Sentinode X - Satellite Telemetry Simulator

Deshad Senevirathne
deshadsenevirathne@gmail.com
Independent Researcher
Colombo, Sri Lanka

Abstract

Sentinode X framework is a Systematic, multi-tiered software architecture designed for precise simulation and visualization of satellite mission dynamics. The framework is built on a microservices pattern, ensuring scalability, robust decoupling and maintainability across its seperate functional layers. The Simulation Core is scripted in Python, utilizing specialized open source libraries to manage orbital propagation, including environmental effects and hardware modeling. A critical feature is the integration of a Finite State Machine (FSM) that operates on sensor data which are bias simulated by Neural Networks. The system's integrity is managed by a Node.js API Gateway which serves as the single entry point. The gateway manages Cross-Origin Resource Sharing (CORS), unified error handling, and essential data services, including the transformation of large telemetry logs into exportable CSV formats. This architecture supports both analytical batch processing and the emulation of a real-time telemetry stream for the client dashboard. The complete environment is deployed using Docker Compose, with an Nginx reverse proxy providing a unified entry and intelligently routing requests to the corresponding backend services. This containerized strategy guarantees portability and repeatable deployment across diverse operational environments. The resulting system is a fully functional prototype test framework for validating satellite mission parameters and autonomous control system performance.

Keywords

node.js, poliastro, sentinode, simulation, telemetry

1 Introduction

Successful operations of satellite missions fundamentally relies on the rigorous testing and validation of Ground Control Software (GCS), command pipelines and mission planning systems and architectures. It requires high-precision telemetry data streams that accurately reflect orbital mechanics, sensor inputs and spacecraft state vectors. Present industry practices often rely on large, monolithic simulation tools which, while accurate, present significant challenges in modern, distributed computing environments[3]. Traditional GCS simulators frequently built using complex desktop based environments like Systems Tool Kit (STK), legacy MATLAB/Simulink models or compiled C++ physical engines are tightly coupled[2]. This comes with the difficulty of integration with ephemeral, web based dashboards, cumbersome to update and fragile to deploy across diverse cloud infrastructures (e.g., containerized Kubernates or EC2). Furthermore they often suffer from inherent instability when faced with continuous, long-duration data requests over standard network protocols (HTTP/REST). Without a protective API layer, the simulation's long processing time often triggers mandatory network timeouts at the gateway or client level, leading to an inconsistent and unreliable user expierience where the simulation prematurely halts[4]. This presents a major obstacle to using these tools effectively for front end development, operator training, and continuous integration testing.

Sentinode X is a satellite telemetry simulator specifically designed to address both architectural and stability challenges through a containerized, three-tier microservices appro The system comprises a decoupled computational backend (Python/FastAPI), a routing and stability layer (Node.js/Express), and a modern, responsive web frontend. The core technical contribution of Sentinode X lies in the design of the Node.js API Gateway as a dedicated stability mechanism. This gateway implements a mandatory, extended network timeout for all long running computational requests and utilizes a data chunking strategy which ensures that the heavy simulation workload can execute fully without triggering network disconnection or flooding client resources.

2 System Architecture

2.1 Architecture Overview

The system adheres to an API Gateway pattern, effectively decoupling the Frontend from the computational backend. This isolation allows services to be scaled and maintained independently. The three components are: Computational Backend (/python-sim-service/), API Gateway/Stability Layer (/node-api-gateway/), and Client Interface (/frontend/). The Node.js API Gateway is the sole external entry point, ensuring all stability mechanisms are enforced at the network boundary.

2.2 Python Simulation Service

The Python Simulation Service (/python-sim-service/) forms the analytical engine of the entire web application. Its primary responsibility is to generate realistic satellite telemetry data based on user defined orbital parameters. It is built on the FastAPI framework, and it provides a robust and asynchronous API endpoint that the Node.js API Gateway then utilizes to initiate and retrieve simulation results.

The workflow within this service is modular in nature, separating concerns between API exposure, simulation orchestration, and the underlying scientific models. When a request for a simulation arrives via its main API endpoint, the service first validates the input parameters. It then delegates the actual time series simulation generation to a dedicated driver script, which in turn leverages a suite of object oriented models for orbital mechanics, satellite state management, and event handling, resulting a detailed log of simulated telemetry points over the specified duration, which is then structured as a JSON response and returned to the API Gateway. This design ensures that complex computational tasks are encapsulated and efficiently handled.

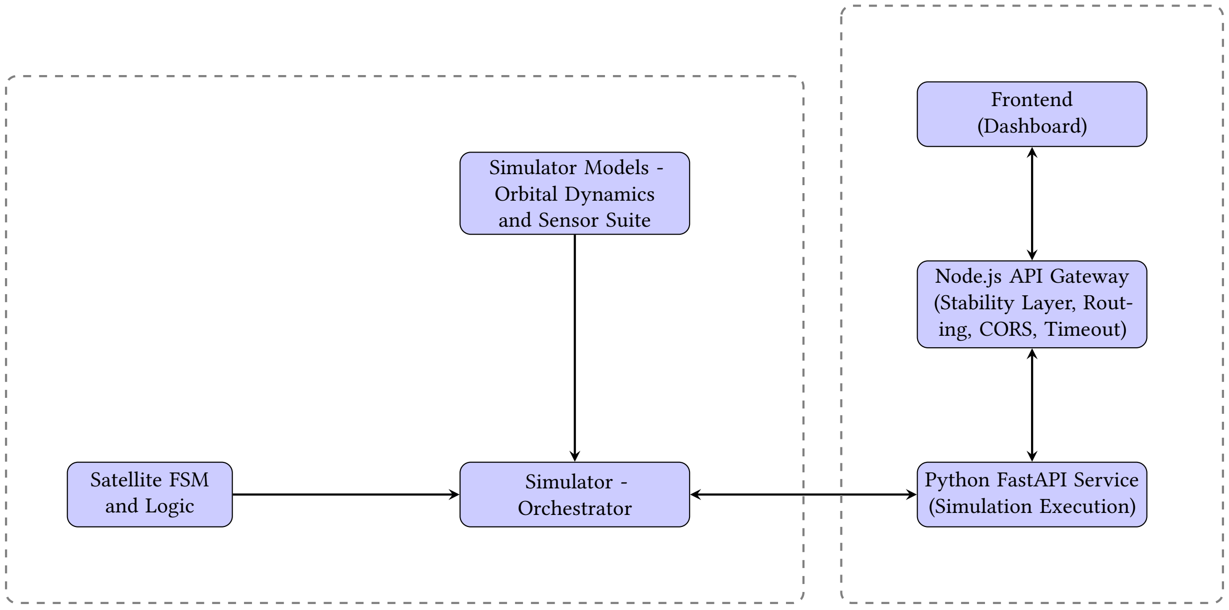

Frontend (Dashboard)
Simulator Models - Orbital Dynamics and Sensor Suite
Node.js API Gateway (Stability Layer, Routing, CORS, Timeout)
Satellite FSM and Logic
Simulator - Orchestrator
Python FastAPI Service (Simulation Execution)

Figure 1: Sentinode X Simulation Architecture: The right box (Frontend, Node.js, FastAPI) handles user requests, while the left box (Simulator, Models, Satellite FSM) handles the computational core.

2.2.1 Simulation Engine

The foundation of the high-fidelity telemetry generation lies in a robust orbital mechanics framework, encapsulated within the primary OrbitSimulator class. This component is responsible for defining, propagating, and managing the spacecraft's trajectory, leveraging open source astrodynamics libraries Poliastro[5] for orbital computations and Astropy[1] for precise time and unit management.

The initial orbital state of the simulated satellite is established upon instantiation, requiring the definition of the mission epoch, a desired altitude, and the orbital inclination. The OrbitSimulator class leverages the poliastro library to model Keplerian orbits, providing a high-fidelity, yet computationally efficient, representation of orbital mechanics for a Low Earth Orbit (LEO) satellite. The initial orbit is defined using Classical Orbital Elements (COE), specifically for a circular orbit: eccentricity $e = 0$. The initial state (position $\mathbf{r}$ and velocity $\mathbf{v}$) is computed via the `Orbit.from_classical` method, which takes the gravitational parameter of Earth μ (from `Earth.k`), the semi-major axis a, the eccentricity e, the inclination i, the Right Ascension of the Ascending Node (RAAN) Ω, the Argument of Perigee ω, and the True Anomaly ν.

The semi-major axis a is derived directly from the user-specified altitude h:

$$a = R_E + h \tag{1}$$

where R_E is the equatorial radius of the Earth (Earth.R). A crucial initial condition is the True Anomaly ν. By setting $\nu = 90^\circ$, the initial position vector $\mathbf{r}$ is placed such that its Z-component (equatorial component perpendicular to the angular momentum vector) is maximized. This corresponds to the point of maximum latitude (equal to the inclination i) in the orbit. The initial COE are:

Table 1: Initial Classical Orbital Elements (COE) for the Orbit

Element	Symbol	Value (Units)
Semi-Major Axis	a	$R_\mathrm{E} + h$ (km)
Eccentricity	e	0 (unitless)
Inclination	i	i_deg (deg)
RAAN	Ω	0 (deg)
Argument of Perigee	ω	0 (deg)
True Anomaly	ν	90 (deg)

Orbital motion is propagated using the Keplerian two-body solution via `self.orbit.propagate(tof)`. This assumes the satellite is only affected by the central gravitational body (Earth) and is described by the solution to the differential equation:

$$\ddot{\mathbf{r}} = -\frac{\mu}{r^3}\mathbf{r} \tag{2}$$

where $\mathbf{r}$ is the position vector and $r = \|\mathbf{r}\|$. The `step_simulation` function applies this propagation over discrete time steps ($\Delta t = 5$ s).

To obtain the geographic coordinates (Latitude ϕ and Longitude λ), the Earth-Centered Inertial (ECI) position vector $\mathbf{r}_\mathrm{ECI}$ must be converted to the Earth-Centered Earth-Fixed (ECEF) frame, $\mathbf{r}_\mathrm{ECEF}$. This transformation involves a rotation around the Z-axis by the Greenwich Mean Sidereal Time (GMST) angle θ_G.

The ECEF position is given by:

$$\mathbf{r}_\mathrm{ECEF} = R_z(-\theta_G)\,\mathbf{r}_\mathrm{ECI} \tag{3}$$

where $R_z(-\theta_G)$ is the negative rotation matrix about the Z-axis.

The approximate GMST in degrees is calculated using the Julian Date (JD) and the time interval from the J2000 epoch (T_UT1):

$$\theta_G = 100.46061837 + 36000.770053608\,T_\mathrm{UT1} + 0.00038793\,T_\mathrm{UT1}^2 \quad \text{(deg)}$$

with

$$T_{\mathrm{UT1}} = \frac{\mathrm{JD} - 2451545.0}{36525.0} \tag{4}$$

The geographic coordinates are then computed from the ECEF Cartesian vector $\mathbf{r}_{\mathrm{ECEF}} = [x, y, z]$:

$$\phi = \arcsin\left(\frac{z}{\|\mathbf{r}_{\mathrm{ECEF}}\|}\right), \qquad \lambda = \arctan 2(y, x) \tag{5}$$

Longitude λ is then normalized to the range $[-180°, 180°]$.

Instantaneous maneuvers are applied using the `apply_maneuver` method. A Delta-V ($\Delta\mathbf{v}$) impulse, represented by the vector $\Delta\mathbf{v}$, is added to the current orbital velocity vector $\mathbf{v}_{\mathrm{current}}$:

$$\mathbf{v}_{\mathrm{new}} = \mathbf{v}_{\mathrm{current}} + \Delta\mathbf{v} \tag{6}$$

The new orbit is then computed from the updated state vectors $(\mathbf{r}, \mathbf{v}_{\mathrm{new}})$ using `self.orbit.apply_maneuver(`$\Delta\mathbf{v}$`)`. The class is designed to automatically update the internal `self.inclination` attribute post-maneuver.

The `solve_lambert` method computes the initial and final velocity vectors $(\mathbf{v}_0, \mathbf{v}_1)$ required for a two-impulse transfer between the current position $\mathbf{r}_1$ and a target position $\mathbf{r}_2$ over a specified time-of-flight (TOF). This is achieved using the iterative algorithm of Izzo (`izzo.lambert`), which solves the boundary value problem defined by Lambert's Theorem:

$$\mathrm{TOF} = f(\mathbf{r}_1, \mathbf{r}_2, \mathbf{v}_0, \mu) \tag{7}$$

The `UncertaintyPropagator` class implements a two-pronged approach—Monte Carlo (MC) sampling and Polynomial Chaos Expansion (PCE)—to model the effect of input uncertainties on the mission's output parameters.

Input uncertainties are defined by a dictionary of named parameters, each associated with a probability distribution (currently Normal/Gaussian). For a parameter X with mean μ_X and standard deviation σ_X, the probability density function (PDF) is:

$$p(X) = \frac{1}{\sigma_X\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{X-\mu_X}{\sigma_X}\right)^2\right] \tag{8}$$

The `orbital_period_model` within the simulator defines the two perturbation parameters: `pos_perturb` and `vel_perturb`, which represent scaling factors applied to the nominal position $\mathbf{r}$ and velocity $\mathbf{v}$ vectors:

$$\mathbf{r}_{\mathrm{perturbed}} = \mathbf{r} \cdot (1 + \delta_r), \qquad \mathbf{v}_{\mathrm{perturbed}} = \mathbf{v} \cdot (1 + \delta_v)$$

where δ_r and δ_v are the sampled values of `pos_perturb` and `vel_perturb`, respectively. The output of the model is the resulting perturbed orbital period $P_{\mathrm{perturbed}}$.

The Monte Carlo (MC) simulation involves generating N_{samples} sets of input parameters by drawing random samples from the defined distributions (`sample_inputs`). The model function $f(\cdot)$ is then evaluated for each sample:

$$Y_k = f(X_k), \quad k = 1, \ldots, N_{\mathrm{samples}}$$

where $X_k = [\delta_{r,k}, \delta_{v,k}]$ is the k-th sample of input perturbations, and Y_k is the corresponding perturbed orbital period.

The MC simulation provides the statistical moments of the output:

$$\mathrm{Mean}(\hat{Y}) = \frac{1}{N_{\mathrm{samples}}} \sum_{k=1}^{N_{\mathrm{samples}}} Y_k \tag{9}$$

$$\mathrm{StdDev}(\hat{Y}) = \sqrt{\frac{1}{N_{\mathrm{samples}} - 1} \sum_{k=1}^{N_{\mathrm{samples}}} \left(Y_k - \mathrm{Mean}(\hat{Y})\right)^2} \tag{10}$$

Polynomial Chaos Expansion (PCE) is used for surrogate modeling of the uncertainty propagation. For Gaussian input variables, the expansion uses Hermite polynomials $\Psi_i(\xi)$.

The standardized input variable ξ is defined as:

$$\xi = \frac{X - \mu_X}{\sigma_X} \tag{11}$$

The model output $\hat{Y}$ is approximated as a finite-order expansion:

$$\hat{Y}(X) \approx \sum_{i=0}^{P} c_i\, \Psi_i(\xi) \tag{12}$$

where P is the total number of terms (determined by the PCE order p_{order}), c_i are the unknown PCE coefficients, and $\Psi_i(\xi)$ are the multivariate Hermite basis polynomials. Since the current implementation treats each input variable separately for fitting (due to the use of `pce_coeffs[name]`), this is effectively a single-variable PCE fit for each input:

$$\hat{Y}_X(\xi) = \sum_{i=0}^{p_{\mathrm{order}}} c_i\, \xi^i \tag{13}$$

where ξ^i are powers of the standardized variable, approximating the Hermite polynomial basis.

The coefficients c_i are determined by minimizing the least-squares error between the MC outputs Y_k and the basis matrix B evaluated at the sampled standardized inputs ξ_k:

$$\mathbf{c} = (\mathbf{B}^T\mathbf{B})^{-1}\mathbf{B}^T\mathbf{Y} \tag{14}$$

The basis matrix B is constructed as:

$$B_{k,i} = \xi_k^i, \quad k = 1, \ldots, N_{\mathrm{samples}}, \quad i = 0, \ldots, p_{\mathrm{order}}$$

This surrogate model allows for rapid prediction of the output $\hat{Y}$ for any new input perturbation ξ without re-running the full orbital dynamics model.

The Machine Learning (ML)-Enhanced Sensor Layer is crucial for generating realistic telemetry data by modeling non-ideal sensor behavior, specifically time-correlated drift and bias accumulation. This layer is implemented by integrating a specialized Recurrent Neural Network (RNN) module, `DriftGRU`, into each high-fidelity sensor class.

The core innovation of the sensor layer is the `DriftGRU` module, a minimalist implementation of a Gated Recurrent Unit (GRU) network, which serves as a parametric model for simulating long-term sensor degradation and drift.

2.2.2 GRU Architecture and Function

The GRU is chosen for its ability to model sequences and capture dependencies over time, making it ideal for simulating bias that evolves based on its previous state and internal memory.

The `DriftGRU` processes a sequence of length $L = 1$ at each simulation step Δt:

$$\mathbf{h}_t = \mathrm{GRU}(\mathbf{x}_t, \mathbf{h}_{t-1}) \tag{15}$$

where $\mathbf{x}_t$ is the input (typically the current accumulated bias/-drift), and $\mathbf{h}_t$ is the hidden state (the network's memory).

The output layer is a simple linear transformation that maps the hidden state $\mathbf{h}_t$ to the bias increment $\Delta\mathbf{d}_t$:

$$\Delta\mathbf{d}_t = W_{\text{linear}}\,\mathbf{h}_t + \mathbf{b}_{\text{linear}} \tag{16}$$

The `DriftGRU` is critically initialized with its linear layer weights and biases set near zero:

$$W_{\text{linear}} \approx 0, \quad \mathbf{b}_{\text{linear}} \approx 0 \tag{17}$$

This initialization prevents the accumulation of an explosive, non-physical drift when the model is untrained. When used in a larger ML pipeline, the network can be trained to learn realistic, time-correlated degradation patterns from historical or synthetic failure data.

The ADCS sensors model both deterministic bias and instantaneous Gaussian noise for a comprehensive representation of measurement error.

2.2.3 Star Tracker (`StarTrackerML`)

The Star Tracker models errors in the spacecraft's attitude quaternion $\mathbf{q}_{\text{true}}$. The measurement error is treated as a small rotation, $\delta\mathbf{q}$, such that

$$\mathbf{q}_{\text{measured}} = \delta\mathbf{q} \otimes \mathbf{q}_{\text{true}} \tag{18}$$

The error rotation $\delta\mathbf{q}$ is derived from an error vector $\mathbf{e}_{\text{vec}}$ (in radians), which is the sum of the accumulated bias $\mathbf{b}$ and the instantaneous Gaussian noise $\boldsymbol{\eta}_{\text{noise}}$:

$$\mathbf{e}_{\text{vec}} = \mathbf{b} + \boldsymbol{\eta}_{\text{noise}} \tag{19}$$

The `DriftGRU` takes the previous bias $\mathbf{b}_{t-1}$ as input to predict the change in bias $\Delta\mathbf{d}_t$:

$$\mathbf{b}_t = \mathbf{b}_{t-1} + \Delta\mathbf{d}_t \tag{20}$$

The Gaussian noise $\boldsymbol{\eta}_{\text{noise}}$ is modeled with $\sigma_{\text{noise}} = 0.01^\circ$ (converted to radians).

2.2.4 Sun Sensor (`SunSensorML`)

The Sun Sensor measures the true sun vector $\mathbf{s}_{\text{true}}$. The error mechanism applies a small error rotation R_{err} derived from an error vector $\mathbf{e}_{\text{vec}}$, similar to the Star Tracker:

$$\mathbf{s}_{\text{measured}} = R_{\text{err}}(\mathbf{e}_{\text{vec}})\,\mathbf{s}_{\text{true}} \tag{21}$$

The bias $\mathbf{b}$ represents a persistent misalignment/miscalibration, which drifts based on the `DriftGRU` output. The noise $\boldsymbol{\eta}_{\text{noise}}$ has $\sigma_{\text{noise}} = 0.2^\circ$.

2.2.5 Magnetometer (`MagnetometerML`)

The Magnetometer measures the local magnetic field vector $\mathbf{B}_{\text{true}}$. The errors are modeled as a simple vector sum:

$$\mathbf{B}_{\text{measured}} = \mathbf{B}_{\text{true}} + \mathbf{b} + \boldsymbol{\eta}_{\text{noise}} \tag{22}$$

Since the field is in Tesla (T), the bias and noise are also modeled in T. The standard deviation for Gaussian noise is set to $\sigma_{\text{noise}} = 50$ nT.

2.2.6 GPS Receiver (`GPSML`)

The GPS sensor measures the spacecraft's position $\mathbf{r}_{\text{true}}$ in meters. The model simulates the degradation of the positioning solution via a simple additive error vector $\mathbf{e}_{\text{vec}}$:

$$\mathbf{r}_{\text{measured}} = \mathbf{r}_{\text{true}} + \mathbf{b} + \boldsymbol{\eta}_{\text{noise}} \tag{23}$$

The bias $\mathbf{b}$ is a position offset vector that accumulates drift $\Delta\mathbf{d}_t$ from the `DriftGRU`. The Gaussian noise $\boldsymbol{\eta}_{\text{noise}}$ has $\sigma_{\text{noise}} = 0.5$ m.

2.2.7 IR Temperature Sensor (`IRSensorML`)

The IR Sensor measures a scalar temperature T_{true}. This is the only sensor with scalar input/output for the `DriftGRU` ($N_{\text{in}} = 1, N_{\text{out}} = 1$):

$$T_{\text{measured}} = T_{\text{true}} + b + \eta_{\text{noise}} \tag{24}$$

The scalar bias b accumulates drift based on b_{t-1}, and the noise η_{noise} has $\sigma_{\text{noise}} = 0.05$. This simulates calibration drift in the thermal measurement path.

The communication and power subsystems are modeled with a simplified ML drift mechanism to simulate complex, coupled degradation, providing realistic telemetry for the Finite State Machine (FSM).

2.2.8 Communication Subsystem (`CommunicationSubsystem`)

This model uses a simplified, hardcoded recurrence relation to represent an internal "hidden state" $\mathbf{h}$ (simulating complex internal drift):

$$\mathbf{h}_t = 0.9 \cdot \mathbf{h}_{t-1} + 0.1 \cdot \boldsymbol{\eta}_{\text{input}} \tag{25}$$

This hidden state is then linearly mapped to output drifts for SNR, FrequencyOffset, and PacketLoss. The final telemetry is the sum of the base value, the mapped drift, and instantaneous Gaussian noise.

2.2.9 Power Subsystem (`PowerSubsystem`)

The Power Subsystem model uses two independent `DriftGRU` modules to simulate measurement drift in key components:

- **Solar Voltage Drift:** $\texttt{DriftGRU}_S$ takes solar_voltage$_{t-1}$ as input to model the drift in the solar panel's voltage reading.
- **Battery Voltage Drift:** $\texttt{DriftGRU}_B$ takes battery_voltage$_{t-1}$ as input to model the drift in the battery's voltage sensor.

This structure allows for independent simulation of degradation in two different electrical components, reflecting the complexity of a real power system. The State of Charge (SoC) is crudely updated

Table 2: DriftGRU Module Parameters

Parameter	Symbol	Description
Input Size	N_{in}	Dimensionality of the bias vector (e.g., 3 for 3D vector sensors, 1 for scalar sensors).
Hidden Size	N_{hidden}	Internal memory dimension (default 16).
Output Size	N_{out}	Dimensionality of the predicted drift $\Delta \mathbf{d}_t$ (defaults to N_{in}).

based on the net current, allowing the FSM to trigger a LOW power state if SoC < 0.3.

2.2.10 Finite State Machine (FSM) for Spacecraft Command and Control

The Satellite Finite State Machine (FSM) is the core decision-making logic of the spacecraft simulation, responsible for autonomously managing operational modes based on the health status reported by various subsystems. Implemented in the `SatelliteFSM` class, this deterministic machine ensures mission integrity by enforcing safe operating conditions in response to anomalies.

A FSM Architecture and Configuration

The FSM is defined entirely by a JSON configuration file (`fsm_json`), which promotes modularity and allows for easy modification of mission rules without altering the core logic. The key components are:

- **States:** A set of discrete, mutually exclusive operational modes (e.g., NOMINAL, SAFE_MODE, LOW_POWER, ANOMALY).
- **Subsystems:** A registry of all critical spacecraft components (e.g., Power, Thermal, Communication), each tracking its current health status (e.g., OK, LOW, CRITICAL, FAIL).
- **Transitions:** A list of rules that dictate how the FSM moves from a source state to a target state, conditional on the current subsystem health.

B. Subsystem Status and Condition Evaluation

The `update_subsystem_status` method provides the crucial interface for the sensor layer (via the `OrbitSimulator.step_simulation` method) to feed health data into the FSM.

The FSM's ability to trigger transitions relies on the `_eval _condition` method, which parses status-check strings defined in the JSON configuration. A condition string is structured as a simple equality statement:

```
Subsystem.Attribute == 'TargetValue'
```

For example, the condition `Power.status == 'LOW'` evaluates to `True` only if the current status of the Power subsystem is `LOW`.

C. Transition Logic and Execution

The `step` method is the execution cycle of the FSM, invoked at every simulation time step. It iterates through the defined transitions and applies the following logic:

(1) **Source State Match:** Check if the transition's `from` state matches the current FSM state S_{current}, or if the transition is defined as an `ANY` state transition.
(2) **Condition Checking:** Evaluate the boolean result of the transition's condition set C.
   - For standard transitions, all conditions must be met (logical AND): $C_1 \wedge C_2 \wedge \dots$
   - For the special `ANY` state transition, the `_check_conditions` logic allows the transition if any of the conditions are met (logical OR), ensuring a rapid fail-safe response.
(3) If both the source state and conditions are satisfied, the FSM immediately transitions to the target state S_{target} and breaks the loop, ensuring only one state change occurs per step.

D. Mission Critical States

The FSM defines key states essential for mission robustness:

- **NOMINAL:** The default operating state where all systems are healthy, and the payload is actively collecting data.
- **SAFE_MODE:** A critical recovery state, typically entered when multiple systems (e.g., Power and Thermal) indicate failure or near-failure. In this state, non-essential operations are shut down to conserve resources and stabilize the spacecraft.
- **ANOMALY:** A terminal state entered via the `ANY` transition rule if any subsystem reports an ANOMALY status. Once in the ANOMALY state, the system is prevented from further automated transitions, requiring ground intervention.

This FSM structure provides a highly modular and robust framework for simulating the autonomous response capabilities of a satellite to both environmental and internal failures, directly linking low-level sensor telemetry to high-level mission operations.

2.2.11 The Simulation Driver: `run_realtime_simulation`

The `run_realtime_simulation` function serves as the central orchestration layer, integrating the Orbital Dynamics Model (`OrbitSimulator`), the Machine Learning-Enhanced Sensor Layer, and the Command and Control Logic (`SatelliteFSM`). This driver manages the simulation timeline, initializes components, executes the time-stepping loop, and logs the resulting telemetry.

A. Initialization and Configuration

The driver begins by setting up the necessary environment based on user inputs for the simulation.

Orbital Initialization:

- The base altitude is subjected to a small, random perturbation (± 0.01 km) to introduce realistic variability in the initial conditions.

- The simulation epoch is set to the system's current UTC time (`Time.now().utc`), ensuring the orbit propagation is grounded in real-world time.
- The `OrbitSimulator` is instantiated with the perturbed altitude and user-specified inclination (default 51.6°).

FSM Initialization:

- The `SatelliteFSM` is initialized using the predefined `FSM _CONFIG` JSON string, establishing the nominal operating state and the rules of transition.

Sensor Reset:

- The entire `sensor_suite` (containing instances of `GPSML`, `StarTrackerML`, etc.) is explicitly reset. This clears any accumulated internal biases (**b**) and resets the hidden states of the embedded `DriftGRU` models, ensuring each simulation run starts from a pristine state.

B. The Time-Stepping Loop

The simulation proceeds iteratively for a total duration (default 600s, or 10 minutes) in fixed time step increments (default 1 s). At each iteration, the core execution cycle is as follows:

(1) **State Propagation (`simulator.step_simulation`):** This function executes the primary simulation tasks in sequence:
  - **Orbital Mechanics:** Propagates the orbit by Δt using the Keplerian two-body solver.
  - **Sensor Measurement:** Iterates through all sensor models (`GPSML`, `StarTrackerML`, etc.), feeding them the true orbital state ($\mathbf{r}_{\text{true}}, \mathbf{v}_{\text{true}}$) and logging the corrupted, measured values, including ML drift and noise.
  - **FSM Status Update:** Based on the latest telemetry (e.g., soc from the `PowerSubsystem` and temperature from `IRSensorML`), the `OrbitSimulator` updates the FSM's internal subsystem health status (e.g., `Power.status=LOW`).
  - **FSM Step:** The FSM executes its transition logic vi `fsm.step()`, potentially moving the satellite to a new operational state (e.g., NOMINAL → LOW_POWER).
  - **Telemetry Logging:** The composite dictionary of the latest true orbital state, measured sensor values, and current FSM state is appended to the `telemetry_log`.
  - **Event Injection:** Predetermined failures can be injected at specific simulation times to test the FSM's autonomous response:
    - At $t = 60$ s, `Power.status` is set to LOW to test the NOMINAL → LOW_POWER transition.
    - At $t = 180$ s, `Power.status` is set back to OK to test the LOW_POWER → NOMINAL recovery transition.
  - **Verbose Output:** When `verbose=True`, a structured snapshot of key telemetry metrics is printed, including the FSM state, altitude, geographic position (Latitude/-Longitude), power statistics, and ADCS sensor errors.

C. Output and Data Structure

The primary output of the simulation is `telemetry_log`, a list of dictionaries that captures the complete history of the satellite's state. This structured data enables post-processing and analysis, including:

- The accuracy of the navigation solution (by comparing $\mathbf{position}_{\text{true}}$ with `gps_measured_position_m`).
- The long-term effects of ML-modeled sensor drift.
- The effectiveness of the FSM's failure response mechanisms.

2.2.12 FastAPI Entry Point and API Definition: `main.py`

The `main.py` module establishes the simulation core as a functional web service using FastAPI. This layer defines the RESTful interface, manages data contracts via Pydantic, and executes the primary simulation logic provided by `sim_driver`.

A. API Service Definition

The service is initialized as a standard FastAPI application:

```
app = FastAPI(
    title="Sentinode Python Simulation Core",
    description="Provides real time telemetry logs via the
     orbital simulation",
    version="1.0.0",
)
```

This metadata enhances the automatically generated documentation (Swagger/OpenAPI), making the service consumable by external systems.

B. Data Contracts (Pydantic Models)

Pydantic models enforce strict validation for incoming requests and outgoing responses, ensuring the API is robust and predictable.

Request Model (`RunSimulationRequest`): Defines the parameters required to start a simulation run, mapping directly to `sim_driver` inputs:

- `altitude_km (float)`: Initial mean orbital altitude in kilometers (default 600.0 km)
- `inclination_deg (float)`: Orbital inclination in degrees (default 51.6°)
- `duration_s (int)`: Total simulation duration in seconds (default 86400 s)

Response Model (`SimulationResponse`): Defines the structure of the returned data:

- `status (str)`: Indicates the result of the operation (success or error)
- `telemetry (List[Dict[str,Any]])`: Complete log of time-stamped telemetry generated by `run_realtime_simulation`
- `message (str)`: Detailed message providing context on success or failure

C. Health Check Endpoint

The `/status` endpoint is a simple GET request for service health monitoring. It reports the service status as `up` and includes a UTC timestamp, enabling external load balancers or monitoring tools to confirm availability.

D. Main Simulation Endpoint: `/run-simulation`

The core functionality is exposed via a POST request to `/run-simulation`.

Execution Flow:

(1) Upon receiving a validated request body, the endpoint extracts `altitude_km`, `inclination_deg`, and `duration_s`.
(2) It invokes the `run_realtime_simulation` function from `sim_driver`.
(3) The `verbose` flag is set to `False` to prevent voluminous console logs during API execution.
(4) **Error Handling:** The execution is wrapped in a `try...except` block. Any exceptions raised during orbital propagation, sensor stepping, or FSM logic are caught, logged, and returned to the client as an error status within the `SimulationResponse` model, maintaining a consistent API contract.

This API layer abstracts the complex simulation core behind a clean, accessible interface, enabling the satellite model to be integrated into broader mission control, visualization, or ground station software applications.

2.3 API Gateway Implementation: Node.js/Express

The Node.js/Express API Gateway is a critical infrastructural component that acts as the single entry point for all client requests. Its primary function is to decouple the client (e.g., a web front-end) from the Python simulation core, providing cross-origin resource sharing (CORS) support, unified error handling, health monitoring, and data transformation services (e.g., JSON to CSV).

A. Architectural Role and Configuration

The gateway is built on the Express.js framework and listens on `PORT 3001`. Key configuration parameters are shown in Table 3.

B. Health and Status Checks (`/api/status`)

The health check endpoint performs a cascaded check to ensure system robustness:

- **Gateway Status:** Confirms the Node.js service is running.
- **Python Service Status:** Uses `axios.get(PYTHON_BASE_URL status)` to check the Python simulation service.
- **Response:** Returns `status: 'up'` if the Python service is reachable, or `status: 'degraded'` with HTTP 503 if it is down.

C. Main Telemetry Endpoint (`/api/telemetry`)

This POST endpoint is the primary interface for initiating short-duration simulations:

- **Parameter Handling:** Accepts optional parameters: `altitude_km`, `inclination_deg`, `duration_s`, `local_timezone`. Defaults are provided (e.g., `Duration=600 s`, `Inclination=51.6°`).
- **Request Forwarding:** Forwards validated parameters to the Python simulation endpoint via `axios.post`.
- **Error Propagation:** Any errors from the Python service (e.g., HTTP 500) are caught, logged, and propagated back to the client with appropriate status codes and messages.

D. Data Transformation and CSV Export (`/api/export-csv`)

This endpoint facilitates downloading full simulation runs as structured CSV files:

- **Full Simulation Run:** Executes a POST request to the Python service with a longer `duration_s` than the real-time endpoint.
- **JSON to CSV Conversion:** The `jsonToCsv` utility performs the transformation:
  - Dynamically extracts all unique keys from the JSON telemetry objects for the CSV header.
  - Iterates through telemetry log:
    * Arrays (e.g., `position_km_true`) converted to semicolon-separated strings (`x; y; z`) formatted to five decimal places.
    * Strings/Numbers inserted directly.
    * Complex objects `JSON.stringify`ed and escaped.
  - Applies standard CSV escaping rules.
- **File Download Response:** Sets HTTP headers (`Content-Type: text/csv` and `Content-Disposition`) to instruct the client to download the CSV. The transformed `csvContent` is then sent.

The API Gateway provides a scalable, resilient, and user-friendly interface to the core Python simulation logic.

2.4 Frontend Interface and Deployment

The Frontend Interface provides a dynamic, browser-based dashboard for interacting with the simulation core. It is built using standard web technologies (HTML5, JavaScript) and styled with Tailwind CSS. The `nginx.conf` file defines the critical reverse proxy configuration necessary to route user requests from the browser to the appropriate backend services.

A. Core Frontend Logic (`index.html` and Embedded JavaScript)

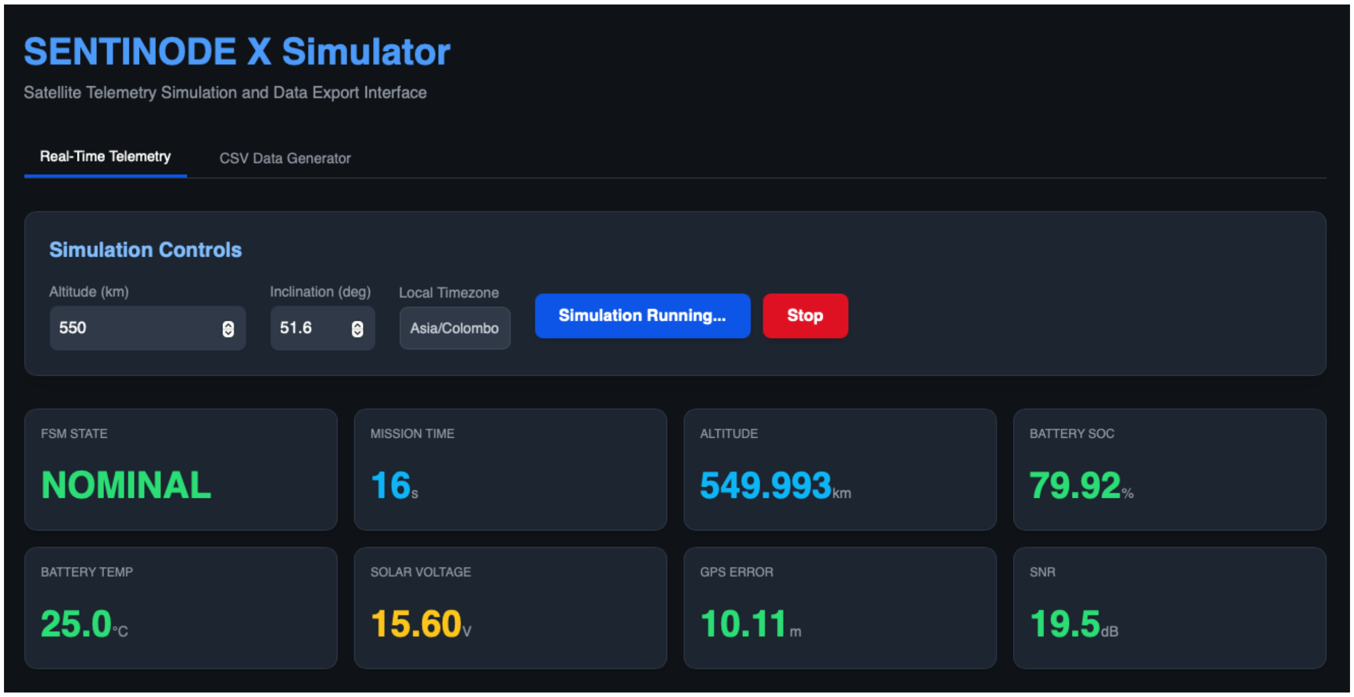
The `index.html` file contains the complete user interface, organized into two main tabs, and the comprehensive JavaScript logic for managing the simulation flow and data visualization.

1. Real-Time Telemetry Tab. This tab is designed for short, continuous simulation runs to provide an interactive "mission control" experience.

- **Continuous Fetch Logic (`fetchAndStartTelemetry`):** The frontend calls the `/api/telemetry` endpoint (via the Node.js Gateway) for short-duration blocks (default 60 s) rather than one long request.
- **Per-Second Display Loop:** After receiving a data block, a `setInterval(..., 1000)` loop iterates through the data, updating the dashboard metrics every second to mimic a real-time feed.
- **Continuous Operation:** When the current data block finishes, `fetchAndStartTelemetry()` recursively requests the next block, allowing indefinite simulation until the user clicks Stop.
- **Data Display (`updateTelemetryDisplay`):** Eight key metrics (e.g., FSM State, Altitude, Battery SOC) are displayed

Table 3: API Gateway Configuration

Configuration	Value	Purpose
`PYTHON_BASE_URL`	`http://python-sim-service:5001`	Internal network address of FastAPI simulator, crucial for Docker deployment
`cors()` Middleware	Enabled	Allows external web applications to safely make cross-origin requests
`express.json()`	Enabled	Parses incoming JSON request bodies
`timeout`	60000 ms (60 s)	Ensures the gateway waits long enough for computationally intensive simulations



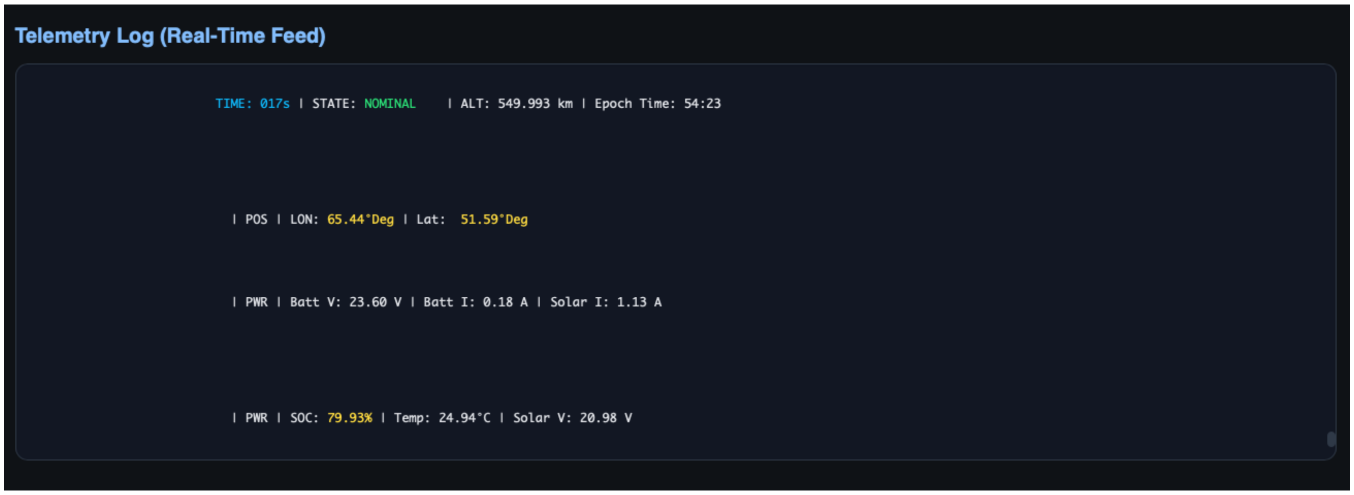
SENTINODE X Simulator

Satellite Telemetry Simulation and Data Export Interface

Altitude (km): 550
Inclination (deg): 51.6
Local Timezone: Asia/Colombo

FSM STATE: NOMINAL
MISSION TIME: 16 s
ALTITUDE: 549.993 km
BATTERY SOC: 79.92 %
BATTERY TEMP: 25.0 °C
SOLAR VOLTAGE: 15.60 V
GPS ERROR: 10.11 m
SNR: 19.5 dB

Figure 2: Sentinode X frontend display - realtime simulation



Telemetry Log (Real-Time Feed)

```
TIME: 017s | STATE: NOMINAL    | ALT: 549.993 km | Epoch Time: 54:23

 | POS | LON: 65.44°Deg | Lat:  51.59°Deg

 | PWR | Batt V: 23.60 V | Batt I: 0.18 A | Solar I: 1.13 A

 | PWR | SOC: 79.93% | Temp: 24.94°C | Solar V: 20.98 V
```

Figure 3: Sentinode X frontend display - telemetry log

in cards with color coding based on health status (Green for NOMINAL, Red for low SOC).
- **Log Area (`updateLog`):** Detailed, multi-line telemetry entries are appended to a scrollable log area, formatted to precisely match `sim_driver` console output, using non-breaking spaces for alignment.

2. CSV Data Generator Tab. Optimized for long-duration, non-interactive simulations for data analysis.
- **Single Request:** `generateCSV` makes a single POST request to `/api/export-csv` with a long duration (e.g., 86400 s).
- **CSV Download Handling:** The Node.js Gateway returns raw CSV content with the correct `Content-Disposition` header. The frontend converts this into a Blob and programmatically creates a temporary <a> tag to trigger the browser's file download dialog.

B. API Interaction (`app.js` Logic Moved to `index.html`)

- The browser's built-in `fetch` API communicates exclusively with the Node.js API Gateway (`/api/telemetry` or `/api export-csv`).
- Simulation parameters (`altitude_km`, `inclination_deg`, `duration_s`) are sent as JSON in POST requests.

C. Reverse Proxy Configuration (`nginx.conf`)

Nginx is the cornerstone of containerized deployment, unifying the frontend and backend services.

```
location / {
    root /usr/share/nginx/html;
    index index.html;
    try_files $uri $uri/ /index.html;
}
```

This block serves the static frontend assets (`index.html` and `app.js`) when users navigate to the base path (`/`).

```
location /api/ {
    proxy_pass http://node-api-gateway:3001;
    # ... standard proxy headers
}
```

All requests beginning with `/api/` (e.g., `/api/telemetry`) are routed internally to the Node.js API Gateway (`node-api-gateway`) on port 3001. This configuration abstracts internal network locations from end users.

By utilizing Nginx as the front-facing server, the entire application stack—Frontend, API Gateway, and Python Core—is accessible under a single external entry point.

3 Containerization and Deployment with Docker Compose

The complete Sentinode X application is designed as a microservices architecture, packaged and deployed using Docker and orchestrated by Docker Compose. This strategy ensures that all components—Python Simulation, Node.js Gateway, and Frontend—run in isolated, repeatable environments, guaranteeing reliable operation regardless of the host machine.

3.1 Dockerfile Definitions (Component Isolation)

Three distinct `Dockerfiles` are used, one for each service layer, ensuring each container environment is minimal and tailored to its specific task.

1. Python Simulation Core (`Dockerfile.python`).
- **Base Image:** `python:3.10-slim`, chosen for a lightweight Python environment.
- **Dependency Management:** Installs necessary system packages (`git`) before running `pip install` for scientific dependencies (NumPy, poliastro, PyTorch), including packages fetched via git URLs.
- **Execution:** Runs the FastAPI application using `uvicorn`, exposing port 5001 internally.

2. Node.js API Gateway (`Dockerfile.node`).
- **Base Image:** `node:18-alpine`, a minimal Node.js environment.
- **Dependency Management:** Copies `package.json` first to leverage Docker build cache, followed by `npm install`.
- **Execution:** Starts the Express server using `npm start`, exposing port 3001 internally.

3. Frontend (`Dockerfile.frontend`).
- **Base Image:** `nginx:alpine`, a high-performance web server.
- **Configuration:** Copies compiled frontend assets (`index.html`, `app.js, ...`) and `nginx.conf`, configuring the reverse proxy for the `/api/` route.
- **Execution:** Serves the frontend on port 80.

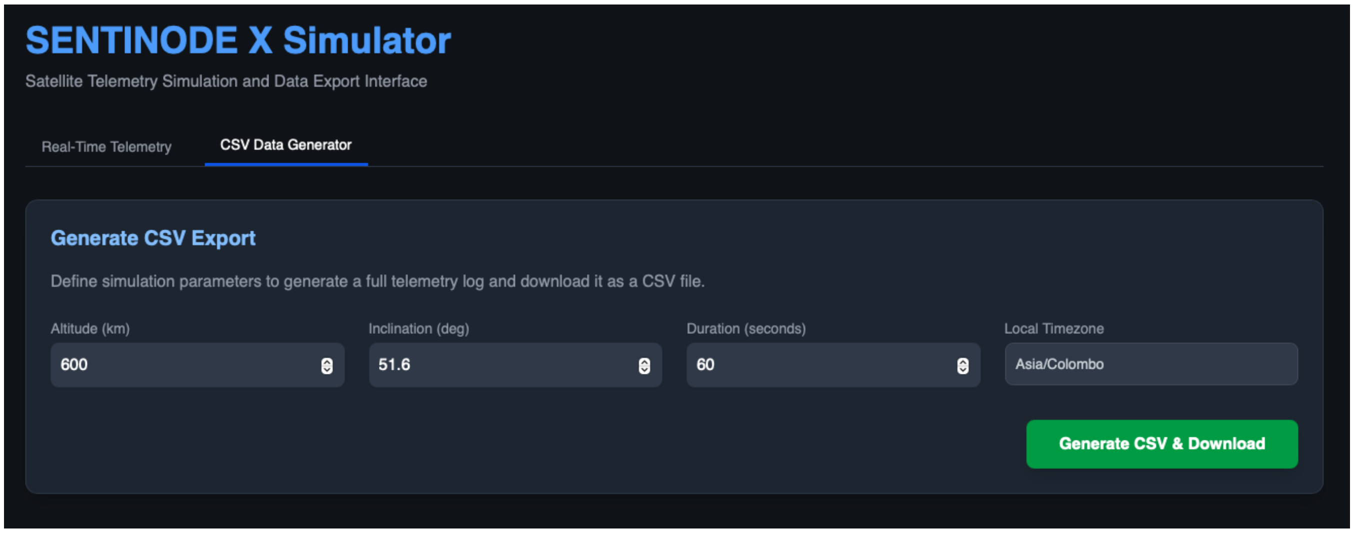
3.2 Docker Compose Configuration (`docker-compose.yml`)

The `docker-compose.yml` file defines how the three services are built, linked, and run together.

Key Features of the Compose File.
- **Shared Network (`sentinode_net`):** All services are connected to a single bridge network. This allows the Node.js gateway to reference the Python service by name (`python-sim-service:5001`), critical for inter-container communication.
- **depends_on:** Ensures the Node.js gateway waits for the Python service, and the frontend waits for the Node.js gateway to start.
- **Port Mapping:** Only the Frontend maps port 80 to the host, making the application accessible via a single URL (e.g., `http://localhost`). Other ports are exposed for monitoring/debugging.

4 Conclusion

The development of the Sentinode X framework successfully achieved its primary objective: to create a robust, end-to-end digital model that seamlessly integrates complex orbital physics, machine learning-driven sensor behaviors, autonomous command-and-control logic,


SENTINODE X Simulator

Satellite Telemetry Simulation and Data Export Interface

Real-Time Telemetry | CSV Data Generator

Generate CSV Export

Define simulation parameters to generate a full telemetry log and download it as a CSV file.

Altitude (km): 600

Inclination (deg): 51.6

Duration (seconds): 60

Local Timezone: Asia/Colombo

Generate CSV & Download

Figure 4: Sentinode X frontend display - CSV generator

Table 4: Sample telemetry data generated by Sentinode X CSV generator

Parameter	**Step 1**	**Step 2**	**Step 3**
time_s	1	2	3
timestamp_utc	2025-10-16T07:06:48.021Z	2025-10-16T07:06:49.021Z	2025-10-16T07:06:50.021Z
position_km_true	-7.55800; 4334.44900; 5468.71400	-15.11600; 4334.44200; 5468.70400	-22.67400; 4334.42900; 5468.68800
altitude_km	599.997	599.997	599.997
latitude_deg	51.6	51.6	51.6
longitude_deg	65.042	65.142	65.242
inclination_deg	51.6	51.6	51.6
gps_error_m	10.31	9.23	10.27
ir_temp_measured	34.96	35.06	34.99
SNR_dB	19.860	19.543	19.451
battery_voltage	23.600	23.600	23.600
battery_current	0.286	-0.079	0.0293
battery_temp	25.090	25.079	25.018
soc	0.800	0.800	0.800
fsm_state	NOMINAL	NOMINAL	NOMINAL

Table 5: Docker Compose Service Configuration

Service Name	**Image/Build**	**Internal Port**	**External Port**	**Dependencies**
python-sim-service	./python-sim-service	5001	5001 (Debug)	None
node-api-gateway	./node-api-gateway	3001	3001 (Debug)	python-sim-service
frontend	./frontend	80	80 (External Access)	node-api-gateway

and a scalable web interface. This system is a functional microservices architecture capable of simulating an entire mission profile from initial orbit insertion through dynamic failure and recovery cycles.

4.1 Architectural Triumphs and Implementation Effectiveness

The strength of this solution lies in its strict adherence to the principle of separation of concerns.

Scientific Integrity and Autonomy. The core `python-sim-service` encapsulates the high-fidelity scientific models. The Orbit Simulator provides the ground truth, corrupted by ML-driven sensor models to reflect real-world drift and noise—a critical step beyond purely deterministic simulations. Crucially, the Satellite Finite State Machine (`SatelliteFSM`) dictates the spacecraft's autonomous response, ensuring that mission-critical decisions (like entering `SAFE_MODE`) are based solely on simulated, noisy telemetry data, providing a true test of mission logic.

Scalable Interface and Decoupling. The architecture benefits significantly from the Node.js API Gateway. By acting as the sole intermediary, it successfully decouples the client from the compute-intensive Python core. The gateway simplifies essential operational tasks, such as translating lengthy JSON telemetry logs into exportable CSV files and managing the flow control necessary to emulate a continuous, real-time telemetry stream for the dashboard. This choice ensures the scientific processing can remain performant without being burdened by typical web server overhead.

Enterprise-Ready Deployment. The use of Docker and Docker Compose transforms the entire application from a collection of scripts into a deployable product. The `nginx.conf` file acts as the intelligent traffic cop, routing client requests to the correct internal container (`node-api-gateway`), while the `docker-compose.yml` guarantees network connectivity and dependency management across the three distinct environments (Python, Node.js, Nginx). This containerization effort makes the entire platform inherently resilient and highly portable.

5 The Path Forward: Enhancing Fidelity and State

While the current implementation is complete, future development should focus on transforming the current execution model into a true digital twin environment.

Persistence and Statefulness. The current API is transactional; it runs the simulation, returns the log, and resets. The next evolutionary step is to convert the Python core into a long-lived, stateful service. This would allow multiple external users to subscribe to the same continuously running simulation instance, enabling interactive control inputs (e.g., commanding an ADCS maneuver) that affect the orbital state in real time.

Advanced FSM Condition Parsing. A known limitation is the fragility of the `_eval_condition` method in `SatelliteFSM`. We intend to replace the current simple string parser with a more robust, expression-based evaluation engine to reliably handle complex logical operators ($\wedge$, $\vee$, $>$, $<$) and variable combinations.

Interactive Visualization. The current frontend provides raw numerical readouts. Enhancing the dashboard with dynamic charting capabilities (e.g., plotting the long-term divergence of true vs. measured position, visualizing the power State of Charge trend, and displaying the orbit path over a map) will unlock the full analytical potential of the generated telemetry.

5.1 Future Work and Digital Twin Evolution

The Sentinode X platform, having established a solid microservices foundation for satellite simulation, is now poised for an essential architectural evolution. The future work outlined below aims to transition the framework from a transactional telemetry generator into a fully interactive, stateful digital twin, thereby maximizing its utility for mission design, operator training, and autonomous control system verification.

5.2 The Stateful and Interactive Revolution

The most significant change will involve migrating the core simulation service from a short-lived API process to a long-running, persistent service that maintains state across user requests.

5.3 The Spatial and Analytical Leap (3D Visualization)

Enhancing the frontend visualization is critical for operational awareness, allowing users to move beyond abstract numbers to a spatial understanding of the mission.

1. Immersive 3D Spatial Context. The introduction of Three.js will establish a dedicated 3D Visualization Tab. This environment will dynamically render the simulated world:

- **Earth Model and Illumination:** A correctly scaled Earth, complete with visual textures and a dynamic lighting model that accurately simulates the day/night terminator based on the simulation epoch. This visually cues the spacecraft's power generation status.
- **Orbital and Attitude Fidelity:** The real-time position vectors (r_{ECI}) received from the telemetry stream will drive the location of the satellite model, while the Attitude Quaternion (q) will correctly rotate the satellite asset, showing its precise orientation in space. The full ephemeris will be plotted as a persistent orbital path trace.

2. Dual Coordinate System Overlay. To satisfy diverse mission requirements, the dashboard will display position information across two concurrent reference frames:

- **Spatial (ECI) Coordinates:** The raw, non-rotating X, Y, Z vector, which is crucial for analyzing the fundamental physics and orbital maneuvers.
- **Geographic (Lat/Lon) Coordinates:** The instantaneous latitude, longitude, and altitude of the spacecraft, which is essential for determining ground station contact windows and identifying payload targets. This display will also include the dynamic Ground Trace, showing the satellite's path projected onto the Earth's surface.

3. Analytical Data Enhancement. Beyond the 3D view, the platform's analytical utility will be expanded:

- **Time-Series Charting:** Integrate an analytical library (e.g., Chart.js) to provide dynamic plots of critical metrics over the duration of the simulation, essential for analyzing trends such as battery State of Charge, thermal cycles, and the performance (drift) of the GPSML and StarTrackerML models.
- **CSV Decimation:** The /api/export-csv endpoint will be modified to accept a decimation parameter, allowing users to request data at a lower resolution (e.g., one log per minute) for extremely long (years-long) simulation runs, thus managing data volume for post-processing.

Table 6: Planned Enhancements: Transactional vs. Stateful Simulation

Current State (Transactional)	**Proposed Enhancement (Stateful)**	**Impact on Mission Control**
Restart on Request: Each /run-simulation call initializes and resets the orbit.	Persistent State Engine: Deploy the python-sim-service to run a continuous, authoritative time loop, exposing the state via WebSockets or a Message Queue (e.g., Kafka).	Enables a true multi-client, collaborative mission environment where all users observe the same, single source of truth simultaneously.
Hardcoded Events: Failures are predetermined by simulation time in `sim_driver`.	Dynamic Command Injection: Introduce a new /api/command endpoint to receive structured user inputs (e.g., "Trigger: Heater Failure, Severity: CRITICAL").	Transforms the system into an active training simulator, allowing operators to test system resilience against unforeseen events in real time.
Simple FSM Conditions: Logic relies on basic == comparison.	Advanced Condition Parser: Refactor the `SatelliteFSM`'s `_eval_condition` to use a safe expression evaluator capable of handling complex boolean logic, such as: `Thermal.temp>80` $\wedge$ `Power.SOC<0.25`	Allows for the implementation of mission rules with high fidelity, preventing nuisance trips and improving operational efficiency.

This comprehensive set of future work items will successfully evolve Sentinode X into a high-fidelity, interactive, and indispensable Mission Visualization and Control Testbed.

Acknowledgments

I express my deepest gratitude to Dr. Sisira K. Amarasinghe for giving me the inspiration to commit to complete this project. The authors also thank Shalutha Rajapakshe, Deepak Tulsani, Thilini Deshika, and Achinthi Chamalee for their insightful discussions, technical support, and encouragement.

Finally, I wish to acknowledge my beloved parents, Mr. M. L. R. Senevirathne and Mrs. M. M. W. N. Senevirathne, my beloved sister Ms. M. K. W. Senevirathne for their unwavering support, inspiration, and belief in my abilities, which has been a constant source of motivation throughout this endeavor.

References

[1] W. Astropy Collaboration et al. 2013. Astropy: a community Python package for astronomy. *Astronomy Astrophysics* 558 (2013), A33. doi:10.1051/0004-6361/201322068

[2] S. Li, Y. Qi, and K. Song. 2025. An Overview of GEO Satellite Communication Simulation Systems. *Electronics* 14, 13 (2025), 2715. doi:10.3390/electronics14132715

[3] M. Mosleh, K. Dalili, and B. Heydari. 2018. Distributed or Monolithic? A Computational Architecture Decision Framework. *IEEE Systems Journal* 12, 1 (March 2018), 125–136. doi:10.1109/JSYST.2016.2594290

[4] Oyekunle Claudius Oyeniran, Adebunmi Okechukwu Adewusi, Adams Gbolahan Adeleke, Lucy Anthony Akwawa, and Chidimma Francisca Azubuko. 2024. Microservices Architecture in Cloud-Native Applications: Design Patterns and Scalability. *International Journal of Advanced Research and Interdisciplinary Scientific Endeavours* 1, 2 (2024), 92–106. doi:10.61359/11.2206-2409

[5] Juan Luis Cano Rodríguez, Helge Eichhorn, and Frazer McLean. 2016. Poliastro: An Astrodynamics Library Written in Python with Fortran Performance. In *6th International Conference on Astrodynamics Tools and Techniques.*