🚀 SPACE DEBRIS TRACKING SYSTEM - COMPLETE UPGRADE

File-by-File Contents

=== requirements.txt ===

```
flask==2.3.3
sgp4==2.21
numpy==1.24.3
scipy==1.11.2
torch==2.0.1
pandas==2.0.3
requests==2.31.0
reportlab==4.0.4
celery==5.3.1
redis==4.6.0
pytest==7.4.0
pytest-cov==4.1.0
python-dateutil==2.8.2
skyfield==1.46
matplotlib==3.7.2
plotly==5.15.0
gunicorn==21.2.0
```

=== backend/app.py ===

```
from flask import Flask, jsonify, request, send_file
from datetime import datetime, timedelta
import os
import json
from .ingest import TLEIngester
from .propagate import SGP4Propagator
from .screen import ConjunctionScreener
from .pc import ProbabilityCalculator
from .realtime import RealtimeProcessor
from .degrade_policy import DegradationPolicy
import logging

app = Flask(__name__)
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Initialize components
tle_ingester = TLEIngester()
```

```
propagator = SGP4Propagator()
screener = ConjunctionScreener()
pc_calculator = ProbabilityCalculator()
realtime_processor = RealtimeProcessor()
degradation_policy = DegradationPolicy()

@app.route('/api/health')
def health_check():
    """System health endpoint"""
    return jsonify({
        'status': 'healthy',
        'timestamp': datetime.utcnow().isoformat(),
        'version': '1.0.0',
        'components': {
            'tle_ingester': tle_ingester.status(),
            'propagator': propagator.status(),
            'screener': screener.status()
        }
    })

@app.route('/api/tles/latest')
def get_latest_tles():
    """Get latest TLE data"""
    try:
        tles = tle_ingester.get_latest()
        return jsonify({
            'count': len(tles),
            'timestamp': datetime.utcnow().isoformat(),
            'tles': tles[:100]  # Limit response size
        })
    except Exception as e:
        logger.error(f"Error fetching TLEs: {e}")
        return jsonify({'error': str(e)}), 500

@app.route('/api/propagate', methods=['POST'])
def propagate_orbit():
    """Propagate orbital elements to specified time"""
    try:
        data = request.json
        tle_line1 = data.get('tle_line1')
        tle_line2 = data.get('tle_line2')
        target_time = datetime.fromisoformat(data.get('target_time'))

        position, velocity, uncertainty = propagator.propagate(
            tle_line1, tle_line2, target_time
        )

        return jsonify({
            'position': position.tolist(),
            'velocity': velocity.tolist(),
            'uncertainty': uncertainty.tolist(),
            'timestamp': target_time.isoformat()
        })
    except Exception as e:
        logger.error(f"Propagation error: {e}")
        return jsonify({'error': str(e)}), 500
```

```
@app.route('/api/conjunction/screen', methods=['POST'])
def screen_conjunctions():
    """Screen for potential conjunctions"""
    try:
        data = request.json
        objects = data.get('objects', [])
        time_window = data.get('time_window_hours', 24)

        candidates = screener.screen(objects, time_window)

        return jsonify({
            'candidates': len(candidates),
            'results': candidates,
            'timestamp': datetime.utcnow().isoformat()
        })
    except Exception as e:
        logger.error(f"Screening error: {e}")
        return jsonify({'error': str(e)}), 500

@app.route('/api/pc/calculate', methods=['POST'])
def calculate_pc():
    """Calculate probability of collision"""
    try:
        data = request.json
        obj1_state = data.get('object1')
        obj2_state = data.get('object2')

        pc, tca, miss_distance = pc_calculator.calculate(obj1_state, obj2_state)

        return jsonify({
            'probability': float(pc),
            'tca': tca.isoformat(),
            'miss_distance': float(miss_distance),
            'timestamp': datetime.utcnow().isoformat(),
            'calibrated': True
        })
    except Exception as e:
        logger.error(f"Pc calculation error: {e}")
        return jsonify({'error': str(e)}), 500

@app.route('/api/realtime/status')
def realtime_status():
    """Get real-time processing status"""
    return jsonify(realtime_processor.get_status())

@app.route('/api/reports/generate', methods=['POST'])
def generate_report():
    """Generate PDF report"""
    try:
        from reports.generate import ReportGenerator
        generator = ReportGenerator()

        report_type = request.json.get('type', 'daily')
        report_path = generator.generate(report_type)
```

```
        return send_file(report_path, as_attachment=True)
    except Exception as e:
        logger.error(f"Report generation error: {e}")
        return jsonify({'error': str(e)}), 500

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5000)
```

=== backend/ingest.py ===

```
import requests
import os
from datetime import datetime, timedelta
import json
import hashlib
from pathlib import Path

class TLEIngester:
    """Ingests TLE data from CelesTrak and manages versioning"""

    def __init__(self, data_dir="./data/tles"):
        self.data_dir = Path(data_dir)
        self.data_dir.mkdir(parents=True, exist_ok=True)
        self.celestrak_urls = [
            "https://celestrak.org/NORAD/elements/gp.php?GROUP=active&FORMAT=tle",
            "https://celestrak.org/NORAD/elements/gp.php?GROUP=debris&FORMAT=tle"
        ]

    def fetch_latest(self):
        """Fetch latest TLE data from CelesTrak"""
        all_tles = []

        for url in self.celestrak_urls:
            try:
                response = requests.get(url, timeout=30)
                response.raise_for_status()

                lines = response.text.strip().split('\n')

                # Parse TLE format (3 lines per object)
                for i in range(0, len(lines), 3):
                    if i + 2 < len(lines):
                        name = lines[i].strip()
                        line1 = lines[i + 1].strip()
                        line2 = lines[i + 2].strip()

                        if line1.startswith('1 ') and line2.startswith('2 '):
                            all_tles.append({
                                'name': name,
                                'line1': line1,
                                'line2': line2,
                                'source': url,
                                'fetched_at': datetime.utcnow().isoformat()
                            })
```

```
            except Exception as e:
                print(f"Error fetching from {url}: {e}")

        return all_tles

    def save_tles(self, tles):
        """Save TLEs with versioning"""
        timestamp = datetime.utcnow().strftime("%Y%m%d_%H%M%S")

        # Create versioned file
        version_file = self.data_dir / f"tles_{timestamp}.json"

        # Save with metadata
        data = {
            'metadata': {
                'count': len(tles),
                'fetched_at': datetime.utcnow().isoformat(),
                'version': timestamp,
                'checksum': self._calculate_checksum(tles)
            },
            'tles': tles
        }

        with open(version_file, 'w') as f:
            json.dump(data, f, indent=2)

        # Update latest symlink
        latest_file = self.data_dir / "latest.json"
        if latest_file.is_symlink():
            latest_file.unlink()
        latest_file.symlink_to(version_file.name)

        return version_file

    def get_latest(self):
        """Get latest TLE data"""
        latest_file = self.data_dir / "latest.json"

        if not latest_file.exists():
            # Fetch if no data exists
            tles = self.fetch_latest()
            self.save_tles(tles)
            return tles

        with open(latest_file, 'r') as f:
            data = json.load(f)

        # Check if data is stale (>6 hours)
        fetched_at = datetime.fromisoformat(data['metadata']['fetched_at'])
        if datetime.utcnow() - fetched_at > timedelta(hours=6):
            # Refetch automatically
            tles = self.fetch_latest()
            self.save_tles(tles)
            return tles

        return data['tles']
```

```
    def _calculate_checksum(self, tles):
        """Calculate checksum for TLE data"""
        content = json.dumps(tles, sort_keys=True)
        return hashlib.sha256(content.encode()).hexdigest()

    def status(self):
        """Get ingester status"""
        latest_file = self.data_dir / "latest.json"

        if not latest_file.exists():
            return {'status': 'no_data', 'last_update': None}

        with open(latest_file, 'r') as f:
            data = json.load(f)

        return {
            'status': 'operational',
            'last_update': data['metadata']['fetched_at'],
            'count': data['metadata']['count'],
            'version': data['metadata']['version']
        }
```

=== backend/propagate.py ===

```
import numpy as np
from sgp4.api import Satrec, jday
from datetime import datetime, timezone
import logging

logger = logging.getLogger(__name__)

class SGP4Propagator:
    """SGP4-based orbit propagator with uncertainty estimation"""

    def __init__(self):
        self.propagation_count = 0
        self.error_count = 0

    def propagate(self, tle_line1, tle_line2, target_time):
        """
        Propagate orbit to target time using SGP4
        Returns position (km), velocity (km/s), and uncertainty estimate
        """
        try:
            # Create satellite object from TLE
            satellite = Satrec.twoline2rv(tle_line1, tle_line2)

            # Convert target time to Julian day
            if isinstance(target_time, datetime):
                if target_time.tzinfo is None:
                    target_time = target_time.replace(tzinfo=timezone.utc)
                jd, fr = jday(target_time.year, target_time.month, target_time.day,
                              target_time.hour, target_time.minute, target_time.second)
            else:
```

```
                raise ValueError("target_time must be datetime object")

            # Propagate
            error_code, position_teme, velocity_teme = satellite.sgp4(jd, fr)

            if error_code != 0:
                self.error_count += 1
                raise RuntimeError(f"SGP4 error code: {error_code}")

            # Convert to numpy arrays
            position = np.array(position_teme)  # km
            velocity = np.array(velocity_teme)  # km/s

            # Estimate uncertainty based on age of TLE
            tle_epoch_jd = satellite.jdsatepoch + satellite.jdsatepochF
            current_jd = jd + fr
            age_days = current_jd - tle_epoch_jd

            # Uncertainty grows with TLE age (simplified model)
            position_uncertainty = self._estimate_position_uncertainty(age_days, satellit

            self.propagation_count += 1

            return position, velocity, position_uncertainty

        except Exception as e:
            self.error_count += 1
            logger.error(f"Propagation failed: {e}")
            raise

    def _estimate_position_uncertainty(self, age_days, satellite):
        """Estimate position uncertainty based on TLE age and orbital parameters"""

        # Base uncertainty (km) - grows with age
        base_sigma = 0.1 + 0.05 * age_days  # 100m + 50m per day

        # Scale by orbital regime
        mean_motion = satellite.no_kozai  # revolutions per day
        altitude_factor = 1.0

        if mean_motion > 14:  # LEO
            altitude_factor = 2.0  # Higher atmospheric drag uncertainty
        elif mean_motion < 1:   # GEO
            altitude_factor = 0.5  # More stable orbits

        sigma = base_sigma * altitude_factor

        # Return 3x3 covariance matrix (simplified diagonal)
        return np.diag([sigma**2, sigma**2, sigma**2])

    def batch_propagate(self, tle_pairs, target_time):
        """Propagate multiple objects to same time"""
        results = []

        for tle_line1, tle_line2 in tle_pairs:
            try:
```

```
                pos, vel, unc = self.propagate(tle_line1, tle_line2, target_time)
                results.append({
                    'position': pos,
                    'velocity': vel,
                    'uncertainty': unc,
                    'success': True
                })
            except Exception as e:
                results.append({
                    'error': str(e),
                    'success': False
                })

        return results

    def status(self):
        """Get propagator status"""
        return {
            'status': 'operational',
            'propagations': self.propagation_count,
            'errors': self.error_count,
            'success_rate': (self.propagation_count / max(1, self.propagation_count + sel
        }
```

=== backend/screen.py ===

```
import numpy as np
from datetime import datetime, timedelta
from scipy.spatial import cKDTree
import logging

logger = logging.getLogger(__name__)

class ConjunctionScreener:
    """Fast conjunction screening using spatial hashing and sweep-line algorithm"""

    def __init__(self, screening_distance=5.0):  # km
        self.screening_distance = screening_distance
        self.screening_count = 0

    def screen(self, objects, time_window_hours=24):
        """
        Screen for potential conjunctions within time window

        Args:
            objects: List of objects with TLE data
            time_window_hours: Screening time window

        Returns:
            List of conjunction candidates
        """
        candidates = []
        current_time = datetime.utcnow()

        # Sample times within window
```

```
        time_steps = np.arange(0, time_window_hours, 0.25)  # 15-minute steps

        for hours_ahead in time_steps:
            target_time = current_time + timedelta(hours=hours_ahead)

            # Propagate all objects to this time
            positions = []
            valid_objects = []

            from .propagate import SGP4Propagator
            propagator = SGP4Propagator()

            for obj in objects:
                try:
                    pos, vel, unc = propagator.propagate(
                        obj['line1'], obj['line2'], target_time
                    )
                    positions.append(pos)
                    valid_objects.append(obj)
                except Exception as e:
                    logger.warning(f"Propagation failed for {obj.get('name', 'unknown')}:
                    continue

            if len(positions) < 2:
                continue

            # Use spatial tree for fast nearest neighbor search
            positions_array = np.array(positions)
            tree = cKDTree(positions_array)

            # Find close approaches
            pairs = tree.query_pairs(r=self.screening_distance)

            for i, j in pairs:
                obj1 = valid_objects[i]
                obj2 = valid_objects[j]

                distance = np.linalg.norm(positions_array[i] - positions_array[j])

                candidates.append({
                    'object1': {
                        'name': obj1.get('name', 'Unknown'),
                        'norad_id': self._extract_norad_id(obj1['line1']),
                        'position': positions_array[i].tolist()
                    },
                    'object2': {
                        'name': obj2.get('name', 'Unknown'),
                        'norad_id': self._extract_norad_id(obj2['line1']),
                        'position': positions_array[j].tolist()
                    },
                    'time': target_time.isoformat(),
                    'distance': float(distance),
                    'screening_distance': self.screening_distance
                })

        # Remove duplicates and sort by distance
```

```
        candidates = self._deduplicate_candidates(candidates)
        candidates.sort(key=lambda x: x['distance'])

        self.screening_count += 1
        logger.info(f"Screened {len(objects)} objects, found {len(candidates)} candidates

        return candidates[:50]  # Limit to top 50 closest approaches

    def _extract_norad_id(self, tle_line1):
        """Extract NORAD ID from TLE line 1"""
        try:
            return int(tle_line1[2:7])
        except ValueError:
            return None

    def _deduplicate_candidates(self, candidates):
        """Remove duplicate object pairs"""
        seen_pairs = set()
        unique_candidates = []

        for candidate in candidates:
            id1 = candidate['object1']['norad_id']
            id2 = candidate['object2']['norad_id']

            if id1 is None or id2 is None:
                continue

            # Create sorted pair key to avoid duplicates
            pair_key = tuple(sorted([id1, id2]))

            if pair_key not in seen_pairs:
                seen_pairs.add(pair_key)
                unique_candidates.append(candidate)

        return unique_candidates

    def refine_conjunction(self, obj1_tle, obj2_tle, rough_tca):
        """Refine conjunction analysis in Hill frame"""
        from .refine import HillFrameRefiner
        refiner = HillFrameRefiner()

        return refiner.refine(obj1_tle, obj2_tle, rough_tca)

    def status(self):
        """Get screener status"""
        return {
            'status': 'operational',
            'screenings_performed': self.screening_count,
            'screening_distance_km': self.screening_distance
        }
```

=== backend/refine.py ===

```
import numpy as np
from datetime import datetime, timedelta
```

```
from scipy.optimize import minimize_scalar
import logging

logger = logging.getLogger(__name__)

class HillFrameRefiner:
    """Hill frame conjunction refinement for precise TCA and miss distance"""

    def __init__(self):
        pass

    def refine(self, obj1_tle, obj2_tle, rough_tca, search_window_minutes=60):
        """
        Refine conjunction in Hill frame coordinate system

        Args:
            obj1_tle: Primary object TLE (line1, line2)
            obj2_tle: Secondary object TLE (line1, line2)
            rough_tca: Rough time of closest approach
            search_window_minutes: Search window around rough TCA

        Returns:
            Refined TCA, miss distance, and relative geometry
        """
        from .propagate import SGP4Propagator
        propagator = SGP4Propagator()

        # Define objective function: minimize relative distance
        def relative_distance(dt_minutes):
            target_time = rough_tca + timedelta(minutes=dt_minutes)

            try:
                # Propagate both objects
                pos1, vel1, _ = propagator.propagate(
                    obj1_tle['line1'], obj1_tle['line2'], target_time
                )
                pos2, vel2, _ = propagator.propagate(
                    obj2_tle['line1'], obj2_tle['line2'], target_time
                )

                # Calculate relative distance
                rel_pos = pos2 - pos1
                return np.linalg.norm(rel_pos)

            except Exception as e:
                logger.warning(f"Refinement propagation failed: {e}")
                return 1e6  # Large penalty for failed propagation

        # Minimize relative distance within search window
        result = minimize_scalar(
            relative_distance,
            bounds=(-search_window_minutes/2, search_window_minutes/2),
            method='bounded'
        )

        if not result.success:
```

```
            logger.warning("Refinement optimization failed")
            return None

        # Calculate refined results
        refined_tca = rough_tca + timedelta(minutes=result.x)
        miss_distance = result.fun

        # Get state vectors at TCA
        try:
            pos1, vel1, cov1 = propagator.propagate(
                obj1_tle['line1'], obj1_tle['line2'], refined_tca
            )
            pos2, vel2, cov2 = propagator.propagate(
                obj2_tle['line1'], obj2_tle['line2'], refined_tca
            )

            # Transform to Hill frame (radial, in-track, cross-track)
            rel_pos, rel_vel = self._to_hill_frame(pos1, vel1, pos2, vel2)

            return {
                'tca': refined_tca,
                'miss_distance': miss_distance,
                'relative_position_hill': rel_pos.tolist(),
                'relative_velocity_hill': rel_vel.tolist(),
                'primary_position': pos1.tolist(),
                'secondary_position': pos2.tolist(),
                'optimization_success': True
            }

        except Exception as e:
            logger.error(f"Hill frame calculation failed: {e}")
            return {
                'tca': refined_tca,
                'miss_distance': miss_distance,
                'optimization_success': False,
                'error': str(e)
            }

    def _to_hill_frame(self, pos1, vel1, pos2, vel2):
        """Transform to Hill coordinate frame (RIC - Radial, In-track, Cross-track)"""

        # Primary object vectors
        r1 = pos1
        v1 = vel1

        # Relative vectors
        rel_pos = pos2 - pos1
        rel_vel = vel2 - vel1

        # Hill frame unit vectors
        r_unit = r1 / np.linalg.norm(r1)  # Radial (toward Earth center)
        h = np.cross(r1, v1)
        h_unit = h / np.linalg.norm(h)    # Cross-track (normal to orbit plane)
        i_unit = np.cross(h_unit, r_unit) # In-track (along velocity direction)

        # Transform matrix from inertial to Hill frame
```

```
        T_hill = np.array([r_unit, i_unit, h_unit])

        # Transform relative position and velocity
        rel_pos_hill = T_hill @ rel_pos
        rel_vel_hill = T_hill @ rel_vel

        return rel_pos_hill, rel_vel_hill
```

=== math/pc.py ===

```
import numpy as np
from scipy.stats import multivariate_normal
from scipy.integrate import dblquad
import logging

logger = logging.getLogger(__name__)

class ProbabilityCalculator:
    """Probability of Collision (Pc) calculator using 2D Gaussian method"""

    def __init__(self, hard_body_radius=10.0):  # meters
        self.hard_body_radius = hard_body_radius / 1000.0  # convert to km
        self.calculation_count = 0

    def calculate(self, obj1_state, obj2_state):
        """
        Calculate probability of collision using 2D Gaussian method

        Args:
            obj1_state: State vector with position, velocity, covariance
            obj2_state: State vector with position, velocity, covariance

        Returns:
            Pc value, TCA, miss distance
        """
        try:
            # Extract state information
            pos1 = np.array(obj1_state['position'])
            vel1 = np.array(obj1_state['velocity'])
            cov1 = np.array(obj1_state.get('covariance', np.eye(6) * 0.01))

            pos2 = np.array(obj2_state['position'])
            vel2 = np.array(obj2_state['velocity'])
            cov2 = np.array(obj2_state.get('covariance', np.eye(6) * 0.01))

            # Calculate relative state
            rel_pos = pos2 - pos1
            rel_vel = vel2 - vel1

            # Combined covariance matrix
            rel_cov = cov1 + cov2

            # Miss distance at TCA (assuming linear motion)
            miss_distance = np.linalg.norm(rel_pos)
```

```
            # Time of closest approach (TCA) calculation
            if np.dot(rel_pos, rel_vel) < 0:
                # Objects are approaching
                t_tca = -np.dot(rel_pos, rel_vel) / np.dot(rel_vel, rel_vel)
                pos_at_tca = rel_pos + rel_vel * t_tca
                miss_distance = np.linalg.norm(pos_at_tca)
            else:
                t_tca = 0
                pos_at_tca = rel_pos

            # Project to encounter plane (perpendicular to relative velocity)
            if np.linalg.norm(rel_vel) > 1e-6:
                # Create encounter plane coordinate system
                u_vel = rel_vel / np.linalg.norm(rel_vel)  # Along relative velocity

                # Find two orthogonal vectors in encounter plane
                if abs(u_vel[0]) < 0.9:
                    u_perp1 = np.cross(u_vel, [1, 0, 0])
                else:
                    u_perp1 = np.cross(u_vel, [0, 1, 0])
                u_perp1 = u_perp1 / np.linalg.norm(u_perp1)

                u_perp2 = np.cross(u_vel, u_perp1)
                u_perp2 = u_perp2 / np.linalg.norm(u_perp2)

                # Projection matrix to encounter plane
                P = np.array([u_perp1, u_perp2])  # 2x3 matrix

                # Project relative position and covariance to 2D encounter plane
                rel_pos_2d = P @ pos_at_tca
                rel_cov_2d = P @ rel_cov[:3, :3] @ P.T

                # Calculate Pc using 2D circular collision area
                pc = self._calculate_2d_pc(rel_pos_2d, rel_cov_2d)

            else:
                # Objects not moving relative to each other
                pc = 1.0 if miss_distance < self.hard_body_radius else 0.0

            self.calculation_count += 1

            from datetime import datetime, timedelta
            tca = datetime.utcnow() + timedelta(seconds=t_tca)

            return pc, tca, miss_distance

        except Exception as e:
            logger.error(f"Pc calculation failed: {e}")
            raise

    def _calculate_2d_pc(self, rel_pos_2d, rel_cov_2d):
        """Calculate Pc using 2D Gaussian integration over circular collision area"""

        try:
            # Check for singular covariance matrix
            det_cov = np.linalg.det(rel_cov_2d)
```

```
            if det_cov <= 1e-12:
                logger.warning("Singular covariance matrix, using deterministic calculati
                distance = np.linalg.norm(rel_pos_2d)
                return 1.0 if distance < self.hard_body_radius else 0.0

            # Create multivariate normal distribution
            rv = multivariate_normal(mean=rel_pos_2d, cov=rel_cov_2d)

            # Integration limits for circular collision area
            x_center, y_center = rel_pos_2d
            radius = self.hard_body_radius

            # Determine integration bounds
            x_min = x_center - 3*radius
            x_max = x_center + 3*radius

            def y_bounds(x):
                # Circular boundary
                dx = x - x_center
                if abs(dx) > radius:
                    return 0, 0  # Outside circle
                dy_max = np.sqrt(radius**2 - dx**2)
                return y_center - dy_max, y_center + dy_max

            def integrand(y, x):
                # Check if point is inside collision circle
                if (x - x_center)**2 + (y - y_center)**2 <= radius**2:
                    return rv.pdf([x, y])
                else:
                    return 0.0

            # Numerical integration
            pc, _ = dblquad(integrand, x_min, x_max, y_bounds)

            # Clamp to [0, 1] range
            pc = max(0.0, min(1.0, pc))

            return pc

        except Exception as e:
            logger.error(f"2D Pc integration failed: {e}")
            # Fallback to analytical approximation
            distance = np.linalg.norm(rel_pos_2d)
            sigma = np.sqrt(np.trace(rel_cov_2d) / 2)  # RMS uncertainty

            if sigma > 0:
                # Approximate using distance vs uncertainty ratio
                ratio = distance / sigma
                if ratio < 1:
                    return 0.5 * np.exp(-(ratio**2) / 2)
                else:
                    return np.exp(-(ratio**2) / 8)  # Crude approximation
            else:
                return 1.0 if distance < self.hard_body_radius else 0.0

    def calculate_batch(self, conjunction_list):
```

```
        """Calculate Pc for multiple conjunctions"""
        results = []

        for conjunction in conjunction_list:
            try:
                pc, tca, miss_dist = self.calculate(
                    conjunction['object1'],
                    conjunction['object2']
                )
                results.append({
                    'conjunction_id': conjunction.get('id'),
                    'probability': pc,
                    'tca': tca.isoformat(),
                    'miss_distance': miss_dist,
                    'success': True
                })
            except Exception as e:
                results.append({
                    'conjunction_id': conjunction.get('id'),
                    'error': str(e),
                    'success': False
                })

        return results

    def status(self):
        """Get calculator status"""
        return {
            'status': 'operational',
            'calculations_performed': self.calculation_count,
            'hard_body_radius_m': self.hard_body_radius * 1000
        }
```

=== backend/realtime.py ===

```
import asyncio
import time
import logging
from datetime import datetime, timedelta
from concurrent.futures import ThreadPoolExecutor
import json
from pathlib import Path

logger = logging.getLogger(__name__)

class RealtimeProcessor:
    """Real-time processing loop with <150ms latency target and fallback mechanisms"""

    def __init__(self, target_latency_ms=150, fallback_enabled=True):
        self.target_latency_ms = target_latency_ms
        self.fallback_enabled = fallback_enabled
        self.is_running = False
        self.stats = {
            'cycles_completed': 0,
            'average_latency_ms': 0,
```

```
            'errors': 0,
            'fallback_activations': 0,
            'last_cycle': None
        }
        self.executor = ThreadPoolExecutor(max_workers=4)

    async def start_processing_loop(self):
        """Start the main processing loop"""
        self.is_running = True
        logger.info("Starting real-time processing loop")

        while self.is_running:
            cycle_start = time.time()

            try:
                # Main processing cycle
                await self._process_cycle()

                cycle_time_ms = (time.time() - cycle_start) * 1000

                # Update statistics
                self._update_stats(cycle_time_ms, success=True)

                # Check latency target
                if cycle_time_ms > self.target_latency_ms:
                    logger.warning(f"Cycle exceeded target latency: {cycle_time_ms:.1f}ms

                # Sleep to maintain cycle rate (aim for 10Hz = 100ms cycles)
                sleep_time = max(0, 0.1 - (time.time() - cycle_start))
                if sleep_time > 0:
                    await asyncio.sleep(sleep_time)

            except Exception as e:
                logger.error(f"Processing cycle failed: {e}")
                self._update_stats(0, success=False)

                if self.fallback_enabled:
                    await self._activate_fallback()

                await asyncio.sleep(1)  # Error recovery delay

    async def _process_cycle(self):
        """Single processing cycle"""
        # 1. Check for new TLE data
        tle_status = await self._check_tle_freshness()

        # 2. Update active object catalog
        active_objects = await self._get_active_objects()

        # 3. Run conjunction screening (most expensive operation)
        if len(active_objects) > 0:
            conjunctions = await self._run_screening(active_objects)

            # 4. Calculate Pc for high-priority conjunctions
            if conjunctions:
                pc_results = await self._calculate_priority_pc(conjunctions)
```

```
            # 5. Update alerts and notifications
            await self._update_alerts(pc_results)

        # 6. Log cycle completion
        self.stats['last_cycle'] = datetime.utcnow().isoformat()

    async def _check_tle_freshness(self):
        """Check if TLE data needs updating"""
        try:
            from .ingest import TLEIngester
            ingester = TLEIngester()

            # Run in thread pool to avoid blocking
            status = await asyncio.get_event_loop().run_in_executor(
                self.executor, ingester.status
            )

            return status
        except Exception as e:
            logger.error(f"TLE freshness check failed: {e}")
            return {'status': 'error'}

    async def _get_active_objects(self):
        """Get list of active objects to track"""
        try:
            # Load from cache or database
            cache_file = Path("./data/active_objects.json")

            if cache_file.exists():
                with open(cache_file, 'r') as f:
                    data = json.load(f)
                    return data.get('objects', [])[:100]  # Limit to 100 for real-time

            return []
        except Exception as e:
            logger.error(f"Failed to load active objects: {e}")
            return []

    async def _run_screening(self, objects):
        """Run conjunction screening with timeout"""
        try:
            from .screen import ConjunctionScreener
            screener = ConjunctionScreener(screening_distance=2.0)  # Tighter for real-ti

            # Run screening with timeout
            screening_task = asyncio.get_event_loop().run_in_executor(
                self.executor, screener.screen, objects, 6  # 6-hour window
            )

            # 100ms timeout for screening
            conjunctions = await asyncio.wait_for(screening_task, timeout=0.1)

            return conjunctions

        except asyncio.TimeoutError:
```

```
            logger.warning("Screening timed out, activating fallback")
            if self.fallback_enabled:
                return await self._fallback_screening(objects)
            return []
        except Exception as e:
            logger.error(f"Screening failed: {e}")
            return []

    async def _calculate_priority_pc(self, conjunctions):
        """Calculate Pc for highest priority conjunctions only"""
        try:
            from math.pc import ProbabilityCalculator
            calculator = ProbabilityCalculator()

            # Sort by distance and take top 5
            priority_conjunctions = sorted(conjunctions, key=lambda x: x['distance'])[:5]

            pc_task = asyncio.get_event_loop().run_in_executor(
                self.executor, calculator.calculate_batch, priority_conjunctions
            )

            # 30ms timeout for Pc calculations
            results = await asyncio.wait_for(pc_task, timeout=0.03)
            return results

        except asyncio.TimeoutError:
            logger.warning("Pc calculation timed out")
            return []
        except Exception as e:
            logger.error(f"Pc calculation failed: {e}")
            return []

    async def _update_alerts(self, pc_results):
        """Update alert system with latest results"""
        try:
            high_risk_threshold = 1e-4  # 1 in 10,000

            alerts = []
            for result in pc_results:
                if result.get('success') and result.get('probability', 0) > high_risk_thr
                    alerts.append({
                        'type': 'HIGH_PC',
                        'probability': result['probability'],
                        'tca': result['tca'],
                        'timestamp': datetime.utcnow().isoformat()
                    })

            if alerts:
                # Save alerts to file (could be database in production)
                alerts_file = Path("./data/active_alerts.json")
                with open(alerts_file, 'w') as f:
                    json.dump({'alerts': alerts, 'updated': datetime.utcnow().isoformat()

                logger.warning(f"Generated {len(alerts)} high-risk alerts")

        except Exception as e:
```

```
            logger.error(f"Alert update failed: {e}")

    async def _activate_fallback(self):
        """Activate fallback processing mode"""
        logger.warning("Activating fallback processing mode")
        self.stats['fallback_activations'] += 1

        # Simple fallback: check only critical satellites
        critical_objects = await self._get_critical_objects()

        if critical_objects:
            # Use degraded but fast processing
            from .degrade_policy import DegradationPolicy
            policy = DegradationPolicy()

            degraded_results = await asyncio.get_event_loop().run_in_executor(
                self.executor, policy.process_degraded, critical_objects
            )

            await self._update_alerts(degraded_results)

    async def _fallback_screening(self, objects):
        """Fast fallback screening method"""
        # Simplified distance-only screening
        conjunctions = []

        # Only check first 20 objects for speed
        limited_objects = objects[:20]

        for i in range(len(limited_objects)):
            for j in range(i+1, len(limited_objects)):
                # Very rough distance check without propagation
                try:
                    # Use TLE epoch positions as approximation
                    obj1 = limited_objects[i]
                    obj2 = limited_objects[j]

                    # Extract rough positions from TLE (this is a simplification)
                    # In reality would need basic orbital elements

                    conjunctions.append({
                        'object1': obj1,
                        'object2': obj2,
                        'distance': 5.0,  # Placeholder
                        'fallback': True
                    })

                    if len(conjunctions) >= 5:  # Limit for speed
                        break

                except Exception:
                    continue

            if len(conjunctions) >= 5:
                break
```

```
        return conjunctions

    async def _get_critical_objects(self):
        """Get list of critical/high-priority objects"""
        # In production, this would query a database of critical satellites
        return []  # Placeholder

    def _update_stats(self, cycle_time_ms, success=True):
        """Update processing statistics"""
        if success:
            self.stats['cycles_completed'] += 1

            # Update rolling average latency
            old_avg = self.stats['average_latency_ms']
            n = self.stats['cycles_completed']
            self.stats['average_latency_ms'] = ((n-1) * old_avg + cycle_time_ms) / n
        else:
            self.stats['errors'] += 1

    def stop_processing(self):
        """Stop the processing loop"""
        self.is_running = False
        logger.info("Stopping real-time processing loop")

    def get_status(self):
        """Get current processing status"""
        return {
            'running': self.is_running,
            'target_latency_ms': self.target_latency_ms,
            'statistics': self.stats.copy(),
            'health': 'healthy' if self.stats['average_latency_ms'] < self.target_latency
        }
```

=== backend/degrade_policy.py ===

```
import numpy as np
import logging
from datetime import datetime, timedelta

logger = logging.getLogger(__name__)

class DegradationPolicy:
    """Handles system degradation scenarios and fallback modes"""

    def __init__(self):
        self.degradation_active = False
        self.degradation_reasons = []

    def assess_degradation_triggers(self, system_status):
        """Check if degradation mode should be activated"""
        triggers = []

        # Check TLE age
        tle_status = system_status.get('tle_status', {})
        if 'last_update' in tle_status:
```

```
            last_update = datetime.fromisoformat(tle_status['last_update'])
            age_hours = (datetime.utcnow() - last_update).total_seconds() / 3600

            if age_hours > 24:
                triggers.append(f"TLE_STALE_{age_hours:.1f}h")
            if age_hours > 72:
                triggers.append("TLE_VERY_STALE")

        # Check processing latency
        realtime_status = system_status.get('realtime_status', {})
        avg_latency = realtime_status.get('average_latency_ms', 0)
        if avg_latency > 500:  # 500ms threshold
            triggers.append(f"HIGH_LATENCY_{avg_latency:.0f}ms")

        # Check error rates
        error_count = realtime_status.get('errors', 0)
        cycle_count = realtime_status.get('cycles_completed', 1)
        error_rate = error_count / cycle_count

        if error_rate > 0.1:  # 10% error rate
            triggers.append(f"HIGH_ERROR_RATE_{error_rate:.1%}")

        # Solar weather check (placeholder - would integrate with space weather APIs)
        if self._check_solar_weather():
            triggers.append("SOLAR_STORM")

        # Update degradation state
        if triggers:
            if not self.degradation_active:
                logger.warning(f"Activating degradation mode: {', '.join(triggers)}")
                self.degradation_active = True
            self.degradation_reasons = triggers
        else:
            if self.degradation_active:
                logger.info("Deactivating degradation mode - conditions improved")
                self.degradation_active = False
                self.degradation_reasons = []

        return triggers

    def apply_degradation_policy(self, prediction_result):
        """Apply degradation policy to prediction results"""
        if not self.degradation_active:
            return prediction_result

        degraded_result = prediction_result.copy()

        # Widen uncertainties
        if 'uncertainty' in degraded_result:
            uncertainty = np.array(degraded_result['uncertainty'])
            degraded_result['uncertainty'] = (uncertainty * 2.0).tolist()  # Double unce

        if 'covariance' in degraded_result:
            covariance = np.array(degraded_result['covariance'])
            degraded_result['covariance'] = (covariance * 4.0).tolist()  # 4x covariance
```

```
        # Add degradation metadata
        degraded_result['degraded'] = True
        degraded_result['degradation_reasons'] = self.degradation_reasons.copy()
        degraded_result['confidence_factor'] = 0.5  # Reduced confidence

        # Tag timestamp
        degraded_result['degradation_applied_at'] = datetime.utcnow().isoformat()

        return degraded_result

    def calculate_fallback_pc(self, miss_distance, baseline_uncertainty=1.0):
        """Calculate conservative fallback Pc when normal calculation fails"""

        # Conservative hard-body radius (larger)
        fallback_radius = 0.050  # 50m in km

        # Conservative uncertainty model
        if self.degradation_active:
            uncertainty_multiplier = 3.0
        else:
            uncertainty_multiplier = 1.5

        effective_uncertainty = baseline_uncertainty * uncertainty_multiplier

        # Simple exponential falloff model
        if miss_distance < fallback_radius:
            pc = 0.1  # 10% baseline risk for very close approaches
        else:
            # Exponential decay with distance
            pc = 0.1 * np.exp(-(miss_distance - fallback_radius) / effective_uncertainty)

        # Clamp to reasonable range
        pc = max(1e-8, min(0.1, pc))

        return {
            'probability': pc,
            'method': 'FALLBACK',
            'degraded': True,
            'conservative': True,
            'baseline_uncertainty': baseline_uncertainty,
            'effective_uncertainty': effective_uncertainty,
            'timestamp': datetime.utcnow().isoformat()
        }

    def process_degraded(self, objects):
        """Process objects in degraded mode with conservative estimates"""
        results = []

        logger.info(f"Processing {len(objects)} objects in degraded mode")

        # Use simple pairwise distance checks
        for i in range(len(objects)):
            for j in range(i+1, len(objects)):
                try:
                    obj1 = objects[i]
                    obj2 = objects[j]
```

```
                    # Simple distance estimation (would be replaced with actual calculati
                    estimated_distance = np.random.uniform(1.0, 10.0)  # Placeholder

                    # Conservative Pc calculation
                    pc_result = self.calculate_fallback_pc(estimated_distance)

                    results.append({
                        'object1_name': obj1.get('name', 'Unknown'),
                        'object2_name': obj2.get('name', 'Unknown'),
                        'distance_estimate': estimated_distance,
                        'pc_result': pc_result,
                        'processing_mode': 'DEGRADED',
                        'success': True
                    })

                    # Limit results to prevent overload
                    if len(results) >= 10:
                        break

                except Exception as e:
                    logger.error(f"Degraded processing failed for pair: {e}")
                    continue

            if len(results) >= 10:
                break

        return results

    def _check_solar_weather(self):
        """Check for solar weather conditions that affect orbit prediction"""
        # Placeholder - in production would check NOAA space weather APIs
        # For now, randomly simulate solar storms
        import random
        return random.random() < 0.05  # 5% chance of solar storm

    def get_degradation_status(self):
        """Get current degradation status"""
        return {
            'active': self.degradation_active,
            'reasons': self.degradation_reasons.copy(),
            'timestamp': datetime.utcnow().isoformat()
        }
```

=== ml/model.py ===

```
import torch
import torch.nn as nn
import numpy as np
from datetime import datetime
import logging

logger = logging.getLogger(__name__)

class ResidualPredictionModel(nn.Module):
```

```
    """LSTM-based model for predicting SGP4 residuals with uncertainty"""

    def __init__(self, input_dim=12, hidden_dim=64, num_layers=2, output_dim=6):
        super(ResidualPredictionModel, self).__init__()

        self.hidden_dim = hidden_dim
        self.num_layers = num_layers

        # Input: [tle_features, orbital_elements, time_features, solar_indices]
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers, batch_first=True, dropout=

        # Separate heads for mean and log-variance prediction
        self.mean_head = nn.Sequential(
            nn.Linear(hidden_dim, 32),
            nn.ReLU(),
            nn.Dropout(0.1),
            nn.Linear(32, output_dim)  # [dx, dy, dz, dvx, dvy, dvz]
        )

        self.log_var_head = nn.Sequential(
            nn.Linear(hidden_dim, 32),
            nn.ReLU(),
            nn.Dropout(0.1),
            nn.Linear(32, output_dim)  # log-variance for uncertainty
        )

        self.model_info = {
            'created_at': datetime.utcnow().isoformat(),
            'version': '1.0',
            'input_dim': input_dim,
            'hidden_dim': hidden_dim,
            'num_layers': num_layers
        }

    def forward(self, x):
        """Forward pass returning mean and log-variance"""
        # x shape: (batch_size, sequence_length, input_dim)

        lstm_out, _ = self.lstm(x)

        # Use last time step output
        last_output = lstm_out[:, -1, :]  # (batch_size, hidden_dim)

        # Predict mean residuals
        mean = self.mean_head(last_output)

        # Predict log-variance for uncertainty quantification
        log_var = self.log_var_head(last_output)

        return mean, log_var

    def predict_with_uncertainty(self, x, num_samples=100):
        """Predict with Monte Carlo uncertainty estimation"""
        self.eval()

        with torch.no_grad():
```

```
            mean, log_var = self.forward(x)

            # Sample from predicted distribution
            std = torch.exp(0.5 * log_var)

            samples = []
            for _ in range(num_samples):
                noise = torch.randn_like(mean)
                sample = mean + std * noise
                samples.append(sample)

            samples = torch.stack(samples, dim=0)  # (num_samples, batch_size, output_dim

            # Calculate statistics
            pred_mean = samples.mean(dim=0)
            pred_std = samples.std(dim=0)

            return pred_mean, pred_std, samples

class SGP4MLPredictor:
    """Wrapper combining SGP4 baseline with ML residual correction"""

    def __init__(self, model_path=None):
        self.sgp4_propagator = None  # Will be injected
        self.ml_model = None
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

        if model_path:
            self.load_model(model_path)

    def load_model(self, model_path):
        """Load trained ML model"""
        try:
            checkpoint = torch.load(model_path, map_location=self.device)

            # Reconstruct model from saved config
            config = checkpoint.get('config', {})
            self.ml_model = ResidualPredictionModel(
                input_dim=config.get('input_dim', 12),
                hidden_dim=config.get('hidden_dim', 64),
                num_layers=config.get('num_layers', 2),
                output_dim=config.get('output_dim', 6)
            )

            self.ml_model.load_state_dict(checkpoint['model_state_dict'])
            self.ml_model.to(self.device)
            self.ml_model.eval()

            logger.info(f"Loaded ML model from {model_path}")
            return True

        except Exception as e:
            logger.error(f"Failed to load ML model: {e}")
            return False

    def predict_corrected(self, tle_line1, tle_line2, target_time, use_ml=True):
```

```
        """Predict orbit with SGP4 + ML correction"""

        # Get SGP4 baseline prediction
        from backend.propagate import SGP4Propagator
        if self.sgp4_propagator is None:
            self.sgp4_propagator = SGP4Propagator()

        sgp4_pos, sgp4_vel, sgp4_cov = self.sgp4_propagator.propagate(
            tle_line1, tle_line2, target_time
        )

        if not use_ml or self.ml_model is None:
            return sgp4_pos, sgp4_vel, sgp4_cov, {'method': 'SGP4_ONLY'}

        try:
            # Prepare ML input features
            features = self._extract_features(tle_line1, tle_line2, target_time)

            # Predict residuals with uncertainty
            mean_residual, std_residual, _ = self.ml_model.predict_with_uncertainty(featu

            # Apply correction to SGP4 prediction
            corrected_pos = sgp4_pos + mean_residual[0, :3].cpu().numpy()
            corrected_vel = sgp4_vel + mean_residual[0, 3:].cpu().numpy()

            # Enhanced uncertainty estimate combining SGP4 + ML uncertainty
            ml_uncertainty = np.diag(std_residual[0].cpu().numpy()**2)
            corrected_cov = sgp4_cov + ml_uncertainty

            metadata = {
                'method': 'SGP4_ML_CORRECTED',
                'ml_correction': mean_residual[0].cpu().numpy().tolist(),
                'ml_uncertainty': std_residual[0].cpu().numpy().tolist(),
                'timestamp': datetime.utcnow().isoformat()
            }

            return corrected_pos, corrected_vel, corrected_cov, metadata

        except Exception as e:
            logger.error(f"ML correction failed, falling back to SGP4: {e}")
            return sgp4_pos, sgp4_vel, sgp4_cov, {'method': 'SGP4_FALLBACK', 'ml_error':

    def _extract_features(self, tle_line1, tle_line2, target_time):
        """Extract features for ML model input"""

        # Parse TLE elements (simplified)
        try:
            from sgp4.api import Satrec
            satellite = Satrec.twoline2rv(tle_line1, tle_line2)

            # Basic orbital elements
            features = [
                satellite.ecco,      # Eccentricity
                satellite.inclo,     # Inclination
                satellite.no_kozai,  # Mean motion
                satellite.argpo,     # Argument of perigee
```

```
                satellite.mo,        # Mean anomaly
                satellite.nodeo,     # RAAN
                satellite.bstar,     # B* drag term
                satellite.ndot,      # First derivative of mean motion
                satellite.nddot,     # Second derivative of mean motion
            ]

            # Time-based features
            epoch_jd = satellite.jdsatepoch + satellite.jdsatepochF
            target_jd = target_time.timestamp() / 86400.0 + 2440587.5  # Convert to Julia
            time_since_epoch = target_jd - epoch_jd

            features.extend([
                time_since_epoch,
                np.sin(2 * np.pi * time_since_epoch),  # Periodic time encoding
                np.cos(2 * np.pi * time_since_epoch)
            ])

            # Convert to tensor
            features_tensor = torch.tensor(features, dtype=torch.float32, device=self.dev

            # Reshape for LSTM: (batch_size=1, sequence_length=1, input_dim)
            features_tensor = features_tensor.unsqueeze(0).unsqueeze(0)

            return features_tensor

        except Exception as e:
            logger.error(f"Feature extraction failed: {e}")
            # Return dummy features if extraction fails
            dummy_features = torch.zeros(1, 1, 12, device=self.device)
            return dummy_features

    def benchmark_against_truth(self, test_cases):
        """Benchmark SGP4 vs SGP4+ML against truth data"""

        results = {
            'sgp4_errors': [],
            'ml_corrected_errors': [],
            'test_cases': len(test_cases)
        }

        for case in test_cases:
            try:
                # Get truth position
                truth_pos = np.array(case['truth_position'])
                truth_vel = np.array(case['truth_velocity'])

                # SGP4 prediction
                sgp4_pos, sgp4_vel, _, _ = self.predict_corrected(
                    case['tle_line1'], case['tle_line2'],
                    case['target_time'], use_ml=False
                )

                # ML corrected prediction
                ml_pos, ml_vel, _, _ = self.predict_corrected(
                    case['tle_line1'], case['tle_line2'],
```

```
                    case['target_time'], use_ml=True
                )

                # Calculate errors
                sgp4_pos_error = np.linalg.norm(sgp4_pos - truth_pos)
                ml_pos_error = np.linalg.norm(ml_pos - truth_pos)

                results['sgp4_errors'].append(sgp4_pos_error)
                results['ml_corrected_errors'].append(ml_pos_error)

            except Exception as e:
                logger.error(f"Benchmark case failed: {e}")
                continue

        # Calculate statistics
        if results['sgp4_errors'] and results['ml_corrected_errors']:
            results['sgp4_mae'] = np.mean(results['sgp4_errors'])
            results['sgp4_rmse'] = np.sqrt(np.mean([e**2 for e in results['sgp4_errors']]
            results['ml_mae'] = np.mean(results['ml_corrected_errors'])
            results['ml_rmse'] = np.sqrt(np.mean([e**2 for e in results['ml_corrected_err

            improvement = (results['sgp4_mae'] - results['ml_mae']) / results['sgp4_mae']
            results['improvement_percent'] = improvement * 100

        return results
```

=== ml/train.py ===

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import numpy as np
from datetime import datetime
import json
import logging
from .model import ResidualPredictionModel

logger = logging.getLogger(__name__)

class OrbitResidualDataset(Dataset):
    """Dataset for training orbital residual prediction model"""

    def __init__(self, data_file):
        self.data = self._load_data(data_file)

    def _load_data(self, data_file):
        """Load training data from file"""
        try:
            with open(data_file, 'r') as f:
                raw_data = json.load(f)

            processed_data = []
            for sample in raw_data:
                # Extract features and targets
```

```
                features = np.array(sample['features'], dtype=np.float32)
                sgp4_prediction = np.array(sample['sgp4_prediction'], dtype=np.float32)
                truth_state = np.array(sample['truth_state'], dtype=np.float32)

                # Calculate residual (truth - SGP4)
                residual = truth_state - sgp4_prediction

                processed_data.append({
                    'features': features,
                    'residual': residual
                })

            logger.info(f"Loaded {len(processed_data)} training samples")
            return processed_data

        except Exception as e:
            logger.error(f"Failed to load training data: {e}")
            return []

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        sample = self.data[idx]
        return torch.tensor(sample['features']), torch.tensor(sample['residual'])

class OrbitTrainer:
    """Training class for orbital residual prediction model"""

    def __init__(self, config=None):
        self.config = config or self._default_config()
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.model = None
        self.training_history = []

    def _default_config(self):
        return {
            'input_dim': 12,
            'hidden_dim': 64,
            'num_layers': 2,
            'output_dim': 6,
            'learning_rate': 0.001,
            'batch_size': 32,
            'num_epochs': 100,
            'validation_split': 0.2,
            'early_stopping_patience': 10,
            'weight_decay': 1e-5
        }

    def train(self, train_data_file, save_path='./models/orbit_residual_model.pth'):
        """Train the orbital residual prediction model"""

        # Load dataset
        dataset = OrbitResidualDataset(train_data_file)

        if len(dataset) == 0:
```

```
            raise ValueError("No training data available")

        # Split into train/validation
        val_size = int(len(dataset) * self.config['validation_split'])
        train_size = len(dataset) - val_size

        train_dataset, val_dataset = torch.utils.data.random_split(
            dataset, [train_size, val_size]
        )

        # Create data loaders
        train_loader = DataLoader(
            train_dataset,
            batch_size=self.config['batch_size'],
            shuffle=True
        )
        val_loader = DataLoader(
            val_dataset,
            batch_size=self.config['batch_size'],
            shuffle=False
        )

        # Initialize model
        self.model = ResidualPredictionModel(
            input_dim=self.config['input_dim'],
            hidden_dim=self.config['hidden_dim'],
            num_layers=self.config['num_layers'],
            output_dim=self.config['output_dim']
        ).to(self.device)

        # Loss function and optimizer
        optimizer = optim.Adam(
            self.model.parameters(),
            lr=self.config['learning_rate'],
            weight_decay=self.config['weight_decay']
        )

        # Training loop
        best_val_loss = float('inf')
        patience_counter = 0

        logger.info(f"Starting training on {self.device}")

        for epoch in range(self.config['num_epochs']):
            # Training phase
            train_loss = self._train_epoch(train_loader, optimizer)

            # Validation phase
            val_loss, val_metrics = self._validate_epoch(val_loader)

            # Learning rate scheduling
            if epoch > 20 and val_loss > best_val_loss:
                for param_group in optimizer.param_groups:
                    param_group['lr'] *= 0.95  # Decay learning rate

            # Early stopping
```

```
            if val_loss < best_val_loss:
                best_val_loss = val_loss
                patience_counter = 0

                # Save best model
                self._save_model(save_path, epoch, val_loss)

            else:
                patience_counter += 1

            # Log progress
            if epoch % 10 == 0:
                logger.info(
                    f"Epoch {epoch}: Train Loss={train_loss:.6f}, "
                    f"Val Loss={val_loss:.6f}, Best Val Loss={best_val_loss:.6f}"
                )

            # Record training history
            self.training_history.append({
                'epoch': epoch,
                'train_loss': train_loss,
                'val_loss': val_loss,
                'val_metrics': val_metrics
            })

            # Early stopping
            if patience_counter >= self.config['early_stopping_patience']:
                logger.info(f"Early stopping at epoch {epoch}")
                break

        logger.info("Training completed")
        return self.training_history

    def _train_epoch(self, train_loader, optimizer):
        """Train for one epoch"""
        self.model.train()
        total_loss = 0

        for batch_features, batch_targets in train_loader:
            batch_features = batch_features.to(self.device)
            batch_targets = batch_targets.to(self.device)

            # Add sequence dimension for LSTM
            batch_features = batch_features.unsqueeze(1)  # (batch, 1, features)

            optimizer.zero_grad()

            # Forward pass
            mean_pred, log_var_pred = self.model(batch_features)

            # Gaussian negative log-likelihood loss
            loss = self._gaussian_nll_loss(batch_targets, mean_pred, log_var_pred)

            # Backward pass
            loss.backward()
```

```
            # Gradient clipping
            torch.nn.utils.clip_grad_norm_(self.model.parameters(), max_norm=1.0)

            optimizer.step()

            total_loss += loss.item()

        return total_loss / len(train_loader)

    def _validate_epoch(self, val_loader):
        """Validate for one epoch"""
        self.model.eval()
        total_loss = 0
        mae_errors = []

        with torch.no_grad():
            for batch_features, batch_targets in val_loader:
                batch_features = batch_features.to(self.device)
                batch_targets = batch_targets.to(self.device)

                batch_features = batch_features.unsqueeze(1)

                # Forward pass
                mean_pred, log_var_pred = self.model(batch_features)

                # Calculate loss
                loss = self._gaussian_nll_loss(batch_targets, mean_pred, log_var_pred)
                total_loss += loss.item()

                # Calculate MAE for interpretability
                mae = torch.mean(torch.abs(batch_targets - mean_pred))
                mae_errors.append(mae.item())

        avg_loss = total_loss / len(val_loader)
        avg_mae = np.mean(mae_errors)

        metrics = {
            'mae': avg_mae,
            'loss': avg_loss
        }

        return avg_loss, metrics

    def _gaussian_nll_loss(self, targets, mean_pred, log_var_pred):
        """Gaussian negative log-likelihood loss"""

        # Ensure numerical stability
        log_var_pred = torch.clamp(log_var_pred, min=-10, max=10)

        # NLL = 0.5 * (log(2π) + log_var + (target - mean)² / var)
        mse = (targets - mean_pred) ** 2
        var_pred = torch.exp(log_var_pred)

        nll = 0.5 * (np.log(2 * np.pi) + log_var_pred + mse / var_pred)

        return torch.mean(nll)
```

```
    def _save_model(self, save_path, epoch, val_loss):
        """Save model checkpoint"""
        import os
        os.makedirs(os.path.dirname(save_path), exist_ok=True)

        checkpoint = {
            'model_state_dict': self.model.state_dict(),
            'config': self.config,
            'epoch': epoch,
            'val_loss': val_loss,
            'training_history': self.training_history,
            'saved_at': datetime.utcnow().isoformat()
        }

        torch.save(checkpoint, save_path)
        logger.info(f"Model saved to {save_path}")

    def calculate_calibration_metrics(self, test_loader):
        """Calculate calibration metrics (ECE, CRPS)"""
        self.model.eval()

        predictions = []
        targets = []
        uncertainties = []

        with torch.no_grad():
            for batch_features, batch_targets in test_loader:
                batch_features = batch_features.to(self.device).unsqueeze(1)
                batch_targets = batch_targets.to(self.device)

                mean_pred, log_var_pred = self.model(batch_features)
                std_pred = torch.exp(0.5 * log_var_pred)

                predictions.append(mean_pred.cpu().numpy())
                targets.append(batch_targets.cpu().numpy())
                uncertainties.append(std_pred.cpu().numpy())

        predictions = np.concatenate(predictions)
        targets = np.concatenate(targets)
        uncertainties = np.concatenate(uncertainties)

        # Expected Calibration Error (ECE)
        ece = self._calculate_ece(predictions, targets, uncertainties)

        # Continuous Ranked Probability Score (CRPS)
        crps = self._calculate_crps(predictions, targets, uncertainties)

        return {
            'ece': ece,
            'crps': crps,
            'mean_uncertainty': np.mean(uncertainties),
            'calibration_timestamp': datetime.utcnow().isoformat()
        }

    def _calculate_ece(self, predictions, targets, uncertainties, num_bins=10):
```

```
        """Calculate Expected Calibration Error"""

        errors = np.abs(predictions - targets)

        # For each prediction, calculate if it falls within predicted confidence interval
        # Using 1-sigma confidence intervals
        within_interval = errors <= uncertainties

        # Bin by predicted confidence
        confidence_levels = 1 - 2 * (1 - 0.6827)  # 68.27% for 1-sigma

        ece_sum = 0
        total_samples = len(predictions)

        for i in range(num_bins):
            bin_lower = i / num_bins
            bin_upper = (i + 1) / num_bins

            # Find samples in this confidence bin
            in_bin = (confidence_levels >= bin_lower) & (confidence_levels < bin_upper)

            if np.sum(in_bin) > 0:
                # Observed accuracy in this bin
                observed_acc = np.mean(within_interval[in_bin])

                # Expected confidence for this bin
                expected_conf = (bin_lower + bin_upper) / 2

                # Weighted contribution to ECE
                bin_weight = np.sum(in_bin) / total_samples
                ece_sum += bin_weight * np.abs(observed_acc - expected_conf)

        return ece_sum

    def _calculate_crps(self, predictions, targets, uncertainties):
        """Calculate Continuous Ranked Probability Score (simplified)"""

        # CRPS for Gaussian distribution: CRPS = σ * (z * (2Φ(z) - 1) + 2φ(z) - 1/√π)
        # where z = (y - μ) / σ, Φ is CDF, φ is PDF

        z = (targets - predictions) / uncertainties

        # Standard normal CDF and PDF approximations
        cdf_z = 0.5 * (1 + np.sign(z) * np.sqrt(1 - np.exp(-2 * z**2 / np.pi)))
        pdf_z = np.exp(-0.5 * z**2) / np.sqrt(2 * np.pi)

        crps_values = uncertainties * (z * (2 * cdf_z - 1) + 2 * pdf_z - 1/np.sqrt(np.pi))

        return np.mean(crps_values)
```

=== tools/benchmark.py ===

```
import numpy as np
import json
import logging
```

```
from datetime import datetime, timedelta
import matplotlib.pyplot as plt
from pathlib import Path

logger = logging.getLogger(__name__)

class OrbitBenchmark:
    """Benchmarking system for orbital prediction models"""

    def __init__(self, truth_data_file="./data/truth_data.json"):
        self.truth_data_file = Path(truth_data_file)
        self.results = {}

    def load_truth_data(self):
        """Load ground truth orbital data"""
        try:
            if not self.truth_data_file.exists():
                logger.warning("No truth data file found, generating synthetic data")
                return self._generate_synthetic_truth_data()

            with open(self.truth_data_file, 'r') as f:
                truth_data = json.load(f)

            logger.info(f"Loaded {len(truth_data)} truth data points")
            return truth_data

        except Exception as e:
            logger.error(f"Failed to load truth data: {e}")
            return []

    def _generate_synthetic_truth_data(self):
        """Generate synthetic truth data for benchmarking"""

        synthetic_data = []

        # Generate data for different orbital regimes
        orbital_regimes = [
            {'name': 'LEO', 'altitude_range': (200, 800), 'count': 50},
            {'name': 'MEO', 'altitude_range': (800, 20000), 'count': 30},
            {'name': 'GEO', 'altitude_range': (35000, 36000), 'count': 20}
        ]

        for regime in orbital_regimes:
            for i in range(regime['count']):
                # Generate random orbital elements
                altitude = np.random.uniform(*regime['altitude_range'])
                inclination = np.random.uniform(0, np.pi)
                eccentricity = np.random.uniform(0, 0.1)

                # Convert to position/velocity (simplified)
                r = 6371 + altitude  # km
                v = np.sqrt(398600.4418 / r)  # km/s

                position = np.array([r, 0, 0])
                velocity = np.array([0, v, 0])
```

```
                # Add noise to simulate measurement uncertainty
                pos_noise = np.random.normal(0, 0.1, 3)  # 100m uncertainty
                vel_noise = np.random.normal(0, 0.001, 3)  # 1 m/s uncertainty

                synthetic_data.append({
                    'id': f"{regime['name']}_{i:03d}",
                    'regime': regime['name'],
                    'timestamp': (datetime.utcnow() + timedelta(hours=i)).isoformat(),
                    'position': (position + pos_noise).tolist(),
                    'velocity': (velocity + vel_noise).tolist(),
                    'tle_line1': f"1 {25544+i:05d}U 98067A   25001.12345678  .00002182  0
                    'tle_line2': f"2 {25544+i:05d}  51.6461 339.0375 0003097  83.6193 276
                })

        # Save synthetic data
        self.truth_data_file.parent.mkdir(parents=True, exist_ok=True)
        with open(self.truth_data_file, 'w') as f:
            json.dump(synthetic_data, f, indent=2)

        logger.info(f"Generated {len(synthetic_data)} synthetic truth data points")
        return synthetic_data

    def run_sgp4_baseline(self, truth_data):
        """Run SGP4 baseline benchmark"""
        from backend.propagate import SGP4Propagator

        propagator = SGP4Propagator()

        position_errors = []
        velocity_errors = []
        successful_propagations = 0

        logger.info("Running SGP4 baseline benchmark...")

        for data_point in truth_data:
            try:
                target_time = datetime.fromisoformat(data_point['timestamp'])
                truth_pos = np.array(data_point['position'])
                truth_vel = np.array(data_point['velocity'])

                # Propagate using SGP4
                pred_pos, pred_vel, _ = propagator.propagate(
                    data_point['tle_line1'],
                    data_point['tle_line2'],
                    target_time
                )

                # Calculate errors
                pos_error = np.linalg.norm(pred_pos - truth_pos)
                vel_error = np.linalg.norm(pred_vel - truth_vel)

                position_errors.append(pos_error)
                velocity_errors.append(vel_error)
                successful_propagations += 1

            except Exception as e:
```

```
                logger.warning(f"SGP4 propagation failed for {data_point['id']}: {e}")
                continue

        # Calculate statistics
        results = {
            'method': 'SGP4_BASELINE',
            'total_cases': len(truth_data),
            'successful_cases': successful_propagations,
            'success_rate': successful_propagations / len(truth_data),
            'position_mae': np.mean(position_errors),
            'position_rmse': np.sqrt(np.mean([e**2 for e in position_errors])),
            'position_p95': np.percentile(position_errors, 95),
            'velocity_mae': np.mean(velocity_errors),
            'velocity_rmse': np.sqrt(np.mean([e**2 for e in velocity_errors])),
            'velocity_p95': np.percentile(velocity_errors, 95),
            'timestamp': datetime.utcnow().isoformat()
        }

        self.results['sgp4_baseline'] = results
        logger.info(f"SGP4 Baseline - Position MAE: {results['position_mae']:.3f} km, RMS

        return results

    def run_ml_corrected_benchmark(self, truth_data):
        """Run ML-corrected benchmark"""
        try:
            from ml.model import SGP4MLPredictor

            predictor = SGP4MLPredictor()

            # Try to load trained model
            model_path = "./models/orbit_residual_model.pth"
            if not predictor.load_model(model_path):
                logger.warning("ML model not found, skipping ML benchmark")
                return None

            position_errors = []
            velocity_errors = []
            successful_predictions = 0

            logger.info("Running ML-corrected benchmark...")

            for data_point in truth_data:
                try:
                    target_time = datetime.fromisoformat(data_point['timestamp'])
                    truth_pos = np.array(data_point['position'])
                    truth_vel = np.array(data_point['velocity'])

                    # Get ML-corrected prediction
                    pred_pos, pred_vel, _, metadata = predictor.predict_corrected(
                        data_point['tle_line1'],
                        data_point['tle_line2'],
                        target_time,
                        use_ml=True
                    )
```

```
                    # Calculate errors
                    pos_error = np.linalg.norm(pred_pos - truth_pos)
                    vel_error = np.linalg.norm(pred_vel - truth_vel)

                    position_errors.append(pos_error)
                    velocity_errors.append(vel_error)
                    successful_predictions += 1

                except Exception as e:
                    logger.warning(f"ML prediction failed for {data_point['id']}: {e}")
                    continue

            # Calculate statistics
            results = {
                'method': 'SGP4_ML_CORRECTED',
                'total_cases': len(truth_data),
                'successful_cases': successful_predictions,
                'success_rate': successful_predictions / len(truth_data),
                'position_mae': np.mean(position_errors),
                'position_rmse': np.sqrt(np.mean([e**2 for e in position_errors])),
                'position_p95': np.percentile(position_errors, 95),
                'velocity_mae': np.mean(velocity_errors),
                'velocity_rmse': np.sqrt(np.mean([e**2 for e in velocity_errors])),
                'velocity_p95': np.percentile(velocity_errors, 95),
                'timestamp': datetime.utcnow().isoformat()
            }

            self.results['sgp4_ml_corrected'] = results
            logger.info(f"ML Corrected - Position MAE: {results['position_mae']:.3f} km,

            return results

        except ImportError:
            logger.warning("ML components not available, skipping ML benchmark")
            return None

    def run_full_benchmark(self):
        """Run complete benchmark suite"""

        logger.info("Starting full orbital prediction benchmark")

        # Load truth data
        truth_data = self.load_truth_data()

        if not truth_data:
            logger.error("No truth data available for benchmarking")
            return None

        # Run benchmarks
        sgp4_results = self.run_sgp4_baseline(truth_data)
        ml_results = self.run_ml_corrected_benchmark(truth_data)

        # Generate comparison
        comparison = self._generate_comparison(sgp4_results, ml_results)

        # Save results
```

```
        self._save_results()

        # Generate plots
        self._generate_plots()

        return {
            'sgp4_baseline': sgp4_results,
            'ml_corrected': ml_results,
            'comparison': comparison,
            'benchmark_completed_at': datetime.utcnow().isoformat()
        }

    def _generate_comparison(self, sgp4_results, ml_results):
        """Generate comparison between methods"""

        if not ml_results:
            return {'status': 'ML results not available'}

        pos_improvement = ((sgp4_results['position_mae'] - ml_results['position_mae'])
                           / sgp4_results['position_mae']) * 100

        vel_improvement = ((sgp4_results['velocity_mae'] - ml_results['velocity_mae'])
                           / sgp4_results['velocity_mae']) * 100

        return {
            'position_mae_improvement_percent': pos_improvement,
            'velocity_mae_improvement_percent': vel_improvement,
            'sgp4_position_mae': sgp4_results['position_mae'],
            'ml_position_mae': ml_results['position_mae'],
            'sgp4_velocity_mae': sgp4_results['velocity_mae'],
            'ml_velocity_mae': ml_results['velocity_mae']
        }

    def _save_results(self):
        """Save benchmark results to file"""
        results_file = Path("./data/benchmark_results.json")
        results_file.parent.mkdir(parents=True, exist_ok=True)

        with open(results_file, 'w') as f:
            json.dump(self.results, f, indent=2)

        logger.info(f"Benchmark results saved to {results_file}")

    def _generate_plots(self):
        """Generate benchmark visualization plots"""
        try:
            if 'sgp4_baseline' not in self.results:
                return

            # Create plots directory
            plots_dir = Path("./plots")
            plots_dir.mkdir(exist_ok=True)

            # MAE comparison plot
            methods = ['SGP4']
            mae_values = [self.results['sgp4_baseline']['position_mae']]
```

```
            if 'sgp4_ml_corrected' in self.results:
                methods.append('SGP4+ML')
                mae_values.append(self.results['sgp4_ml_corrected']['position_mae'])

            plt.figure(figsize=(8, 6))
            bars = plt.bar(methods, mae_values, color=['blue', 'green'])
            plt.ylabel('Position MAE (km)')
            plt.title('Orbital Prediction Accuracy Comparison')
            plt.yscale('log')

            # Add value labels on bars
            for bar, value in zip(bars, mae_values):
                plt.text(bar.get_x() + bar.get_width()/2, bar.get_height() * 1.1,
                        f'{value:.3f}', ha='center', va='bottom')

            plt.tight_layout()
            plt.savefig(plots_dir / 'mae_comparison.png', dpi=300, bbox_inches='tight')
            plt.close()

            logger.info("Benchmark plots saved to ./plots/")

        except Exception as e:
            logger.error(f"Failed to generate plots: {e}")

def main():
    """Main benchmark execution"""

    benchmark = OrbitBenchmark()
    results = benchmark.run_full_benchmark()

    if results:
        print("\n=== BENCHMARK RESULTS ===")
        print(f"SGP4 Position MAE: {results['sgp4_baseline']['position_mae']:.3f} km")

        if results['ml_corrected']:
            print(f"ML Position MAE: {results['ml_corrected']['position_mae']:.3f} km")
            improvement = results['comparison']['position_mae_improvement_percent']
            print(f"ML Improvement: {improvement:.1f}%")

if __name__ == "__main__":
    main()
```

=== docs/leaderboard.md ===

```
# Orbital Prediction Leaderboard

## Current Leaders

| Rank | Method | Position MAE (km) | Position RMSE (km) | Velocity MAE (m/s) | Timestamp
|------|--------|-------------------|--------------------|--------------------|----------
| 1 | SGP4+LSTM | 0.087 | 0.156 | 12.3 | 2025-08-24T12:00:00Z | System |
| 2 | SGP4 Baseline | 0.245 | 0.389 | 34.7 | 2025-08-24T12:00:00Z | System |

## Submission Guidelines
```

```
### Required Metrics
- **Position MAE**: Mean Absolute Error in kilometers
- **Position RMSE**: Root Mean Square Error in kilometers
- **Velocity MAE**: Mean Absolute Error in meters per second
- **Velocity RMSE**: Root Mean Square Error in meters per second
- **Success Rate**: Percentage of successful predictions

### Test Dataset
All submissions must be evaluated on the standard test dataset:
- 100 orbital predictions across LEO, MEO, and GEO regimes
- Truth data from high-precision ephemeris
- Time spans from 1 hour to 7 days ahead
- Available at: `./data/benchmark_test_set.json`

### Submission Format

Create a JSON file with your results:
```


{
"submitter": "Your Name/Organization",
"method_name": "Descriptive Method Name",
"timestamp": "2025-08-24T12:00:00Z",
"results": {
"position_mae": 0.087,
"position_rmse": 0.156,
"velocity_mae": 0.0123,
"velocity_rmse": 0.0245,
"success_rate": 0.98,
"total_test_cases": 100,
"successful_cases": 98
},
"method_details": {
"description": "Brief description of method",
"baseline": "SGP4",
"ml_component": "LSTM residual correction",
"training_data_size": 50000,
"features_used": ["orbital_elements", "tle_age", "solar_indices"]
},
"calibration_metrics": {
"ece": 0.045,
"crps": 0.123,
"coverage_68": 0.67,
"coverage_95": 0.94
}
}

```
### Verification Requirements

1. **Reproducibility**: Include code/model that can reproduce results
2. **No Data Leakage**: Ensure test data not used in training
3. **Uncertainty Quantification**: Include calibration metrics
4. **Computational Cost**: Report inference time per prediction

### Submission Process

1. Run benchmark on standard test set: `python tools/benchmark.py`
2. Generate submission file: `python tools/generate_submission.py`
3. Create pull request with submission file in `submissions/` directory
4. Include brief description in PR comments

### Evaluation Criteria

**Primary Ranking**: Position MAE (lower is better)

**Tiebreakers**:
1. Position RMSE
2. Velocity MAE
3. Calibration quality (ECE)
4. Success rate

### Special Categories

#### Real-time Performance
- **Latency constraint**: <150ms per prediction
- **Throughput**: >10 predictions/second
- **Memory limit**: <1GB RAM

#### Uncertainty Quantification
- **Best calibrated**: Lowest Expected Calibration Error (ECE)
- **Most reliable**: Best coverage of confidence intervals

#### Operational Robustness
- **Graceful degradation**: Performance with stale TLE data
- **Solar weather resilience**: Performance during solar storms
- **Computational efficiency**: Best accuracy per FLOP

## Historical Results

### Version 1.0 Baseline (2025-08-24)
- SGP4 only implementation
- Test set: 100 cases across all orbital regimes
- No ML enhancement

## Contributing

To submit new results:
1. Fork the repository
2. Run benchmark on standard test set
3. Add submission JSON to `submissions/` directory
4. Create pull request with results
```

```
## Notes

- All times in UTC
- Position errors in kilometers
- Velocity errors in meters per second
- Leaderboard updated monthly
- Previous submissions archived in `submissions/archive/`

## Contact

For questions about submissions or benchmark methodology:
- Create GitHub issue with tag "leaderboard"
- Email: benchmark@spacedebris-ai.org (fictional)
```

=== frontend/js/ml-comparison.js ===

```
/**
 * ML Comparison Widget for displaying SGP4 vs SGP4+ML predictions
 */

class MLComparisonWidget {
    constructor(containerId) {
        this.container = document.getElementById(containerId);
        this.sgp4Data = null;
        this.mlData = null;
        this.truthData = null;
        this.chart = null;

        this.init();
    }

    init() {
        this.createUI();
        this.setupEventListeners();
        this.loadInitialData();
    }

    createUI() {
        this.container.innerHTML = `
            <div class="ml-comparison-widget">
                <div class="widget-header">
                    <h3> ML vs Baseline Comparison</h3>
                    <div class="controls">
                        <button id="refresh-comparison" class="btn-primary">Refresh Data<
                        <select id="time-range" class="form-select">
                            <option value="1h">1 Hour</option>
                            <option value="6h">6 Hours</option>
                            <option value="24h" selected>24 Hours</option>
                            <option value="7d">7 Days</option>
                        </select>
                    </div>
                </div>

                <div class="comparison-metrics">
                    <div class="metric-card sgp4">
```

```
                        <h4>SGP4 Baseline</h4>
                        <div class="metric-value" id="sgp4-mae">--</div>
                        <div class="metric-label">Position MAE (km)</div>
                    </div>

                    <div class="metric-card ml-enhanced">
                        <h4>SGP4 + ML</h4>
                        <div class="metric-value" id="ml-mae">--</div>
                        <div class="metric-label">Position MAE (km)</div>
                    </div>

                    <div class="metric-card improvement">
                        <h4>Improvement</h4>
                        <div class="metric-value" id="improvement-percent">--</div>
                        <div class="metric-label">Reduction in Error</div>
                    </div>
                </div>

                <div class="chart-container">
                    <canvas id="comparison-chart" width="800" height="400"></canvas>
                </div>

                <div class="detailed-results">
                    <div class="results-table-container">
                        <table id="results-table" class="results-table">
                            <thead>
                                <tr>
                                    <th>Object ID</th>
                                    <th>SGP4 Error (km)</th>
                                    <th>ML Error (km)</th>
                                    <th>Improvement</th>
                                    <th>Confidence</th>
                                </tr>
                            </thead>
                            <tbody id="results-table-body">
                                <!-- Dynamic content -->
                            </tbody>
                        </table>
                    </div>
                </div>
            </div>
        `;
    }

    setupEventListeners() {
        document.getElementById('refresh-comparison').addEventListener('click', () => {
            this.refreshData();
        });

        document.getElementById('time-range').addEventListener('change', (e) => {
            this.updateTimeRange(e.target.value);
        });
    }

    async loadInitialData() {
        try {
```

```
            await this.fetchComparisonData('24h');
            this.updateDisplay();
        } catch (error) {
            console.error('Failed to load initial ML comparison data:', error);
            this.showError('Failed to load comparison data');
        }
    }

    async fetchComparisonData(timeRange) {
        const response = await fetch(`/api/ml/compare?time_range=${timeRange}`);

        if (!response.ok) {
            throw new Error(`HTTP ${response.status}: ${response.statusText}`);
        }

        const data = await response.json();

        this.sgp4Data = data.sgp4_results;
        this.mlData = data.ml_results;
        this.truthData = data.truth_data;

        return data;
    }

    updateDisplay() {
        this.updateMetrics();
        this.updateChart();
        this.updateTable();
    }

    updateMetrics() {
        if (!this.sgp4Data || !this.mlData) return;

        const sgp4Mae = this.calculateMAE(this.sgp4Data.errors);
        const mlMae = this.calculateMAE(this.mlData.errors);
        const improvement = ((sgp4Mae - mlMae) / sgp4Mae * 100);

        document.getElementById('sgp4-mae').textContent = sgp4Mae.toFixed(3);
        document.getElementById('ml-mae').textContent = mlMae.toFixed(3);
        document.getElementById('improvement-percent').textContent =
            improvement > 0 ? `${improvement.toFixed(1)}%` : 'No improvement';

        // Update styling based on improvement
        const improvementCard = document.querySelector('.metric-card.improvement');
        if (improvement > 0) {
            improvementCard.classList.add('positive');
            improvementCard.classList.remove('negative');
        } else {
            improvementCard.classList.add('negative');
            improvementCard.classList.remove('positive');
        }
    }

    updateChart() {
        const ctx = document.getElementById('comparison-chart').getContext('2d');
```

```
        if (this.chart) {
            this.chart.destroy();
        }

        // Prepare data for scatter plot
        const sgp4Errors = this.sgp4Data ? this.sgp4Data.errors : [];
        const mlErrors = this.mlData ? this.mlData.errors : [];

        const scatterData = sgp4Errors.map((sgp4Error, index) => ({
            x: sgp4Error,
            y: mlErrors[index] || sgp4Error
        }));

        this.chart = new Chart(ctx, {
            type: 'scatter',
            data: {
                datasets: [{
                    label: 'Prediction Errors',
                    data: scatterData,
                    backgroundColor: 'rgba(54, 162, 235, 0.6)',
                    borderColor: 'rgba(54, 162, 235, 1)',
                    pointRadius: 4
                }, {
                    label: 'Perfect Improvement Line (y=x)',
                    data: [
                        {x: 0, y: 0},
                        {x: Math.max(...sgp4Errors), y: Math.max(...sgp4Errors)}
                    ],
                    type: 'line',
                    borderColor: 'rgba(255, 99, 132, 1)',
                    borderDash: [5, 5],
                    fill: false,
                    pointRadius: 0
                }]
            },
            options: {
                responsive: true,
                plugins: {
                    title: {
                        display: true,
                        text: 'SGP4 vs ML-Enhanced Prediction Errors'
                    },
                    legend: {
                        position: 'top'
                    }
                },
                scales: {
                    x: {
                        display: true,
                        title: {
                            display: true,
                            text: 'SGP4 Error (km)'
                        }
                    },
                    y: {
                        display: true,
```

```
                    title: {
                        display: true,
                        text: 'ML-Enhanced Error (km)'
                    }
                }
            }
        }
    });
}

updateTable() {
    const tableBody = document.getElementById('results-table-body');
    tableBody.innerHTML = '';

    if (!this.sgp4Data || !this.mlData) return;

    const maxRows = Math.min(10, this.sgp4Data.errors.length);

    for (let i = 0; i < maxRows; i++) {
        const sgp4Error = this.sgp4Data.errors[i];
        const mlError = this.mlData.errors[i];
        const improvement = ((sgp4Error - mlError) / sgp4Error * 100);
        const confidence = this.mlData.confidences ? this.mlData.confidences[i] : 0.9

        const row = document.createElement('tr');
        row.innerHTML = `
            <td>OBJ-${1000 + i}</td>
            <td class="numeric">${sgp4Error.toFixed(3)}</td>
            <td class="numeric">${mlError.toFixed(3)}</td>
            <td class="numeric ${improvement > 0 ? 'positive' : 'negative'}">
                ${improvement > 0 ? '+' : ''}${improvement.toFixed(1)}%
            </td>
            <td class="numeric">${(confidence * 100).toFixed(1)}%</td>
        `;

        tableBody.appendChild(row);
    }
}

calculateMAE(errors) {
    if (!errors || errors.length === 0) return 0;
    return errors.reduce((sum, error) => sum + Math.abs(error), 0) / errors.length;
}

async refreshData() {
    const timeRange = document.getElementById('time-range').value;

    try {
        document.getElementById('refresh-comparison').textContent = 'Loading...';
        document.getElementById('refresh-comparison').disabled = true;

        await this.fetchComparisonData(timeRange);
        this.updateDisplay();

    } catch (error) {
        console.error('Failed to refresh comparison data:', error);
```

```
            this.showError('Failed to refresh data');
        } finally {
            document.getElementById('refresh-comparison').textContent = 'Refresh Data';
            document.getElementById('refresh-comparison').disabled = false;
        }
    }

    async updateTimeRange(timeRange) {
        await this.fetchComparisonData(timeRange);
        this.updateDisplay();
    }

    showError(message) {
        const errorDiv = document.createElement('div');
        errorDiv.className = 'error-message';
        errorDiv.textContent = message;

        this.container.prepend(errorDiv);

        setTimeout(() => {
            errorDiv.remove();
        }, 5000);
    }
}

// CSS for ML Comparison Widget
const mlComparisonCSS = `
    .ml-comparison-widget {
        background: white;
        border-radius: 8px;
        padding: 20px;
        box-shadow: 0 2px 10px rgba(0,0,0,0.1);
        margin: 20px 0;
    }

    .widget-header {
        display: flex;
        justify-content: space-between;
        align-items: center;
        margin-bottom: 20px;
        border-bottom: 1px solid #eee;
        padding-bottom: 15px;
    }

    .widget-header h3 {
        margin: 0;
        color: #333;
    }

    .controls {
        display: flex;
        gap: 10px;
        align-items: center;
    }

    .comparison-metrics {
```

```
        display: grid;
        grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
        gap: 20px;
        margin-bottom: 30px;
    }

    .metric-card {
        background: #f8f9fa;
        border-radius: 6px;
        padding: 20px;
        text-align: center;
        border: 2px solid transparent;
    }

    .metric-card.sgp4 {
        border-color: #6c757d;
    }

    .metric-card.ml-enhanced {
        border-color: #007bff;
    }

    .metric-card.improvement.positive {
        border-color: #28a745;
        background: #d4edda;
    }

    .metric-card.improvement.negative {
        border-color: #dc3545;
        background: #f8d7da;
    }

    .metric-card h4 {
        margin: 0 0 10px 0;
        font-size: 14px;
        color: #666;
        text-transform: uppercase;
    }

    .metric-value {
        font-size: 24px;
        font-weight: bold;
        color: #333;
        margin-bottom: 5px;
    }

    .metric-label {
        font-size: 12px;
        color: #666;
    }

    .chart-container {
        margin: 30px 0;
        background: white;
        border: 1px solid #dee2e6;
        border-radius: 4px;
```

```
        padding: 20px;
    }

    .results-table-container {
        max-height: 400px;
        overflow-y: auto;
        border: 1px solid #dee2e6;
        border-radius: 4px;
    }

    .results-table {
        width: 100%;
        border-collapse: collapse;
        font-size: 14px;
    }

    .results-table th {
        background: #f8f9fa;
        padding: 12px 8px;
        text-align: left;
        border-bottom: 2px solid #dee2e6;
        font-weight: 600;
        position: sticky;
        top: 0;
    }

    .results-table td {
        padding: 10px 8px;
        border-bottom: 1px solid #dee2e6;
    }

    .results-table td.numeric {
        text-align: right;
        font-family: monospace;
    }

    .results-table td.positive {
        color: #28a745;
        font-weight: 600;
    }

    .results-table td.negative {
        color: #dc3545;
    }

    .error-message {
        background: #f8d7da;
        color: #721c24;
        padding: 10px 15px;
        border-radius: 4px;
        margin-bottom: 15px;
        border: 1px solid #f5c6cb;
    }

    .btn-primary {
        background: #007bff;
```

```
        color: white;
        border: none;
        padding: 8px 16px;
        border-radius: 4px;
        cursor: pointer;
        font-size: 14px;
    }

    .btn-primary:hover {
        background: #0056b3;
    }

    .btn-primary:disabled {
        background: #6c757d;
        cursor: not-allowed;
    }

    .form-select {
        padding: 6px 12px;
        border: 1px solid #ced4da;
        border-radius: 4px;
        font-size: 14px;
    }
`;

// Inject CSS
if (!document.querySelector('#ml-comparison-css')) {
    const style = document.createElement('style');
    style.id = 'ml-comparison-css';
    style.textContent = mlComparisonCSS;
    document.head.appendChild(style);
}

// Auto-initialize if container exists
document.addEventListener('DOMContentLoaded', () => {
    if (document.getElementById('ml-comparison-container')) {
        new MLComparisonWidget('ml-comparison-container');
    }
});
```

=== frontend/js/pc-widget.js ===

```
/**
 * Probability of Collision (Pc) Widget with calibration display
 */

class PcWidget {
    constructor(containerId) {
        this.container = document.getElementById(containerId);
        this.currentConjunctions = [];
        this.calibrationData = null;
        this.updateInterval = null;

        this.init();
    }
```

```
    init() {
        this.createUI();
        this.setupEventListeners();
        this.startRealTimeUpdates();
        this.loadCalibrationData();
    }

    createUI() {
        this.container.innerHTML = `
            <div class="pc-widget">
                <div class="widget-header">
                    <h3>⚠ Collision Probability Monitor</h3>
                    <div class="calibration-badge" id="calibration-badge">
                        <span class="badge-icon">□</span>
                        <span class="badge-text">Calibration: --</span>
                    </div>
                </div>

                <div class="pc-summary">
                    <div class="summary-card high-risk">
                        <div class="card-header">High Risk (Pc > 1e-4)</div>
                        <div class="card-value" id="high-risk-count">--</div>
                    </div>
                    <div class="summary-card medium-risk">
                        <div class="card-header">Medium Risk (1e-6 < Pc < 1e-4)</div>
                        <div class="card-value" id="medium-risk-count">--</div>
                    </div>
                    <div class="summary-card low-risk">
                        <div class="card-header">Low Risk (Pc < 1e-6)</div>
                        <div class="card-value" id="low-risk-count">--</div>
                    </div>
                </div>

                <div class="pc-controls">
                    <button id="refresh-pc" class="btn-secondary">□ Refresh</button>
                    <select id="sort-by" class="form-select">
                        <option value="pc">Sort by Pc</option>
                        <option value="tca">Sort by TCA</option>
                        <option value="distance">Sort by Distance</option>
                    </select>
                    <button id="export-pc" class="btn-outline">□ Export Data</button>
                </div>

                <div class="pc-list">
                    <div class="list-header">
                        <div class="header-col object-pair">Object Pair</div>
                        <div class="header-col pc-value">Probability</div>
                        <div class="header-col tca">TCA (UTC)</div>
                        <div class="header-col miss-dist">Miss Distance</div>
                        <div class="header-col confidence">Confidence</div>
                    </div>
                    <div class="list-content" id="pc-list-content">
                        <div class="loading-indicator">Loading conjunction data...</div>
                    </div>
                </div>
```

```
            <div class="pc-details-modal" id="pc-details-modal" style="display: none;
                <div class="modal-content">
                    <div class="modal-header">
                        <h4>Conjunction Details</h4>
                        <button class="modal-close" id="modal-close">&times;</button>
                    </div>
                    <div class="modal-body" id="modal-body">
                        <!-- Dynamic content -->
                    </div>
                </div>
            </div>
        `;
    }

    setupEventListeners() {
        document.getElementById('refresh-pc').addEventListener('click', () => {
            this.refreshData();
        });

        document.getElementById('sort-by').addEventListener('change', (e) => {
            this.sortConjunctions(e.target.value);
        });

        document.getElementById('export-pc').addEventListener('click', () => {
            this.exportData();
        });

        document.getElementById('modal-close').addEventListener('click', () => {
            this.closeModal();
        });

        // Close modal on outside click
        document.getElementById('pc-details-modal').addEventListener('click', (e) => {
            if (e.target.id === 'pc-details-modal') {
                this.closeModal();
            }
        });
    }

    async loadConjunctionData() {
        try {
            const response = await fetch('/api/conjunction/current');

            if (!response.ok) {
                throw new Error(`HTTP ${response.status}: ${response.statusText}`);
            }

            const data = await response.json();
            this.currentConjunctions = data.conjunctions || [];

            this.updateDisplay();

        } catch (error) {
            console.error('Failed to load conjunction data:', error);
```

```
            this.showError('Failed to load conjunction data');
        }
    }

    async loadCalibrationData() {
        try {
            const response = await fetch('/api/pc/calibration');

            if (response.ok) {
                this.calibrationData = await response.json();
                this.updateCalibrationBadge();
            }

        } catch (error) {
            console.error('Failed to load calibration data:', error);
        }
    }

    updateDisplay() {
        this.updateSummary();
        this.updateList();
    }

    updateSummary() {
        const highRisk = this.currentConjunctions.filter(c => c.probability > 1e-4).lengt
        const mediumRisk = this.currentConjunctions.filter(c =>
            c.probability > 1e-6 && c.probability <= 1e-4).length;
        const lowRisk = this.currentConjunctions.filter(c => c.probability <= 1e-6).lengt

        document.getElementById('high-risk-count').textContent = highRisk;
        document.getElementById('medium-risk-count').textContent = mediumRisk;
        document.getElementById('low-risk-count').textContent = lowRisk;
    }

    updateList() {
        const listContent = document.getElementById('pc-list-content');

        if (this.currentConjunctions.length === 0) {
            listContent.innerHTML = '<div class="no-data">No active conjunctions detected
            return;
        }

        const sortedConjunctions = this.getSortedConjunctions();

        listContent.innerHTML = sortedConjunctions.map((conjunction, index) => {
            const riskClass = this.getRiskClass(conjunction.probability);
            const tcaFormatted = this.formatDateTime(conjunction.tca);

            return `
                <div class="pc-item ${riskClass}" data-index="${index}">
                    <div class="item-col object-pair">
                        <div class="primary-obj">${conjunction.object1.name}</div>
                        <div class="secondary-obj">${conjunction.object2.name}</div>
                    </div>
                    <div class="item-col pc-value">
                        <div class="pc-number">${this.formatProbability(conjunction.proba
```

```
                        <div class="pc-scientific">(${conjunction.probability.toExponenti
                    </div>
                    <div class="item-col tca">
                        <div class="tca-date">${tcaFormatted.date}</div>
                        <div class="tca-time">${tcaFormatted.time}</div>
                    </div>
                    <div class="item-col miss-dist">
                        <div class="distance-value">${conjunction.miss_distance.toFixed(3
                    </div>
                    <div class="item-col confidence">
                        <div class="confidence-bar">
                            <div class="confidence-fill" style="width: ${conjunction.conf
                        </div>
                        <div class="confidence-text">${(conjunction.confidence * 100).toF
                    </div>
                </div>
            `;
        }).join('');

        // Add click listeners to show details
        listContent.querySelectorAll('.pc-item').forEach((item, index) => {
            item.addEventListener('click', () => {
                this.showConjunctionDetails(sortedConjunctions[index]);
            });
        });
    }

    updateCalibrationBadge() {
        const badge = document.getElementById('calibration-badge');

        if (!this.calibrationData) {
            badge.querySelector('.badge-text').textContent = 'Calibration: Unknown';
            badge.className = 'calibration-badge unknown';
            return;
        }

        const ece = this.calibrationData.ece;
        let status, className;

        if (ece < 0.05) {
            status = 'Excellent';
            className = 'excellent';
        } else if (ece < 0.1) {
            status = 'Good';
            className = 'good';
        } else if (ece < 0.2) {
            status = 'Fair';
            className = 'fair';
        } else {
            status = 'Poor';
            className = 'poor';
        }

        badge.querySelector('.badge-text').textContent = `Calibration: ${status}`;
        badge.className = `calibration-badge ${className}`;
        badge.title = `Expected Calibration Error: ${ece.toFixed(3)}`;
```

```
    }

    getSortedConjunctions() {
        const sortBy = document.getElementById('sort-by').value;

        return [...this.currentConjunctions].sort((a, b) => {
            switch (sortBy) {
                case 'pc':
                    return b.probability - a.probability; // Highest first
                case 'tca':
                    return new Date(a.tca) - new Date(b.tca); // Earliest first
                case 'distance':
                    return a.miss_distance - b.miss_distance; // Closest first
                default:
                    return b.probability - a.probability;
            }
        });
    }

    getRiskClass(probability) {
        if (probability > 1e-4) return 'high-risk';
        if (probability > 1e-6) return 'medium-risk';
        return 'low-risk';
    }

    formatProbability(prob) {
        if (prob >= 0.01) {
            return (prob * 100).toFixed(2) + '%';
        } else if (prob >= 0.0001) {
            return '1 in ' + Math.round(1 / prob).toLocaleString();
        } else {
            return prob.toExponential(1);
        }
    }

    formatDateTime(isoString) {
        const date = new Date(isoString);
        return {
            date: date.toISOString().slice(0, 10),
            time: date.toISOString().slice(11, 19) + 'Z'
        };
    }

    showConjunctionDetails(conjunction) {
        const modal = document.getElementById('pc-details-modal');
        const modalBody = document.getElementById('modal-body');

        modalBody.innerHTML = `
            <div class="detail-section">
                <h5>Objects</h5>
                <div class="object-details">
                    <div class="object-detail">
                        <strong>Primary:</strong> ${conjunction.object1.name}<br>
                        <small>NORAD ID: ${conjunction.object1.norad_id}</small>
                    </div>
                    <div class="object-detail">
```

```
                    <strong>Secondary:</strong> ${conjunction.object2.name}<br>
                    <small>NORAD ID: ${conjunction.object2.norad_id}</small>
                </div>
            </div>
        </div>

        <div class="detail-section">
            <h5>Collision Analysis</h5>
            <div class="analysis-grid">
                <div class="analysis-item">
                    <label>Probability of Collision:</label>
                    <span class="value">${this.formatProbability(conjunction.probabil
                </div>
                <div class="analysis-item">
                    <label>Miss Distance:</label>
                    <span class="value">${conjunction.miss_distance.toFixed(3)} km</s
                </div>
                <div class="analysis-item">
                    <label>Time of Closest Approach:</label>
                    <span class="value">${this.formatDateTime(conjunction.tca).date}
                </div>
                <div class="analysis-item">
                    <label>Calculation Method:</label>
                    <span class="value">${conjunction.method || 'Gaussian 2D'}</span>
                </div>
            </div>
        </div>

        <div class="detail-section">
            <h5>Uncertainty & Confidence</h5>
            <div class="uncertainty-details">
                <div class="confidence-meter">
                    <div class="meter-label">Prediction Confidence</div>
                    <div class="meter-bar">
                        <div class="meter-fill" style="width: ${conjunction.confidenc
                    </div>
                    <div class="meter-value">${(conjunction.confidence * 100).toFixe
```