Technical Design Document: Orbital-Sim

Project: Orbital-Sim: Fault-Tolerant Distributed AI Uplink

Version: 1.0

Author: L. Elaine Dazzio

Status: Draft

Date: January 2026

1. Executive Summary

Orbital-Sim is a distributed microservices architecture designed to simulate the operational constraints of a space-based AI training cluster. Unlike terrestrial RAG (Retrieval-Augmented Generation) systems, this project explicitly models the **physics** of the Starcloud architecture: high-latency uplinks, thermal radiative cooling limits, and batch-processing power budgets.

The system demonstrates **reliability engineering** by implementing a fault-tolerant "Data Shuttle" transmission protocol and a "Thermal Manager" that throttles compute based on simulated radiative heat rejection in a vacuum.

2. System Architecture

The system consists of three Dockerized microservices orchestrated via `docker-compose`.

```
graph TD
    subgraph "GROUND STATION (Low Latency / High Availability)"
        User[User / Client] -->|Upload Job| API[Ground API (FastAPI)]
        API -->|Package Payload| Shuttle[Data Shuttle Service]
    end

    subgraph "THE UPLINK (Chaos Layer)"
        Shuttle -.->|Variable Latency / Packet Loss| Link[Network Simulator]
        Link -.->|Receive Payload| Node[Orbital Node]
    end

    subgraph "ORBITAL CLUSTER (Thermal Constraints)"
        Node -->|Check Thermal Headroom| Therm[Thermal Manager]
        Therm -->|Permit Compute| SK[Semantic Kernel Worker]
        SK -->|Vectorize| VDB[(Qdrant Vector DB)]

        Therm -- "Too Hot (>80C)" --> Reject[Reject / Throttle]
    end

    style User fill:#f9f,stroke:#333,stroke-width:2px
```

```
style Link fill:#ff9,stroke:#333,stroke-width:2px,stroke-dasharray: 5 5
style Therm fill:#f99,stroke:#333,stroke-width:2px
```

2.1 Service A: Ground Station (The "Client")

- **Role:** Simulates the terrestrial interface.
- **Function:** Accepts large "Training Jobs" (PDFs/Datasets) and queues them for "Upload."
- **Key Feature (Data Shuttle Simulation):** Implements a delayed transmission protocol. Instead of instant upload, data is packaged into "Payloads" with transmission delays, mimicking the physical transport of data storage units (Data Shuttles) to orbit.
- **Tech Stack:** Python (FastAPI), Azure Storage Emulator (Blob).

2.2 Service B: The Uplink (Network Simulator)

- **Role:** The Chaos Middleware.
- **Function:** Intercepts traffic between Ground and Orbit to validate fault tolerance.
- **Physics Implementation:**
  - **Latency Injection:** Adds variable latency (500ms - 2s) to simulate RF/Optical link alignment.
  - **Bit Flips:** Randomly corrupts packet data to test application-layer error correction.
  - **Connection Drops:** Simulates "Line of Sight" loss during orbital passes.

2.3 Service C: Orbital Node (The "Satellite")

- **Role:** The Compute Worker.
- **Function:** Runs the Semantic Kernel pipeline to process data and generate embeddings.
- **Physics Implementation (Derived from Starcloud White Paper):**
  - **Thermal Throttling:** Cooling is limited by radiator surface area and deep space temperature (-270°C).
  - **Logic:** The service tracks a virtual `current_temp`. High CPU usage increases `current_temp`. If `current_temp` exceeds `max_temp` (simulating radiator saturation), the service **rejects** new jobs until it "cools down."

3. Detailed Component Specifications

3.1 The Thermal Manager Class (Python)

Logic derived from Stefan-Boltzmann law constraints cited in Starcloud technical documentation.

```
class RadiatorSystem:
    def __init__(self, surface_area_m2=1.0):
        self.temp_k = 293.15  # 20C starting temp
        self.surface_area = surface_area_m2

    def calculate_rejection(self):
        """
        Calculates heat rejection capability in vacuum.
        P = epsilon * sigma * T^4
        """
        epsilon = 0.92 # Emissivity of radiator coating
        sigma = 5.67e-8
        # Radiates to deep space (effectively 3K, negligible for influx)
        power_radiated = epsilon * sigma * (self.temp_k ** 4) * self.surface_arec
        return power_radiated

    def tick(self, cpu_load_watts):
        # Physics loop: Heat In vs. Heat Out
        heat_out = self.calculate_rejection()
        net_heat = cpu_load_watts - heat_out

        # Update temp based on thermal mass (simplified for 50kg node)
        thermal_mass = 50 * 900 # mass * specific heat of aluminum
        temp_change = net_heat / thermal_mass
        self.temp_k += temp_change

        if self.temp_k > 353.15: # 80C Overheating safety trip
             raise ThermalThrottlingException("Radiator Saturation Reached - Cool
```

3.2 The Cluster Network Topology

- **Requirement:** Low-latency training cluster.
- **Implementation:** Docker Compose defines an internal `orbital_mesh` network isolated from the `ground_link` .
- **Constraint:** The Ground Station **cannot** communicate directly with the Vector Database. All queries must be routed through the `Orbital Node` , enforcing the constraint of limited downlink bandwidth.

4. Testing & Validation Strategy

4.1 Unit Testing

- **Target:** `RadiatorSystem` logic.

- **Test:** Validate that heat rejection scales correctly with temperature (T^4) and that the system triggers exceptions at the 80°C limit.

4.2 Chaos Testing (Integration)

- **Tool:** `tox` automation / Custom Chaos Scripts.
- **Scenario: "Eclipse Event"**
  - **Action:** Manually set `available_power` flag to 0 via specific API call.
  - **Expected Behavior:** System must gracefully pause ingestion, queue incoming jobs, and auto-resume when power is restored, without data corruption.
- **Scenario: "Loss of Signal (LOS)"**
  - **Action:** `Network Simulator` drops 100% of packets for 10 seconds.
  - **Expected Behavior:** Ground Station retry logic must buffer requests and successfully retransmit once the link is re-established.

5. Development Roadmap

Phase 1: Infrastructure

- [ ] Initialize `.devcontainer` with Python 3.12 and Docker-in-Docker support.
- [ ] Create `docker-compose.yml` defining the three service containers.
- [ ] Establish basic HTTP communication between Ground and Link services.

Phase 2: The Physics Engine

- [ ] Implement `RadiatorSystem` class.
- [ ] Implement `NetworkSimulator` with configurable latency/drop rates.
- [ ] Write Unit Tests validating the physics math against white paper specifications.

Phase 3: The AI Worker

- [ ] Integrate **Semantic Kernel** into the Orbital Node.
- [ ] Connect Node to **Qdrant** (Vector DB) for embedding storage.