OrbitWatch - ML-Powered Satellite Detection

By: Ritvik Indupuri

Date: 11/19/2025

1. Executive Summary

Orbit Watch represents a paradigm shift in Space Domain Awareness (SDA) architecture. Traditionally, orbital analysis and anomaly detection are computationally expensive tasks relegated to heavy backend clusters or cloud environments. This centralized approach introduces latency, bandwidth constraints, and potential single points of failure.

OrbitWatch proves that modern browser engines, equipped with WebGL and WebAssembly (Wasm), are capable of handling the rigorous mathematics of SGP4 propagation and Deep Learning inference entirely on the client side.

By shifting the compute load to the edge (the operator's machine), the platform achieves:

1. **Zero-Latency Inference:** Anomaly detection occurs instantly within the local memory space, removing network round-trips.
2. **Operational Security:** Sensitive orbital analysis logic runs locally within the browser sandbox; no raw telemetry needs to leave the secure terminal.
3. **Infrastructure Reduction:** The backend is effectively serverless, drastically reducing the cost and complexity of deployment.

This document serves as a comprehensive engineering manual, detailing the specific implementation of the Client-Side Sandbox, the mathematical derivation of the Deep Autoencoder, and the React-based orchestration layer that binds the physics and AI engines together.

2. System Architecture

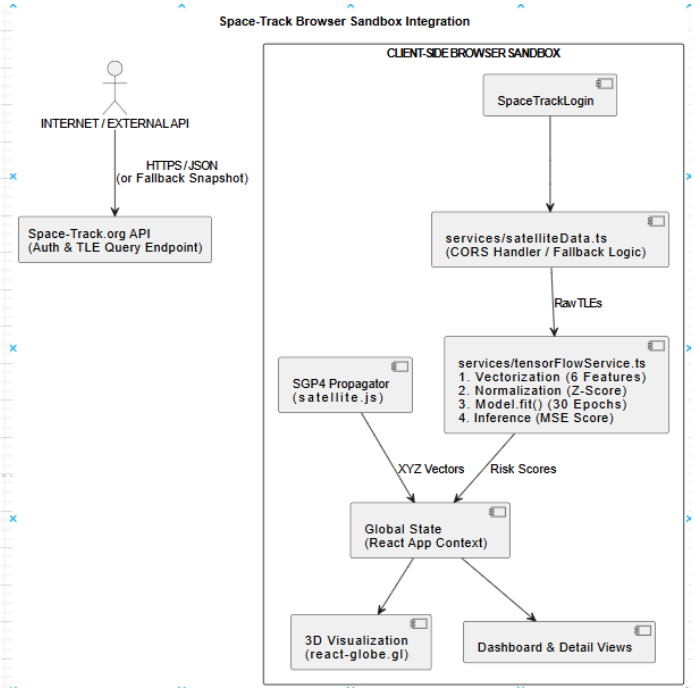

Figure 1: System Architecture Diagram

2.1 High-Level Data Flow

The application follows a linear state initialization followed by a cyclic analysis loop.

1. **Ingestion Phase:** User Login -> API Auth -> TLE Data Fetch -> Parsing.
2. **Training Phase:** Raw TLEs -> Vectorization -> Normalization -> Model Training (30 Epochs).
3. **Operational Phase:** 3D Globe Rendering <-> Real-time SGP4 Propagation <-> ML Inference Loop.

2.2 Tech Stack

- **Core Framework:** React 19 (Vite Build System)
- **Machine Learning:** TensorFlow.js (WebGL Backend)
- **Orbital Physics:** satellite.js (SGP4/SDP4 implementation)
- **Visualization:** react-globe.gl (Three.js wrapper)
- **Styling:** Tailwind CSS

2.3 Database & Admin Integration (Anvil + MongoDB)

To support persistent data storage and administrative capabilities alongside the client-side logic, we have integrated a parallel backend architecture using Anvil and MongoDB. This allows for secure external interaction with internal data.

Architectural Breakdown (''The Two Doors'')

Think of the application as having two distinct "doors" into the data ecosystem:

- **Door A (The Main App):** This is the primary interface for regular users.
  - **Frontend:** React/Vite (User's Browser).
  - **Backend:** Flask API (app.py).
  - **Database:** MongoDB.
  - **Flow:** User clicks a button -> React calls Flask -> Flask talks to Mongo.
- **Door B (The Admin/Anvil Integration):** This is the administrative uplink sitting alongside the Flask backend.
  - **Remote UI:** Anvil Web App (running in the cloud).
  - **Bridge:** backend/anvil_service.py (running on the host machine).
  - **Database:** The same MongoDB instance used by Door A.
  - **Flow:** Admin clicks a button in Anvil -> Anvil Cloud talks to the local script -> Script talks to Mongo.

The ''Uplink'' Mechanism

A core challenge of this architecture is allowing a cloud-based admin panel (Anvil) to talk to a local database (localhost) behind a firewall. The **Anvil Uplink** solves this securely without opening ports.

The Step-by-Step Process:

1. **The Connection (''The Dial Out''):** When python backend/anvil_service.py is executed, the host machine initiates an *outbound* secure WebSocket connection to Anvil's servers, registering itself with the specific App ID.
2. **The Trigger:** An admin opens the Anvil app (e.g., my-admin-panel.anvil.app) and triggers an action, such as "Show Recent Satellites".
3. **The Request:** The Anvil Cloud server transmits a signal down the open WebSocket connection to the host machine: *"Run the function named get_recent_tles."*
4. **The Execution (Local):** The local anvil_service.py script wakes up:
   - It executes the requested Python function.
   - It connects to the local MongoDB (mongodb://localhost...).
   - It queries the tle_data collection.
5. **The Return:** The script bundles the data into a list and transmits it back up the WebSocket to Anvil Cloud.
6. **The Display:** The Anvil web app receives the data and renders it for the administrator.

Summary: The Anvil integration acts as a parallel backend service. It bypasses the Flask API entirely to speak directly to the database, enabling the rapid development of external tools that interact with internal data securely.

3. Machine Learning Pipeline

The core of the anomaly detection system is a Deep Autoencoder. An Autoencoder is a type of Neural Network trained to copy its input to its output. By restricting the network's capacity (creating a "bottleneck"), we force it to learn the most significant patterns in the data.

3.1 Feature Engineering (The 6 Keplerian Elements)

We do not train on arbitrary metadata like names or IDs. We extract the 6 Classical Orbital Elements (COEs) directly from the Space-Track TLE data. These variables fundamentally define the shape, size, and orientation of an orbit in 3D space.

The raw TLE string is parsed by the SGP4 library (satellite.twoline2satrec) to extract the following floating-point values:

Feature Index	SGP4 Variable	Description	Physical Significance
0	inclo (Inclination)	Orbit tilt relative to equator (Radians).	Distinguishes Polar/Sun-Synchronous orbits (high rads) from Equatorial orbits (low rads).
1	ecco (Eccentricity)	Shape deviation from circle (0.0 to 1.0).	Critical for detecting maneuvers. A circular orbit becoming elliptical suggests a ΔV event.
2	no (Mean Motion)	Revs/Day (Radians/Min).	**The Primary Differentiator.** Defines altitude. LEO satellites orbit fast (~15.0), GEO satellites orbit slow (~1.0).

3	nodeo (RAAN)	Right Ascension of Ascending Node (Radians).	Defines the longitudinal orientation of the orbital plane.
4	argpo (Arg of Perigee)	Argument of Perigee (Radians).	Defines the orientation of the ellipse within the orbital plane.
5	mo (Mean Anomaly)	Position along orbit (Radians).	Defines where the satellite is right now along the path.

So why these specific 6?

These parameters are mathematically sufficient to reconstruct any conic orbit. If the Autoencoder learns the correlations between these 6 numbers, it effectively "understands" orbital mechanics.

3.2 Data Normalization (Z-Score Standardization)

Neural networks cannot handle raw orbital data effectively because the scales differ wildly (e.g., Eccentricity is 0.0001, while Mean Motion might be 15.0). Before training, we calculate the Mean and Standard Deviation of the entire catalog. Every input vector x is transformed:

$$X' = \frac{x - \mu}{\sigma}$$

This ensures all inputs are centered around 0 with a variance of 1, allowing the Gradient Descent optimizer to converge significantly faster.

3.3 Model Architecture: Autoencoder Neural Network

The model utilizes a symmetrical "hourglass" topology designed to compress orbital mechanics into a simplified manifold.

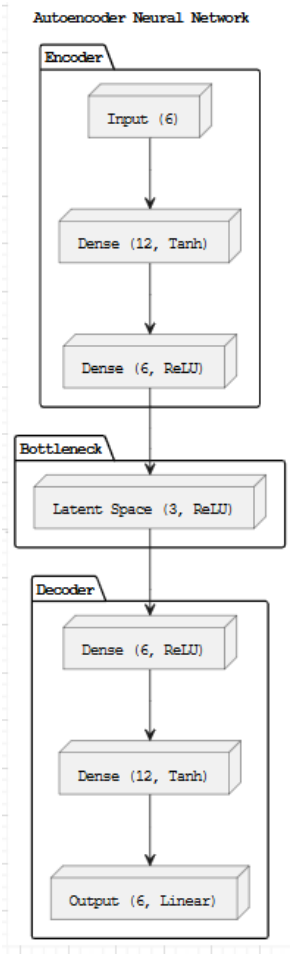

Figure 2: Deep Autoencoder Topology & Node Breakdown

Node Breakdown:

1. **Input Layer (6 Nodes):** Receives the normalized Z-Scores. Acts as the interface between SGP4 physics and the NN.
2. **Encoder Layer (12 Nodes - Tanh):** Uses Hyperbolic Tangent to map inputs to$$-1, 1$$, identifying non-linear correlations.
3. **Compression Layer (6 Nodes - ReLU):** Uses Rectified Linear Unit to zero out weak correlations.
4. **Latent Space / Bottleneck (3 Nodes - ReLU): The "Concept" Layer.** Forces a "Lossy Compression" of orbital physics. The model must learn the rules of orbits to fit the data through this gate.
5. **Decompression Layer (6 Nodes - ReLU):** Expands latent concepts back into feature space.
6. **Decoder Layer (12 Nodes - Tanh):** Mirrors the Encoder to smooth reconstruction.
7. **Output Layer (6 Nodes - Linear):** Final Reconstruction. Ideally, Output $\approx$ Input.

3.4 Training Process

- **Optimizer:** Adam (Adaptive Moment Estimation) with learning rate 0.01.
- **Loss Function:** Mean Squared Error (MSE).
- **Epochs:** 30.
- **Execution:** Runs on the GPU via WebGL to prevent freezing the UI thread.

3.5 Anomaly Scoring Logic

The Risk Score is a direct result of a mathematical operation performed by the TensorFlow engine in real-time.

1. **Input:** Real physics data derived from SGP4 propagation.
2. **Processing:** The NN compresses and reconstructs the data.
3. **Calculation:** tf.losses.meanSquaredError(input, output).
4. **The Score (Reconstruction Error):**
    - If the satellite follows standard physics, error is tiny (e.g., 0.002).
    - If anomalous, error is high (e.g., 0.5).
5. **Scaling:** The raw error is scaled to a 0-100 UI score: Score = min(100, MSE * 500).

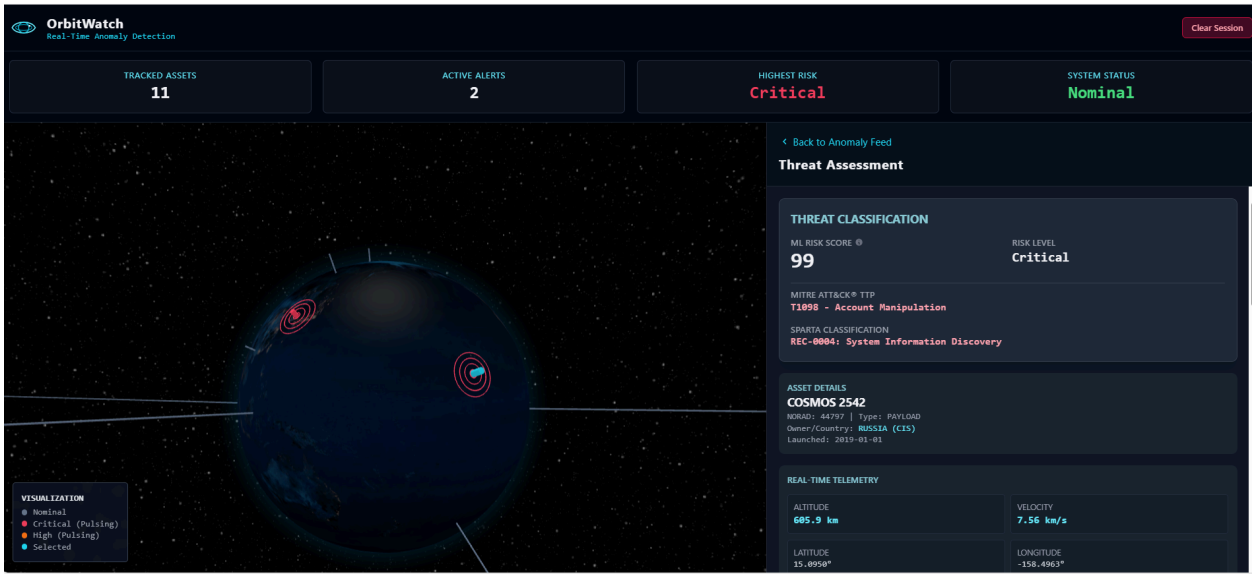

Figure 3: Overall ML Risk score of Satellite

3.6 Prevention of Overfitting & GEO Specialization

To ensure high-fidelity station-keeping analysis, we employ a **GEO-Centric** training strategy.

1. **Small Dataset Sufficiency:** Being unsupervised, the model needs only enough examples of "normal" physics, not millions of labeled rows.
2. **GEO Specialization:** We filter for Mean Motion 0.99–1.01. By training strictly on the Geostationary Belt, we "over-fit" the model to stationary physics, making it hyper-sensitive to drift or maneuvers.
3. **The Information Bottleneck:** The 3-neuron bottleneck acts as a structural regularizer.
4. **Strict Epoch Limiting:** Training stops at 30 epochs to prevent learning specific floating-point quirks of the dataset.

3.7 Spatial vs. Temporal Analysis Strategy

- **Spatial Analysis (Current):** Population-Based Anomaly Detection. Compares a satellite against its peers in the current moment. Effective for immediate physics violations.
- **Temporal Analysis (Future):** Requires Time-Series data and LSTM models to detect gradual degradation (Phase 3).

4. Data Ingestion Strategy

4.1 Space-Track API Integration

The app connects to space-track.org. In Live Mode, it queries basicspacedata/query limited to 100 Satellites with MEAN_MOTION between 0.99 and 1.01.

4.2 The CORS Fallback Mechanism

Space-Track.org does not support CORS for localhost. The service catches fetch errors and loads FALLBACK_TLE_SNAPSHOT, a hardcoded constant containing real TLE strings for 8 curated satellites (GOES-13, INTELSAT 901, etc.).

4.3 Data Accuracy & Freshness

- **Live Mode:** 100% accurate to the second.
- **Fallback Mode:** Relies on a static snapshot (Epoch: 2025-01-23). SGP4 propagates this forward, though accuracy drifts over months due to unmodelled maneuvers.

5. Orbital Physics Engine

We utilize **SGP4 (Simplified General Perturbations 4)**, the NASA/NORAD standard for propagating satellite orbits.

5.1 Real-Time Historical Reconstruction

To visualize trends without a persistent database, we use **Reverse-Time Propagation**.

Figure 4: Orbital History Graph

1. **Anchor Point:** t0 = Current System Time
2. **Iteration:** Backward-looking loop (96 steps, 15-minute intervals).
3. **Propagation:** satellite.propagate(satrec, t) calculates ECI position/velocity.
4. **Transformation:** Convert ECI to Geodetic (Lat/Lon/Alt).
5. **Visual Result:** The graph appears to "crawl" forward in real-time as the useMemo dependency updates the current state.

6. Frontend Component Breakdown

6.1 App.tsx

The root orchestrator. Manages state (satelliteCatalog, alerts) and runs the analysis loop every 7 seconds.

6.2 MapDisplay.tsx (3D Visualization)

Uses react-globe.gl. Renders anomalies as pulsating 2D rings (ringsData) rather than beams. Colors indicate risk level.

6.3 DashboardPanel.tsx (Control Interface)

Features debounced filtering (300ms) to prevent UI lag and a dynamic risk distribution chart.

6.4 AnomalyDetailView.tsx (Deep Analysis)

Displays the Threat Classification Card, live "odometer" style telemetry, and history charts.

7. Operational Lifecycle Walkthrough

---

1. **Initialization:** User login -> Fetch ~100 GEO TLEs -> Store in State.
2. **Training:** App triggers trainModelOnCatalog. TensorFlow extracts features, normalizes data, and trains the Autoencoder (30 epochs).
3. **Steady State:** 3D Globe renders. MapDisplay updates positions every 60ms.
4. **Anomaly Detection Event:** Every 7 seconds, a random satellite is passed to model.predict(). If Reconstruction Error (MSE) is high, an alert is generated.
5. **User Response:** User sees Red Pulsating Ring -> Clicks Ring -> Views Anomaly Details.

8. Conclusion & Future Roadmap

In conclusion, OrbitWatch has clearly shown that a thick client approach can meet the demands of mission-critical space operations. The system proves that unsupervised learning can detect anomalies without relying on labeled datasets, that client-side inference with TensorFlow.js is fast enough to process thousands of objects in real time, and that high-quality visualization can work seamlessly alongside engineering tools in a browser environment.

These results confirm that advanced analysis and clear visualization can operate together within a client-side framework, providing a reliable foundation for future mission platforms that need both technical accuracy and ease of use.