep the design such that one could swap the storage with a persistent DB or an in-memory database if needed later (this is low priority now).

Concurrency Handling

To handle concurrency: - We will use a read-write lock (`sync.RWMutex`) to protect the maps. This allows multiple concurrent reads (list, get) without blocking each other, but writes (add, update) will lock exclusively. This is a simple and effective strategy given the operations. During each simulation tick, the orbit updater will perform writes (position updates) potentially to many platforms – we can choose to lock per platform update (fine-grained) or lock once and update all then unlock. The latter ensures a consistent snapshot at the tick boundary but could block readers slightly longer. The former yields more concurrency but with overhead of many lock/unlock cycles. Considering typical usage (reads for queries can happen often, writes happen every tick perhaps hundreds of ms apart in real-time mode or tight loop in accelerated), either approach could work. We lean towards a coarse lock for simplicity: on a tick, lock KB, update all necessary fields, then unlock. In between ticks, reads can happen freely. The overhead is negligible for tens or hundreds of platforms updated per tick. - We must avoid data races: e.g. if a user thread calls `ListPlatforms()` while a tick thread is mid-update. Using locks as above prevents that. In tests, we will use Go's `-race` detector to ensure safety. - Another approach is copy-on-write or snapshotting: for instance, each tick produces a new copy of platform state and swaps it. That's more for functional programming style and not needed here due to overhead. Instead, we ensure proper locking and perhaps document that external consumers should use provided API (which does locking) rather than accessing the maps directly.

For **observing platform/node state**, concurrency is also a concern: if observers are called from within the update (hold lock and call out), that could deadlock or slow things if observers then call back into KB. We must design carefully: - One pattern: collect all changes, release lock, then notify observers. For example, orbit module updates positions for all platforms, it could accumulate a list of "changed platform IDs" or new values, then after releasing KB lock, loop through those and send out notifications. - Or we make a copy of relevant data and send it under lock if sending via channel (to avoid external modification). - The simplest is to do notifications outside the locked section to avoid blocking the KB for long. We'll implement it accordingly: update state under lock, build events, unlock, then deliver events.

APIs for Reading and Writing

We enumerated main APIs under Module Breakdown, but to reiterate in design terms: - **Write Operations:** - `AddPlatform(p)`: Lock KB for writing, ensure `p.id` not exists, then insert into `platforms` map. Potentially also initialize `platformToNodes[p.id] = []`. If the platform has an initial coordinate and motion model, we might also call Orbit module to register it. - `AddNetworkNode(n)`: Lock, check `n.id` unique, insert into `nodes`. If `n.platform_ref` is set, verify the referenced platform exists. Optionally append the node to that platform's list (or just skip if not maintaining reverse map). Also, perhaps set a back-pointer: e.g. platform might list its attached nodes. - `UpdatePlatformPosition(id, newCoords)`: Lock, find platform, update its `coordinates` field. Optionally record a timestamp of last update. Unlock. Then maybe trigger notify. - We may not need a generic `UpdateNode` yet as not much changes for a node in Scope 1 aside from maybe marking storage usage or such (which we are not simulating). - **Read Operations:** - `GetPlatform(id)`: Lock for reading (or use RLock), retrieve pointer. The caller should treat it as read-only (or clone it) to avoid data races; in internal code we might just use pointers and assume no modification outside KB. - `ListPlatforms()`: RLock, copy or slice out all PlatformDefinitions (to avoid

holding lock while external code iterates). Possibly deep-copy if external code might modify, or at least copy pointers. - Similarly, `GetNode` and `ListNodes`. - We might provide convenience queries, like `GetNodePosition(nodeID)`: This would get the node, check if `platform_ref` is set, get that platform's coordinates and return them. This is read-only and can be done under RLock easily. This could simplify external components that want a node's location without doing two lookups. It's not strictly necessary but helpful.

All API methods will ensure thread safety internally. We will document that external users should not bypass these (e.g. not iterate the maps directly) unless they handle locking.

Observing State Changes (Subscriptions)

To allow other parts of the simulator (or even external clients) to react to changes, we will include an **observer/subscribe** mechanism: - An `Event` could be a struct like `{Type string; ID string; OldValue interface{}; NewValue interface{}; Timestamp time.Time}` or simpler, or we can have specific event types (e.g. `PositionUpdated{platformID, newCoord}`). - We'll provide `Subscribe(fn EventHandler, filter FilterOptions)` where FilterOptions might allow subscribing only to certain events (like only position updates, or only certain entity types). In Scope 1, we might not implement filtering beyond perhaps "platform updates" vs "node updates". - Under the hood, we maintain a list of subscriber callbacks similar to the tick listeners. Each time an entity is changed, we notify. However, as noted, we should release the main lock before invoking callbacks to avoid blocking the KB. - If thread-safety of events is a concern (e.g. an event contains a pointer to the new PlatformDefinition), we can copy the relevant data in the event so the subscriber gets an immutable snapshot. But since the update just happened and no further change will occur until next tick, giving the pointer is not entirely unsafe if they only read it. To be safe, though, we might give a copy of the coordinates or so to avoid any concurrency issues if they read while another tick might start updating (in accelerated mode, this could be concurrently happening if the subscriber is slow). Alternatively, document that subscribers should quickly handle the event or make their own copy.

- Potential uses of subscriptions: A logging component could subscribe to all position updates and write them to a file (telemetry output). A future connectivity module could subscribe to platform moves to recalc link geometry immediately (or it might simply run each tick as well – either model works). The subscription model is more asynchronous/event-driven which can be efficient if updates are sparse or irregular, but since ours are periodic, polling each tick is equally viable. Still, we design it for generality.

Consistency and Atomicity

We want the Knowledge Base to present a coherent view of the system state at each simulation time. That means: - During the middle of a tick update (say we have 100 satellites and we've updated 50 of them but not the rest), no external read should see some updated and some not (that would be a half-consistent state for the tick). By locking around the whole update loop for all platforms, we ensure that any read either sees the state before tick or after the tick, not in between [74]. - This atomic tick update might temporarily block reads, but typically a tick update (SGP4 calculations etc.) might take a few milliseconds even for many satellites, which is fine. If it became a performance issue, we could subdivide locks by grouping or allow some stale reads (depending on needs), but likely unnecessary. - If an external thread (like an API call) adds or removes an entity concurrently with a tick, that's another scenario. We can choose to handle scenario changes (like adding a platform during run) carefully. Possibly we freeze adding entities while simulation is running or allow it but with lock protection. For scope 1, dynamic addition is probably not needed (scenario is static once started). If we allow, it just

means the KB lock will serialize those with ticks. We'll mention that adding entities at runtime is supported but not common.

Aligning with Aalyria KB concept

Spacetime's knowledge base concept is that all these entities (platforms, nodes, links, etc.) are accessible and queryable via an API [5] [6]. Our KB is an internal version of that. We'll provide in future scopes an API (gRPC) to query these, but for now, a developer can use the KB's methods directly in tests. The design ensures we can later implement the ListEntities RPC by simply calling `ListPlatforms` etc. and formatting results in the Entity message format.

Example Usage: Suppose we have a simulation with two satellites and one ground station. The orbit module, every tick, updates the satellites' PlatformDefinition coordinates. After each tick, a subscriber (maybe a debug logger) gets events like "Platform X new position: {x,y,z}". Meanwhile, an external query could call `GetPlatform("Sat1")` and get its latest position. The concurrency design guarantees that that query won't see a partially updated position vector (like an X from old tick and Y from new tick – impossible since it's one struct updated in one go under lock).

Garbage Collection: If in later use we remove entities, we need to clean up references. For now, removal is not a primary concern (scenarios are typically fixed set of entities). We may implement a `DeletePlatform(id)` and `DeleteNode(id)` for completeness, which would also remove linkages and notify, but it's optional in Scope 1.

By implementing the Knowledge Base with these considerations, we fulfill the requirement of maintaining an internal knowledge base of all platforms and nodes with their IDs, names, types, and continuously updating coordinates [4]. This KB underpins everything else, allowing other modules to retrieve current positions, and providing a single point to integrate with APIs or UIs for scenario inspection.

Test Strategy

To ensure each part of Scope 1 works correctly and to facilitate Test-Driven Development (TDD), we outline a comprehensive testing approach. Tests will be written for each module (unit tests) and for integrated behavior of the system with multiple entities. Where possible, we will use known reference data (for orbital calculations) to validate correctness.

Unit Tests by Module

Platform & NetworkNode Models: - *Construction and Field Access:* Create sample PlatformDefinition and NetworkNode objects (both through constructor functions and by unmarshaling example proto messages, if available). Verify that all fields are set correctly and can be retrieved. For instance, ensure that if we set a Platform's name and type, those fields read back properly. - *Linking Node to Platform:* Create a Platform and a Node, set the node's `platform_ref` to the platform's ID. In the test, verify that we can resolve that link (e.g. by a helper function in KB or Node like `node.GetPlatform(kb)` returning the correct Platform). We might provide a test helper to simulate attaching a node to a platform and ensure that position inheritance logic will work (see integration tests below for actual movement). - *Default Values:* If certain fields are optional and not provided, ensure they default to expected values. E.g., if no motion_source is given for a satellite but a TLE is present, we might default `motion_source` to UNKNOWN or SPACETRACK. If no orientation is given, ensure orientation is identity. Tests can construct minimal objects and ensure no nil pointers issues etc. - While these data

classes are simple, it's still useful to verify their integrity especially after operations (like no unexpected changes).

Knowledge Base: - *Add/Get/List:* Write tests to add a few PlatformDefinitions and NetworkNodes to the KB and retrieve them. Example: Add Platform "sat1", Node "node1" attached to "sat1", and Node "node2" unattached. Then: - `GetPlatform("sat1")` returns the correct object (same pointer that was added). - `GetNetworkNode("node2")` returns correct node. - `ListPlatforms()` returns a list containing at least the one we added (size matches count). Similarly for nodes. - Test that listing doesn't expose internal mutability issues (e.g., if we modify the returned object, does it affect KB? Depending on implementation, we might have to copy; our test can enforce that if we decide to copy). - *Thread Safety:* We can simulate concurrency in tests by using goroutines. For instance, start a goroutine that continuously reads from KB (get or list in a loop) while another goroutine updates positions or adds entities, and run for some time to see if any data races are reported (Go's `-race` flag will detect concurrent unsynchronized access). We expect none if locks are used properly. - *Subscribe Notifications:* Register a dummy subscriber that records events into a slice. Perform a known update (e.g., add a platform, or update a platform's position through KB API). Verify that the subscriber was called with the appropriate event. For example, update Platform "sat1" position, expect an event of type "PositionUpdated" with ID "sat1". We can also test that multiple subscribers receive events and that unsubscribing stops receiving further events. - *Link integrity:* If we maintain the `platformToNodes` mapping, test that adding a node with platform triggers the mapping update (like KB's internal list of node IDs on that platform includes the new node). If a platform is removed (if we implement removal), test that nodes are handled (maybe also removed or marked stale). - We might also simulate an error case: attempt to add two platforms with the same ID and ensure KB returns an error or panic is avoided. Same for duplicate node ID. Our design said we'd error out; tests should confirm that.

Time Controller: - *Tick timing in Real-time Mode:* This is tricky to unit test precisely without waiting in real time, but we can simulate by using a small tick interval (say 100ms) and measuring intervals. Or use a fake clock if we abstract the time source (could be overkill). At least, we test that after `Start(realTime=true)` with a given tick interval, the controller produces roughly the expected number of ticks in a given time. We can allow a tolerance since OS scheduling might not be exact. - Alternatively, abstract the time dependency using an interface so we can inject a deterministic ticker in tests. For example, an interface `Ticker { C() <-chan Time; Stop() }`, which normally wraps `time.Ticker` but in tests we use a manual channel to send ticks. This adds complexity, though. - Simpler: run `Start(realTime=false)` (accelerated mode) with a loop for N ticks and ensure it completes in much less wall time than N seconds (meaning it's faster than real-time). - *Accelerated progression:* Configure a simulation start at time T0, with accelerated mode. After X ticks, check that `Now()` returns T0 + X*step. That ensures the simulation clock is advancing correctly. For example, start at 00:00:00, tick 10 times with step 1s, confirm Now() == 00:00:10.* - Pause/Resume: *Start the controller (maybe accelerated to avoid waiting). Let it tick a few times, call Pause(), wait a short period (or a few loop iterations), ensure no ticks were emitted during the pause (we can have a counter increment in the tick listener to see if it stopped increasing). Then call Resume(), ensure ticks resume and the simulation time continues from where paused (not jumped). Specifically, record sim time at pause, then after a couple ticks post-resume, confirm it equals pause_time + couple*step. - *Subscriber management:* If using callbacks for ticks, test that adding a listener works (it gets called), and removing it stops receiving calls. Could add two listeners that append to two lists, run a few ticks, and ensure both lists have entries. Then unsubscribe one, run more ticks, ensure one list stops growing while the other continues. - Also test `Stop()`: after stop, no further ticks happen even if we try to resume (stop might implicitly pause permanently). Possibly also ensure that calling Stop properly terminates the tick goroutine (which is hard to test directly, but we can test that a listener no longer gets ticks after stop). - If time controller uses an internal channel, we test no resource leaks (goroutine should exit on Stop; we can use a sync.WaitGroup or a channel to signal done and wait in test to verify).

Orbit Motion Models: - *Static Model Test:* Create a StaticMotionModel with a known coordinate (say ECEF for some point). Call UpdatePosition at a series of times (or in a loop) and verify that the platform's coordinate remains exactly the same. No floating drift, etc. If orientation is part of Motion, verify it stays same too. - *Orbital Model Test (Basic):* This requires a sample TLE. Use a well-known TLE (e.g. for the ISS). Use our OrbitalSGP4MotionModel to propagate to two different times. If possible, cross-check the results with an authoritative source: for instance, the SGP4 library itself, or published position. One approach: the Python `sgp4` library or another reference can give position which we embed as expected result. Or simpler, ensure that two calls yield two different positions and the difference is plausible. For example, propagate at epoch (should match the TLE's stated position vector if we could get it) and epoch + 1 minute – the ISS moves ~7.5 km/sec, so ~450 km difference; we can check the distance between points is ~450 km. Also check something like altitude remains ~400 km (by computing sqrt(x^2+y^2+z^2) - EarthRadius). - Another test: propagate a full orbital period in steps and ensure the path is continuous (we could check that no two consecutive points are too far apart given a small step, indicating continuity). - Edge cases: propagate far from epoch (e.g. several days beyond TLE epoch, though SGP4 accuracy might degrade – but just see that it returns something reasonable). Also test behavior if the TLE is malformed (we should handle parse errors gracefully – maybe our model constructor returns an error which KB or scenario loader would catch; we test that error path). - *Orbital Model vs Static Check:* If a platform is erroneously set with static model but it has a TLE, obviously that's a misconfiguration. We can have tests to ensure our factory chooses correct model given inputs: e.g. provide a PlatformDefinition with `motion_source=SPACETRACK` or a TLE, verify `NewMotionModel` returns an OrbitalSGP4MotionModel, not a Static. - *External Motion Test:* If we implement a simple trajectory model (say linear motion), test that over ticks it produces expected positions. For example, start at (0,0,0) with velocity (1,0,0) m/s; after 10 one-second ticks, expect position (10,0,0). If using Earth surface points, maybe test an aircraft moving east at certain m/s and check after known time. - If we implement a model that reads from a predefined waypoint list: simulate a list and ensure interpolation works. - If we have a Live external model stub (that does nothing on Update), we can test that external update actually changes position. For example, set up an ExternalMotionModel, manually call KB.UpdatePlatformPosition to mimic an outside update, then verify that the position changed even though UpdatePosition did nothing on that tick (meaning the KB update itself is what changed it, which is expected).

- *Precision and Frame:* A subtle test is to ensure Earth-centric distances are correct. For one, test that a static ground station at lat/long (0,0) alt 0, after converting to ECEF, has magnitude ~Earth radius (~6371 km). If altitude = 0, |ECEF| should equal Earth radius. For another location, maybe test the convert latlon->ECEF->latlon returns the original (within tolerance). This validates our coordinate conversion.
- *Orientation consistency:* If we set orientation default (like identity quaternion), test that it remains a unit quaternion if we propagate (should, since we likely never change it). If we ever decide to set an orientation for satellites (say pointing to Earth), we could test that the satellite's orientation vector is indeed pointing roughly towards Earth's center vector (dot product positive, etc.).

Integration Tests (Multi-Component)

Scenario Simulation Test: Construct a small scenario programmatically: e.g. - 2 satellites (Sat1, Sat2) with known TLEs (perhaps different orbits), - 1 ground station (GS1) at a fixed location, - 1 network node per platform (Node1 on Sat1, Node2 on Sat2, Node3 on GS1). Start the simulation (maybe accelerated to run quickly) for a certain number of ticks (say simulate 1 hour with 10-second ticks, meaning 360 ticks). During the run, collect data and then assert certain conditions: - Sat1 and Sat2 positions change over time in a smooth orbit. For example, record their altitude at each tick and ensure the variation is within expected bounds (i.e., roughly periodic for an orbit). Specifically, altitude should remain roughly

constant for a circular orbit or oscillate between perigee and apogee if elliptical, but not drift unboundedly. - The distance between Sat1 and GS1 should drop below a threshold at some point if the orbit passes overhead (if we choose TLE such that it does). Or simpler: check that GS1 coordinates remain constant and Sat1's coordinate moves; maybe find the minimum distance between Sat1 and GS1 over the hour and assert it's reasonable (like not zero unless we expect a pass directly overhead). - Verify that Node1 (on Sat1) always "follows" Sat1's platform position. Since we might not explicitly store node positions, this can be indirectly tested: we know Node1's platform_ref = Sat1. We can query KB each tick for Node1's platform's coordinates and ensure it matches Sat1's coordinates (basically a consistency check in code rather than a property of the code – because by design it should always match as it's the same object – but this test just ensures our node linking logic isn't broken). - If we had any external update, e.g. we could simulate an external event: after half the simulation, manually update GS1's position (imagine we "moved" a ground station, even though unrealistic). Then ensure subsequent ticks respect the new position. But moving ground station might not be a normal operation – perhaps skip this scenario unless testing external injection. - Ensure no platform's position goes NaN or Inf (this can be tested by checking all coordinates are finite numbers after simulation). - The Knowledge Base at end of simulation should list exactly those 3 platforms and 3 nodes we created, with final positions. - If logging enabled, perhaps verify that some expected log entries appear (e.g. "Tick count" or certain events).

Time Controller & Orbit Integration: We should test that the time controller and orbit propagation work in tandem. For example: - Use a short simulation with known tick count and ensure that orbit updates happened that many times. We can insert a counter in the MotionModel or use log events: e.g. have the orbit model log each update. Then assert that the count of updates equals number of ticks (for orbital and static models alike). - Test pause with orbit: e.g. run simulation, pause, during pause maybe manually change something or wait, then resume, ensure that during pause the orbit did not update coordinates (i.e., positions remained frozen), and upon resume they continue from where they left off. This might involve checking that the coordinate at pause and after a pause gap (with no ticks) is identical, then after resume the next coordinate is updated normally.

Validation Strategies (Physical & Logical Invariants): - **Altitude Bounds:** For each platform with an orbital model, verify that its distance from Earth's center is within a reasonable range based on its orbit. If we know the TLE or orbit, we can compute expected perigee/apogee. Or at least check it stays above Earth's radius (no decaying below ground) and not way too high (unless it's supposed to – but if we use a known LEO TLE, altitude ~ 6771 km from center for ISS, check it stays ~6771 ± maybe few tens of km). If we see a huge drift, it could indicate a time or frame error (like if we forgot to convert time units properly). - **Ground Station Position Drift:** Check that any static platform's coordinates do not change throughout the simulation (a simple equality check initial vs final). This ensures our static model truly doesn't introduce numerical drift. - **Orbital Periodicity (for regression):** If feasible, simulate a satellite for the duration of one orbit (e.g. ~90 minutes for LEO) and verify that its final position is close to initial (SGP4 won't be exact due to perturbations not modeled in simple two-body, but it should be quite close). This can catch errors in time stepping or using wrong epoch: e.g. if we accidentally advanced time incorrectly, after one orbit the satellite might not come back around as expected. - **Node Integrity:** If a node is not attached to any platform, confirm that it being "isolated" doesn't break anything. For example, one of our scenario nodes might have no platform (like an infrastructure node). Ensure that simulation runs without needing to update or access a null platform. Essentially, confirm that our code either skips or handles nodes with no platform gracefully (they simply have no position update). We might assert that such a node remains present in KB and can be listed, and maybe has some default state. If we decide to give unlinked nodes a dummy static coordinate (maybe not), then test that if we do, it remains constant. - **Time Controller Accuracy:** For long simulations in accelerated mode, ensure that the number of ticks * tick duration equals the total simulated time elapsed (within off-by-one if you count final state). This is more of a consistency check on time progression. For example, run 1000 ticks

of 1s step, verify simulation clock advanced 1000 ±1 seconds. This catches any cumulative error in how we increment time or if we accidentally dropped/added an extra tick.

- **Resource and Performance Checks:** Not a strict requirement, but we can include tests or at least measurements to ensure that adding more platforms linearly increases tick processing time and no memory leaks. For instance, simulate 1 satellite vs 100 satellites for a few ticks and measure time per tick to see if roughly linear. Also, run a simulation for many ticks and ensure memory usage is stable (no unbounded growth – implying no accumulating slice or event backlog). This can be done manually or with test instrumentation rather than an automated unit test.

Test-Driven Development Approach

We will write tests alongside implementing the modules, often writing the test first (e.g. define expected behavior of TimeController in a test, then implement minimal code to pass it). This ensures each feature is well-defined and verifiable. For example, before coding the pause logic, write a test that calls pause and expects no further ticks, etc., then implement until it passes. This approach is feasible because our modules have clear, separable responsibilities: - We can use a fake tick emitter to test Orbit Propagation in isolation (simulate calling UpdatePosition with incrementing times). - We can simulate a fake MotionModel to test the Time Controller's callback mechanism (e.g. a dummy listener that flips a flag on tick). - We will create stub SGP4 outputs for testing if needed (to avoid dependency on actual orbit library in pure unit tests – but integration tests will use the real library to ensure it works as expected with actual data).

By thoroughly testing each part, we can confidently combine them. When we run full scenarios in integration tests, any discrepancy (like a node not updating or a data race) should surface, and our unit tests should help pinpoint which module might be the culprit.

Error Handling, Metrics, and Logging

Even in Scope 1 (with no external networking effects yet), robust error handling and observability are important. We outline the design considerations for how the system will react to errors and produce logs/metrics:

Error Handling Design

- **Initialization Errors:** These include invalid scenario configurations. For example, if a NetworkNode references a platform ID that is not defined, the simulator should log an error and either (a) create the node without a platform (assuming maybe it's intentional but platform definition missing) or (b) fail scenario loading. We likely treat it as a scenario definition error and halt initialization with an error message to the user (since it's better to catch such mismatches early). This could be implemented by the scenario loading function validating that every `node.platform_ref` has a matching platform in KB.
- Other init errors: Malformed TLE string (cannot be parsed by SGP4). In that case, for Scope 1, we have two options: fail the simulation start for that satellite (because orbit can't be propagated), or mark that platform as static (no motion) as a fallback. It's safer to notify the user. We will likely propagate the error (e.g. the function that adds the platform returns an error "Invalid TLE for platform X"). The user (or higher-level API) can decide to remove or replace it. Logging this clearly is important too.
- **Runtime Errors:**

- **SGP4 Propagation Errors:** SGP4 itself typically doesn't "throw" errors for time ranges, but if we propagate extremely far from epoch, the results might be nonsense. If the library indicates an error (some libraries provide an error code if orbit decay or bad epoch), we should catch that. At runtime, if a propagation fails, what to do? We can log a warning "Propagation failed for SatX at time T (maybe TLE outdated)". We might then skip updating that platform for this tick (leaving its last known position). Since the simulation can continue without one satellite updating (though connectivity results might be off for it), we prefer to continue rather than crash. So catching exceptions or error codes from the propagator and converting to log messages is the plan.
- **Time Controller Errors:** Not many – perhaps if an invalid configuration is given (negative tick interval or scale), we can default or error. If the ticker goroutine panics (it shouldn't normally), we wrap it in a recover to log and stop gracefully.
- **Concurrent Modification Safety:** If someone tries to modify a PlatformDefinition from outside the KB (which is against design), it could cause issues. We cannot easily prevent a determined user messing with pointers, but in our own code, we avoid that. We rely on code discipline rather than try to enforce immutability of PlatformDefinition (could do with copying, but that's heavy).
- **Interface and Unused Field Handling:** If scenario inputs contain fields we don't support (like trying to set an interface's promisc mode or a transceiver frequency), we will accept them but log a debug note that they are not used. This way the user knows those fields are essentially ignored in the simulation. E.g., "NOTE: power_budget defined for node X is not enforced in this simulation scope [58] ".

- **Out-of-scope Actions:** If an API call tries to do something not available yet (e.g. call a connectivity query in Scope 1), we might return a "Not Implemented" error or stubbed result. For internal design, since we haven't implemented those features, we likely won't have such API to call at all, but it's something to keep in mind as we develop further scopes.

- **Graceful Degradation:** Because we want the simulation to keep running even if some parts fail, we adopt a strategy of *failing soft* where possible:

- If one platform's update fails in a tick (e.g. one bad TLE), we handle that error, mark that platform as "failed this tick" (maybe keep its old position) and continue updating others. We then also possibly raise an event or flag that something is wrong with that platform. The simulation continues and others are fine.

- If Knowledge Base experiences an error (like an impossible condition), that's more severe. But our KB logic is straightforward; main possible error is duplicate IDs or missing references, which we handle at scenario load.

- **Panics and Recovery:** In Go, we try not to panic except on truly unrecoverable issues. But since this might run as a long-lived service, we could use recover in the top-level simulation loop to catch any panic, log it, and perhaps stop the simulation gracefully. For example, wrap the tick loop in a defer-recover so if any unforeseen bug causes panic, we output stack trace to log and terminate the sim rather than crash the whole process (especially if integrated in a server). For development, we might let it panic so we notice and fix, but the design acknowledges the need for resilience.

Metrics

We will integrate basic metrics instrumentation so we can monitor performance and state: - **Tick Rate Metrics:** e.g. a counter for ticks executed, perhaps measure tick processing duration (histogram or gauge for last tick duration). This can show if tick processing is real-time safe or if accelerated mode is

hitting any limits. - **Platform/Node Counts:** Metrics for number of platforms and nodes loaded (and maybe subdivided by type, e.g. number of satellites vs ground stations). These can be set at scenario load or updated if dynamic. - **Propagation metrics:** If using Prometheus or similar, we might add a counter for propagation errors or out-of-bounds (to alert if, say, many TLE errors occur). - **Latency metrics for control-plane (though we assume negligible in Scope 1) –** might skip now, but place hooks for later (like measure how long it takes from issuing a command to node acknowledging, if we had that). - **Resource usage (if we had internal instrumentation):** We can measure memory usage periodically or just rely on external observation. Not typically part of app metrics, but we can track, for example, the length of event queues if any (ensuring we're not building up backlog).

Given that we might use Prometheus client in Go as suggested [75], we would design our code with metric collection in mind. For example, after each tick, update a gauge `sim_time_seconds` = current simulation Unix timestamp (so external observer sees what sim time it is now). Or count `sim_ticks_total++` each tick. Also measure and record if any tick fell behind schedule in real-time mode (if we find that the loop couldn't keep up real-time, that could be a metric or log warning).

Logging

Logging will be designed to be **structured and with adjustable verbosity** [75]: - We will likely use a structured logger (in Go, e.g. `logrus` or `zap` or even the built-in with some format) that can output key=value pairs. Each log entry can include simulation context like current sim time, module, entity IDs, etc. - **Startup logs:** On simulation start, log the scenario summary: number of platforms, nodes, start time, mode (real-time or accelerated), tick interval. - **Per-tick logs:** We will *not* log every tick at info level (that would be too verbose at 1Hz). We can log every Nth tick or at debug level if needed. Instead, key events are logged: - If running real-time, and a tick was delayed (maybe GC pause caused a delay), log a warning "Tick X delayed, simulation falling behind real-time by Y seconds" – just for performance monitoring. - We can log when transitioning states: e.g. "Simulation paused at sim time T", "Simulation resumed", "Simulation stopped". - **Platform updates:** We may enable a debug log that logs new position of each platform each tick, but that's huge for many platforms. Instead, perhaps log at info when a platform is added or removed, and log at debug a formatted position occasionally. Alternatively, rely on telemetry output rather than logs for continuous data. For Scope 1, maybe an option to turn on orbit debug logging. - **Event logging:** If a notable event occurs (like a satellite passes over ground – but we aren't computing that yet in Scope 1), we might log it. For now, events might be: - Detection of an error as discussed (invalid TLE, out-of-range propagation). - When hitting a stubbed feature: e.g. if a node's power budget is exceeded. In Scope 1 we're not actively computing power usage, but say we had a test where we pretend a node uses some power and it goes over budget – we would log `"Power budget exceeded for Node X: using YW > budget ZW (not enforced)` [58] `"`. This aligns with the plan to **log or ignore** in stubbed cases rather than enforce [58]. - Similarly, if storage capacity would be exceeded, we might log a warning (though we have no data flow to cause that yet). - **Subsystem tags:** Each module can log with a tag, e.g. `[TimeCtrl] Simulation started`, `[Orbit] Sat1 position updated lat=..., lon=...`, `[KB] Added Platform Sat1`. This helps filter logs by module. - **Logging SDN agent info:** As all nodes are considered SDN-enabled by default, we could log that once: e.g. "Node X SDN agent type: AIRFLOW, control latency: none (negligible in this scope)" to acknowledge the config [54] [76]. - **Configuration logging:** Dumping the initial scenario config in the log can help debugging. Perhaps log each entity loaded (id, type, any special fields like altitude or TLE snippet). - **Error logs:** Any caught exception or error should be logged at WARN or ERROR level with enough detail. E.g., if SGP4 returns an error code, log "ERROR: Failed to propagate orbit for [Sat1] at time … (possibly decayed)". - **Metrics logging:** If not using Prometheus in a test environment, we might periodically log metrics ourselves. For example, each minute of real time, log how many ticks executed, or how many seconds of sim time covered per real second (useful in accelerated mode to see speed-up factor achieved).

All logs should avoid flooding. We will use levels appropriately (DEBUG for per tick details, INFO for high-level progress, WARN for recoverable issues, ERROR for severe issues). We also ensure that sensitive or large data (like full interface lists) aren't dumped repeatedly.

In summary, the design for errors, metrics, and logging is to **fail loudly on config errors**, **fail softly on runtime errors**, and to provide transparency through logs and metrics about the simulation's operation. This will greatly assist a developer or user in understanding what the simulator is doing and in diagnosing any issues that arise.

Sample Scope 1 Scenario Definitions

To illustrate the capabilities of Scope 1 (Core Entities & Orbital Dynamics), here are example scenario components that a user might define and the expected simulation behavior. This scenario includes a mix of Low Earth Orbit (LEO) satellites, a Geostationary (GEO) satellite, and ground assets:

- **LEO Satellite – "SatLeo1":**
  PlatformDefinition: ID `"sat_leo_1"`, name `"LEO-1"`, type `"SATELLITE"`. Motion defined by a TLE (two-line element set) for a ~500 km altitude polar orbit. For example, a TLE indicating ~95-minute orbital period, inclination ~90°. We set `motion_source = SPACETRACK_ORG` (meaning we use TLE propagation) [27] and provide the TLE lines in the config. Orientation is left default (nadir-pointing assumed).
  NetworkNode: ID `"node_leo_1"`, name `"LEO-1 Node"`, type `"SATELLITE_NODE"`. This node is attached to platform `"sat_leo_1"` (so `platform_ref = "sat_leo_1"`). It might have a routing_config with a router ID (placeholder), and perhaps a power_budget of e.g. 100 W constant (we include this in the definition to demonstrate the field, though we won't enforce it) [58]. Storage could be set, say 1 MB (`storage.available_bytes = 1e6`) to mark DTN capability (again not actively used yet, but noted) [63]. The node could also have one or two NetworkInterface entries (say an RF interface for space links and maybe an optical interface as a placeholder) – but no links are defined in scope 1, so these serve as stubs.

Expected Behavior: The simulation will propagate "SatLeo1" along its orbit. Every tick, its ECEF coordinates update, moving rapidly around Earth. Over ~95 minutes of sim time, it completes a full orbit. The attached node "node_leo_1" effectively moves with it (any query for node's position at time T would fetch the platform's coordinates). No connectivity is calculated, but we can later use its position to determine when it passes over ground stations, etc. If power budget or storage are exceeded in any hypothetical operation, we would just log a warning (but since no traffic in scope 1, that likely won't occur).

- **LEO Satellite – "SatLeo2":**
  Another LEO satellite to demonstrate multiple orbiting objects. Platform ID `"sat_leo_2"`, perhaps different orbital plane (e.g. inclination 45°, or different RAAN). Provide a TLE for this as well (maybe a slightly higher orbit ~700 km).
  NetworkNode ID `"node_leo_2"` attached to `"sat_leo_2"`. Similar settings: type "SATELLITE_NODE", maybe a different router ID. This node's presence shows we can handle multiple moving platforms concurrently. Both orbits will be propagated each tick.

- **GEO Satellite – "SatGeo1":**
  PlatformDefinition: ID `"sat_geo_1"`, name `"GEO-1"`, type `"SATELLITE"`. Motion via TLE for a geostationary orbit (approx 35,786 km altitude, 0° inclination). The TLE would have period ~24h. Set `motion_source = SPACETRACK_ORG` as well.

NetworkNode: ID `"node_geo_1"`, name `"GEO-1 Node"`, type `"SATELLITE_NODE"`. Attached to `"sat_geo_1"`. This node might represent a bent-pipe transponder, for example. We could give it a bent_pipe_payload in the platform definition to illustrate that field (e.g. one payload with certain frequency bands). Again, in Scope 1 that's just stored data.

Expected Behavior: "SatGeo1" will appear nearly stationary relative to Earth in the simulation (its ECEF coordinates might change slightly due to orbital eccentricity or drift, but ideally if we use a perfectly geostationary TLE, it will remain roughly fixed over a longitude). This tests that our propagation can handle 24h period orbits and that the time step (if 1s or so) doesn't accumulate significant error for GEO. The node "node_geo_1" remains at that fixed location as well. If we had a ground station under it, we'd later see constant coverage, but that's for later scopes.

- **Ground Station – "GS1":**
  PlatformDefinition: ID `"gs_1"`, name `"GroundStation-1"`, type `"GROUND_STATION"`. This platform has a static position. For example, lat = 0°, lon = 0° (on the equator near Gulf of Guinea), altitude = 0. We convert that to ECEF coordinates at init. We set `motion_source = UNKNOWN_SOURCE` (since it's manually positioned) or we could leave it unset. This platform won't move (we assign a StaticMotionModel). We could also include a `category_tag` like `"GROUND_SITE"` for UI grouping.
  NetworkNode: ID `"node_gw_1"`, name `"Gateway-1"`, type `"GROUND_STATION_NODE"`. Attached to `"gs_1"`. This node might have a subnet field (e.g. `subnet = ["10.0.0.0/24"]` indicating it connects a certain network) just to populate that field. It might also have an `agent` (SDN agent) field indicating it's SDN-controlled (type AIRFLOW). Power budget could be high or not set for a ground station (could assume mains power, so not limiting). Storage maybe 0 (no DTN for this gateway, or maybe yes if it's DTN gateway – either way is fine to show usage of field).

Expected Behavior: "GS1" stays fixed at its ECEF coordinate (on Earth's surface). The simulation will maintain its position constant through all ticks (the static model ensures no drift). We can verify that satellites periodically come into view of this ground station by post-processing their relative geometry (not computed by the simulator in scope 1, but if we output positions we can analyze if SatLeo1 passes above horizon at GS1, for instance). The node "node_gw_1" doesn't move either since it's tied to GS1. It's essentially an endpoint on the ground.

- **User Terminal – "UT1":** *(Optional in scenario)*
  We can include a user terminal as another ground or maybe airborne asset. For example, a platform `"ut_1"` type `"USER_TERMINAL"`, which could be static at some other location (or moving car, but that would require a trajectory – perhaps keep static for now). Attach a node `"node_ut_1"` type `"USER_TERMINAL_NODE"`. This demonstrates another category. It might have limited power (say battery powered, so power_budget of 50W). We set storage maybe a few MB to indicate DTN capable (like a handheld device that can store messages). This terminal could be placed at a different lat/lon than GS1 to represent an end-user device location. If static, it behaves just like GS1 (no motion). If we wanted to test motion, we could make UT1 move along a simple path: e.g. if it's an aircraft, we could assign a scripted trajectory model. For example, UT1 at lat 10°, lon 0°, moving north at 200 m/s (if it were a plane). We can simulate that with ExternalMotionModel by updating its lat/lon each tick or having a preset function. But to keep it simple, let's assume UT1 is fixed on the ground for now, to focus on primary use-cases.

- **Infrastructure Node – "InternetGateway":** *(Optional node without platform)*
  To illustrate a node with no physical platform, define a NetworkNode ID `"node_inet_1"`, type `"INFRASTRUCTURE_NODE"` (meaning it represents a point-of-presence or core network

gateway). This node's platform_ref is unset (nil). In the scenario, this might represent the distant internet – not a physical thing in our simulation. It could have a routing_config (like router ID) and maybe connect to GS1 via a terrestrial network (not modeled in scope 1 except as a conceptual existence). This node will have no PlatformDefinition entry. The simulation will include it in the KB, but it won't have coordinates. In any outputs or future connectivity logic, it would only connect via non-spatial links. For now, it tests that we can handle nodes with `platform_ref = nil`.

Expected Behavior: The simulation doesn't attempt to update any position for this node (since none exists). It remains just an entity in the KB for completeness. There should be no errors from it being present. Queries for its position might return "none" or an error – but we likely won't query a non-physical node's position in scope 1.

With these definitions, a solo developer can set up the scenario as a combination of PlatformDefinition and NetworkNode objects (perhaps build them from JSON or directly in code). When the simulation runs: - **Time progression:** If run in real-time mode, one could start it and let it run for (say) a few minutes, observing that LEO satellites complete several orbits relative to the ground. In accelerated mode, one could simulate a full day in a short time – verifying the GEO satellite stays roughly fixed above GS1, while LEO ones periodically go in and out of range if one were to check. - **Data inspection:** The developer can call `ListPlatforms()` at any time to get current positions. For instance, after 45 minutes sim time, SatLeo1 might be on the opposite side of Earth from GS1 (depending on orbit). Although we're not computing connectivity, one could manually compute if needed by checking line-of-sight. - **Logging/Output:** The logs would show entries like: "Platform sat_leo_1 added (LEO satellite, orbit via TLE)", "Platform gs_1 added (Ground station, static)", etc. On running, logs (if debug enabled) might show tick updates: "T+60s: sat_leo_1 at (x,y,z)=..., sat_geo_1 at ... (should be nearly initial), gs_1 at ... (constant)". - **No connectivity yet:** Indeed, this scenario does not include link definitions, so the simulator just maintains these entities as isolated points with positions [77]. This is expected for Scope 1 – it lays the groundwork; in Scope 2, one would add interfaces and links between (e.g. a link from node_leo_1 to node_gw_1 representing a satellite downlink, etc., and then simulate when that link is active based on geometry).

This scenario demonstrates the **mix of LEO, GEO, and ground**: - LEO satellites moving quickly, - a GEO satellite seemingly fixed, - a fixed ground station, - and possibly a stationary user terminal and a non-physical network node.

A developer can use this setup to verify the system: e.g., print out SatLeo1's latitude/longitude over time to see it moving; ensure GS1's position remains constant; ensure no errors occur linking node_inet_1 (no platform) in the KB, etc. It provides a solid test that our core entities and orbital dynamics are functioning as intended.

With this implementation plan, a solo engineer can proceed to develop each module in isolation (following the outlined interfaces and responsibilities) and write tests for each. The design adheres to Aalyria's Spacetime data model and meets all Scope 1 requirements: defining core entities, updating their positions over time (via SGP4 for orbiting platforms) [12] [73], maintaining a knowledge base of state, and preparing the system for subsequent network-centric features. Each component can be built and tested independently, and combined to run a basic constellation simulation scenario as illustrated above. The result is a clear and extensible foundation upon which the more advanced Scopes (connectivity, scheduling, etc.) can be layered.

Sources:

- Aalyria Spacetime API Protobufs for PlatformDefinition and NetworkNode (for data model alignment) 20 36 78 79
- Project Requirements and Roadmap documents (Scope 1 goals, assumptions, and simplifications) 35
- Architecture Overview for simulation design principles (time controller, orbit propagation, knowledge base concept) 75 80 8

---

1 5 6 8 9 10 11 13 14 15 16 67 68 69 71 72 74 75 80 Architecture for a Spacetime-Compatible Constellation Simulator.pdf
file://file_000000000ac47207829eeb8ec2dd94b2

2 4 7 12 17 33 34 35 42 45 58 65 73 77 Roadmap for Spacetime-Compatible Constellation Simulator Development.pdf
file://file_00000000f1147207bb22409abe0e8599

3 Architecture for a Spacetime-Compatible Constellation Simulator.pdf
file://file_00000000b25871fa90417933660bf5d3

18 19 nbi.proto
https://github.com/aalyria/api/blob/320219aaa62ccf75690e5633ccfbef77748ddd0b/api/nbi/v1alpha/nbi.proto

20 21 24 25 26 27 28 29 30 31 32 70 platform.proto
https://github.com/aalyria/api/blob/320219aaa62ccf75690e5633ccfbef77748ddd0b/api/common/platform.proto

22 23 40 54 55 63 64 76 Requirements for an Aalyria Spacetime-Compatible Constellation Simulator.pdf
file://file_0000000084507207a89cd89149e01fa2

36 37 38 39 41 43 44 46 47 48 49 50 51 52 53 56 57 59 60 61 62 66 78 79 network_element.proto
https://github.com/aalyria/api/blob/320219aaa62ccf75690e5633ccfbef77748ddd0b/api/nbi/v1alpha/resources/network_element.proto