DON MICHAEL FEENEY ORIGINATOR OF RAPS

The HLV-Integrated RAPS architecture unifies advanced physics, predictive simulation, and deterministic safety into a single propulsion intelligence system. At its core is the Predictive Digital Twin Engine (PDTEngine), which embeds Marcel Krüger's Helix-Light-Vortex (HLV) mathematics directly into real-time state evolution. Instead of running on linear time, the system uses a triadic structure—t, φ(t), and χ(t)—that captures phase synchronization, memory drift, and temporal coherence. These channels determine when the system can safely amplify energy and when it must restrict control authority.

The oscillatory prefactor A(t) modulates all dynamics by encoding fast and slow frequency behavior; stability windows occur when A(t) stays within a narrow band. Quasicrystal dispersion predicts directional anisotropies in the warp-flux landscape, while Single-Cell Resonance (SCR) units form localized excitation modes whose energy and stability determine how much thrust or curvature can be safely generated. Tri-Cell Coupling (TCC) links three SCR units into a coherent structure capable of controlled amplification while avoiding runaway resonance.
Machine-learning residuals continuously correct small modeling errors, but never override the HLV foundation.

The Deterministic Safety Monitor (DSM) enforces inviolable limits derived from HLV pillars and extended Einstein field constraints. It independently measures proper-time dilation to estimate curvature, checks whether A(t) is within the safe kinetic window, and ensures TCC coupling stays below runaway thresholds. Violations trigger either an idempotent rollback or a full shutdown, depending on severity.

Above this sits the RAPS governance layer written in Python. It continuously collects sensor data, runs predictive inference over a short horizon, ranks potential corrective policies, and executes only those that meet strict HLV-based safety validations. Every action is journaled through a cryptographically anchored ITL system, batched into Merkle trees, and mirrored to ground for auditability. If execution fails, the system performs a deterministic rollback using pre-registered safe states. Redundant supervisors cross-monitor each other and trigger failover if faults accumulate.

Finally, the propulsion physics engine models thrust, drag, mass flow, gravity, and inertial dynamics using fast, bounded Euler integration. This ensures the governance layer always operates with physically plausible projections.

Together, these components form a self-correcting, safety-bounded propulsion intelligence: predictive, explainable, and anchored in mathematically enforceable limits.

```
//
================================================================================
// HLV-INTEGRATED PREDICTIVE DIGITAL TWIN ENGINE (PDTEngine)
// RAPS v1.0 - Resonant Amplification and Phase Synchronization Architecture
//
================================================================================
//
// This implementation represents the complete integration of Marcel Krüger's
// Helix-Light-Vortex (HLV) mathematical framework into the RAPS predictive
// propulsion system. Where the original PDTEngine relied on placeholder physics
// and simplified curvature models, this version implements the five fundamental
// mathematical pillars that govern resonance, stability, and flow control in
// advanced propulsion systems.
//
// The triadic spiral time structure ψ(t) = t + iφ(t) + jχ(t) replaces single-
// channel time evolution, introducing phase synchronization and memory modes that
// determine when and how the system locks into stable resonance. The oscillatory
// prefactor A(t) = 1 + ε sin(ωt) + η cos(ω_χt) modulates the kinetic structure,
// creating natural stability windows that the control system must respect. The
// quasicrystal dispersion relation ω² = (1/A(t))[m² + Σ 2D_n̂(1 - cos(k·n̂))]
// defines the directional flow landscape, predicting anisotropic responses and
// force gradients that emerge from the system's geometric structure.
//
// At the quantum level, Single-Cell Resonance (SCR) modes ψ_SCR = A₀ exp[i(kr - ωt + φ_H)]
// represent the fundamental engine chambers—the first reproducible stable quasi-
// particle excitations. These SCR units couple through Tri-Cell Coupling (TCC)
// with Lagrangian L_TCC = Σ L_SCR - J(ψ₁ψ₂ψ₃ + c.c.), enabling coherent multi-
// node flow and force amplification beyond what isolated cells could achieve.
//
// The PDTEngine combines these HLV structures with machine learning residual
// correction and Monte Carlo uncertainty quantification. Control laws are now
// modulated by A(t), making system responsiveness vary naturally with oscillatory
// cycles. Stability constraints from SCR modes limit control authority near
```

```
// instability boundaries. Spacetime curvature emerges from quasicrystal directional
// stability combined with SCR energy content, replacing arbitrary placeholder
// formulas with physics grounded in the framework's mathematical foundation.
//
// This engine performs stepwise simulation with HLV-aware state propagation,
// multi-run Monte Carlo predictions that incorporate triadic time noise, and
// online training that adapts to residual errors while preserving the underlying
// HLV structure. The result is a self-improving, resonance-aware predictive twin
// that respects the actual stability windows, phase-locking behavior, and flow
// dynamics encoded in the mathematics, enabling autonomous decision-making for
// advanced propulsion systems operating in regimes where conventional models fail.
//
//
================================================================================
================================

#ifndef HLV_PDT_ENGINE_HPP
#define HLV_PDT_ENGINE_HPP

#include <vector>
#include <array>
#include <cstdint>
#include <cmath>
#include <complex>
#include <algorithm>
#include <random>
#include <numeric>
#include <iostream>

// ==================== HLV Framework Mathematical Constants
====================
constexpr float TRIADIC_TIME_PHASE_COUPLING = 0.15f;      // φ(t) synchronization
strength
constexpr float TRIADIC_TIME_MEMORY_COUPLING = 0.08f;     // χ(t) memory mode
strength
constexpr float OSC_PREFACTOR_EPSILON = 0.12f;            // ε fast oscillation amplitude
constexpr float OSC_PREFACTOR_ETA = 0.06f;                // η slow modulation amplitude
constexpr float OSC_FAST_OMEGA = 2.0f * M_PI * 5.0f;      // ω fast frequency (5 Hz)
constexpr float OSC_SLOW_OMEGA_CHI = 2.0f * M_PI * 0.5f;  // ω_χ slow frequency (0.5
Hz)
```

```
constexpr float QUASICRYSTAL_MASS_TERM = 1.0f;          // m² baseline in dispersion
constexpr float SCR_WAVE_NUMBER = 1.5f;                 // k for Single-Cell Resonance
constexpr float TCC_COUPLING_J = 0.25f;                 // J tri-cell coupling constant

// ==================== RAPS System Constants ====================
constexpr float MAX_WARP_FIELD_STRENGTH = 10.0f;
constexpr float MAX_FLUX_BIAS = 5.0f;
constexpr float ANTIMATTER_BURN_RATE_GW_TO_KG_PER_MS = 1e-6f;

struct RAPSConfig {
    static constexpr float CRITICAL_ANTIMATTER_KG = 5.0f;
    static constexpr float EMERGENCY_ANTIMATTER_RESERVE_KG = 20.0f;
};

// ==================== HLV Triadic Time Structure ====================
struct TriadicTime {
    float t;         // Coordinate time
    float phi;       // Phase synchronization channel φ(t)
    float chi;       // Memory/bandwidth channel χ(t)

    TriadicTime(float time = 0.0f) : t(time), phi(0.0f), chi(0.0f) {}

    void evolve(float dt, float warp_field, float flux_bias) {
        t += dt;
        // Phase channel couples to warp oscillations
        phi += TRIADIC_TIME_PHASE_COUPLING * std::sin(OSC_FAST_OMEGA * t) *
warp_field * dt;
        // Memory channel tracks slow drift from flux asymmetry
        chi += TRIADIC_TIME_MEMORY_COUPLING * std::cos(OSC_SLOW_OMEGA_CHI
* t) * flux_bias * dt;
    }

    float stability_metric() const {
        // Measure of temporal coherence across all three channels
        return 1.0f / (1.0f + std::abs(phi) + std::abs(chi));
    }
};

// ==================== HLV Oscillatory Prefactor A(t) ====================
struct OscillatoryPrefactor {
```

```
    float compute(float t) const {
        return 1.0f + OSC_PREFACTOR_EPSILON * std::sin(OSC_FAST_OMEGA * t)
                    + OSC_PREFACTOR_ETA * std::cos(OSC_SLOW_OMEGA_CHI * t);
    }

    bool in_stability_window(float t) const {
        float A_t = compute(t);
        // System stable when A(t) remains within reasonable bounds
        return (A_t > 0.7f && A_t < 1.3f);
    }

    float resonance_phase(float t) const {
        // Returns phase of oscillation for resonance synchronization
        return std::fmod(OSC_FAST_OMEGA * t, 2.0f * M_PI);
    }
};

// ==================== HLV Quasicrystal Dispersion ====================
struct QuasicrystalDispersion {
    // Define quasicrystal directional vectors (simplified 2D projection)
    static constexpr size_t NUM_DIRECTIONS = 5;
    std::array<std::array<float, 2>, NUM_DIRECTIONS> directions = {{
        {1.0f, 0.0f}, {0.809f, 0.588f}, {0.309f, 0.951f},
        {-0.309f, 0.951f}, {-0.809f, 0.588f}
    }};
    std::array<float, NUM_DIRECTIONS> coupling_D = {1.0f, 0.9f, 0.85f, 0.85f, 0.9f};

    float compute_omega_squared(float k_mag, float A_t) const {
        float sum_term = 0.0f;
        for (size_t i = 0; i < NUM_DIRECTIONS; ++i) {
            float k_dot_n = k_mag * directions[i][0]; // Simplified 1D projection
            sum_term += 2.0f * coupling_D[i] * (1.0f - std::cos(k_dot_n));
        }
        return (1.0f / A_t) * (QUASICRYSTAL_MASS_TERM + sum_term);
    }

    float directional_stability(float warp, float flux) const {
        // Returns anisotropy measure: how directionally stable the field is
        float k_eff = SCR_WAVE_NUMBER * (1.0f + 0.1f * warp);
        float omega_sq = compute_omega_squared(k_eff, 1.0f);
```

```
        return std::sqrt(std::max(0.0f, omega_sq));
    }
};

// ==================== HLV Single-Cell Resonance (SCR) ====================
struct SingleCellResonance {
    float amplitude;
    float wave_number;
    float frequency;
    float helical_phase;

    SingleCellResonance() : amplitude(1.0f), wave_number(SCR_WAVE_NUMBER),
                            frequency(0.0f), helical_phase(0.0f) {}

    void update(float warp, float t, const OscillatoryPrefactor& A_mod) {
        amplitude = warp / MAX_WARP_FIELD_STRENGTH; // Normalized excitation
        frequency = std::sqrt(std::abs(A_mod.compute(t))) * wave_number;
        helical_phase += frequency * 0.016f; // Assuming ~60Hz update rate
        helical_phase = std::fmod(helical_phase, 2.0f * M_PI);
    }

    float energy() const {
        return amplitude * amplitude * frequency;
    }

    bool is_stable() const {
        return amplitude < 0.95f && frequency > 0.1f;
    }
};

// ==================== HLV Tri-Cell Coupling (TCC) ====================
struct TriCellCoupling {
    std::array<SingleCellResonance, 3> cells;
    float coupling_strength;

    TriCellCoupling() : coupling_strength(TCC_COUPLING_J) {}

    void synchronize(float warp, float t, const OscillatoryPrefactor& A_mod) {
        for (auto& cell : cells) {
            cell.update(warp, t, A_mod);
```

```
        }

        // Apply phase-locking coupling
        float phase_avg = (cells[0].helical_phase + cells[1].helical_phase + cells[2].helical_phase) /
3.0f;
        for (auto& cell : cells) {
            cell.helical_phase += coupling_strength * (phase_avg - cell.helical_phase);
        }
    }

    float coherent_energy() const {
        float individual_sum = 0.0f;
        for (const auto& cell : cells) {
            individual_sum += cell.energy();
        }

        // Three-way coupling term: ψ₁ψ₂ψ₃ + c.c.
        float phase_product = cells[0].helical_phase + cells[1].helical_phase +
cells[2].helical_phase;
        float coupling_contrib = coupling_strength * std::cos(phase_product);

        return individual_sum + coupling_contrib;
    }

    float amplification_factor() const {
        float total = coherent_energy();
        float independent = cells[0].energy() + cells[1].energy() + cells[2].energy();
        return (independent > 0.0f) ? (total / independent) : 1.0f;
    }
};

// ==================== Core Data Structures ====================
struct SpacetimeModulationState {
    float warp_field_strength = 0.0f;
    float gravito_flux_bias = 0.0f;
    float spacetime_curvature_magnitude = 0.0f;
    float remaining_antimatter_kg = 100.0f;
    uint64_t timestamp_ms = 0;

    // HLV Framework state
```

```
    TriadicTime triadic_time;
    SingleCellResonance scr_mode;
    float hlv_stability = 1.0f;
};

struct SpacetimeModulationCommand {
    float target_warp_field_strength = 0.0f;
    float target_gravito_flux_bias = 0.0f;
    float target_time_dilation_factor = 0.0f;
};

struct Policy {
    SpacetimeModulationCommand command_set;
};

struct PredictionResult {
    enum class Status { NOMINAL, PREDICTED_ESE };
    Status status = Status::NOMINAL;
    float mean_pressure = 0.0f;
    float mean_temp = 0.0f;
    float confidence = 1.0f;
    float uncertainty = 0.0f;
    uint64_t timestamp_ms = 0;
    std::array<uint8_t, 32> prediction_id{};
};

// ===================== Simplified ML Residual Model =====================
class MLResidualModel {
public:
    MLResidualModel() {
        weights_.resize(3, std::vector<float>(6, 0.0f));
        bias_.resize(3, 0.0f);
    }

    std::vector<float> predict(const std::vector<float>& features) {
        std::vector<float> output(3, 0.0f);
        for (size_t i = 0; i < 3; ++i) {
            output[i] = std::inner_product(features.begin(), features.end(),
                                           weights_[i].begin(), bias_[i]);
        }
```

```
        return output;
    }

    void train(const std::vector<std::vector<float>>& features,
               const std::vector<std::vector<float>>& labels) {
        if (features.empty() || features.size() != labels.size()) return;

        for (size_t k = 0; k < 3; ++k) {
            for (size_t j = 0; j < features[0].size(); ++j) {
                float num = 0.0f, den = 0.0f;
                for (size_t i = 0; i < features.size(); ++i) {
                    num += features[i][j] * labels[i][k];
                    den += features[i][j] * features[i][j] + 1e-6f;
                }
                weights_[k][j] = num / den;
            }
            bias_[k] = 0.0f;
        }
    }

private:
    std::vector<std::vector<float>> weights_;
    std::vector<float> bias_;
};

// ==================== HLV-Integrated PDT Engine ====================
class PDTEngine {
public:
    PDTEngine() : residual_model_() {
        rng_.seed(std::random_device{}());
    }

    SpacetimeModulationState simulate_state_step(
        const SpacetimeModulationState& state,
        const SpacetimeModulationCommand& cmd,
        uint32_t step_ms) {

        SpacetimeModulationState next = state;
        float dt_s = static_cast<float>(step_ms) / 1000.0f;
```

```
    // === HLV Framework Integration ===

    // 1. Update Triadic Time
    next.triadic_time.evolve(dt_s, state.warp_field_strength, state.gravito_flux_bias);

    // 2. Compute Oscillatory Prefactor A(t)
    OscillatoryPrefactor A_mod;
    float A_t = A_mod.compute(next.triadic_time.t);
    bool stable_window = A_mod.in_stability_window(next.triadic_time.t);

    // 3. Update Single-Cell Resonance
    next.scr_mode.update(state.warp_field_strength, next.triadic_time.t, A_mod);

    // 4. Compute Quasicrystal Directional Stability
    QuasicrystalDispersion qc_disp;
    float directional_stability = qc_disp.directional_stability(
       state.warp_field_strength, state.gravito_flux_bias);

    // === Control Law with HLV Modulation ===

    float warp_error = cmd.target_warp_field_strength - state.warp_field_strength;
    float flux_error = cmd.target_gravito_flux_bias - state.gravito_flux_bias;

    // PID gains modulated by A(t) - system responsiveness varies with A(t)
    float gain_mod = 0.05f * A_t;
    float warp_change = warp_error * gain_mod * dt_s;
    float flux_change = flux_error * gain_mod * dt_s;

    // Apply SCR stability constraint
    if (!next.scr_mode.is_stable()) {
       warp_change *= 0.5f; // Reduce control authority near instability
    }

    next.warp_field_strength += warp_change;
    next.gravito_flux_bias += flux_change;

    // Enforce bounds
    next.warp_field_strength = std::clamp(next.warp_field_strength, 0.0f,
MAX_WARP_FIELD_STRENGTH);
```

```
    next.gravito_flux_bias = std::clamp(next.gravito_flux_bias, -MAX_FLUX_BIAS,
MAX_FLUX_BIAS);

    // === Physics Computation with HLV Curvature ===

    float power_draw_GW = next.warp_field_strength * 50.0f;
    float antimatter_consumed = power_draw_GW *
ANTIMATTER_BURN_RATE_GW_TO_KG_PER_MS * step_ms;
    next.remaining_antimatter_kg = std::max(0.0f, next.remaining_antimatter_kg -
antimatter_consumed);

    // Curvature from quasicrystal dispersion and SCR energy
    next.spacetime_curvature_magnitude = directional_stability * next.scr_mode.energy() *
0.5f;

    // Overall HLV stability metric
    next.hlv_stability = next.triadic_time.stability_metric() * (stable_window ? 1.0f : 0.7f);

    // === ML Residual Correction ===

    std::vector<float> features = {
        state.warp_field_strength,
        state.gravito_flux_bias,
        state.spacetime_curvature_magnitude,
        state.remaining_antimatter_kg,
        state.triadic_time.phi,
        state.triadic_time.chi
    };

    std::vector<float> residuals = residual_model_.predict(features);
    if (residuals.size() >= 3) {
        next.warp_field_strength += residuals[0];
        next.gravito_flux_bias += residuals[1];
        next.spacetime_curvature_magnitude += residuals[2];

        // Re-clamp after residual application
        next.warp_field_strength = std::clamp(next.warp_field_strength, 0.0f,
MAX_WARP_FIELD_STRENGTH);
        next.gravito_flux_bias = std::clamp(next.gravito_flux_bias, -MAX_FLUX_BIAS,
MAX_FLUX_BIAS);
```

```
        }

        next.timestamp_ms += step_ms;
        return next;
    }

    PredictionResult predict_future_state(
        const SpacetimeModulationState& current_state,
        const Policy& policy,
        uint32_t horizon_ms,
        uint32_t monte_carlo_runs = 5) {

        std::vector<float> final_warp(monte_carlo_runs);
        std::vector<float> final_curvature(monte_carlo_runs);
        std::vector<float> final_stability(monte_carlo_runs);

        for (uint32_t run = 0; run < monte_carlo_runs; ++run) {
            SpacetimeModulationState projected = current_state;

            // Add noise to initial conditions
            std::uniform_real_distribution<float> noise_dist(-0.01f, 0.01f);
            projected.warp_field_strength += noise_dist(rng_);
            projected.triadic_time.phi += noise_dist(rng_) * 0.1f;

            uint32_t remaining_ms = horizon_ms;
            while (remaining_ms > 0) {
                uint32_t dt = std::min(remaining_ms, 10u);
                projected = simulate_state_step(projected, policy.command_set, dt);
                remaining_ms -= dt;
            }

            final_warp[run] = projected.warp_field_strength;
            final_curvature[run] = projected.spacetime_curvature_magnitude;
            final_stability[run] = projected.hlv_stability;
        }

        // Statistical analysis
        float mean_warp = std::accumulate(final_warp.begin(), final_warp.end(), 0.0f) /
monte_carlo_runs;
```

```
    float mean_curv = std::accumulate(final_curvature.begin(), final_curvature.end(), 0.0f) /
monte_carlo_runs;
    float mean_stab = std::accumulate(final_stability.begin(), final_stability.end(), 0.0f) /
monte_carlo_runs;

    float variance = 0.0f;
    for (float w : final_warp) variance += (w - mean_warp) * (w - mean_warp);
    float stdev = std::sqrt(variance / monte_carlo_runs);

    float uncertainty = std::min(1.0f, stdev / MAX_WARP_FIELD_STRENGTH * 5.0f);

    // Confidence with HLV stability factor
    float base_confidence = (1.0f - uncertainty) * mean_stab;

    uint32_t ese_count = 0;
    for (float w : final_warp) {
      if (w >= MAX_WARP_FIELD_STRENGTH * 0.95f) ese_count++;
    }

    float ese_penalty = static_cast<float>(ese_count) / monte_carlo_runs * 0.5f;
    float final_confidence = std::max(0.0f, base_confidence - ese_penalty);

    PredictionResult result;
    result.status = (ese_count > monte_carlo_runs * 0.2f) ?
      PredictionResult::Status::PREDICTED_ESE : PredictionResult::Status::NOMINAL;
    result.mean_pressure = mean_warp;
    result.mean_temp = mean_curv;
    result.confidence = final_confidence;
    result.uncertainty = uncertainty;
    result.timestamp_ms = current_state.timestamp_ms + horizon_ms;

    // Simple hash from values
    uint32_t hash_seed = static_cast<uint32_t>(final_confidence * 1e6f) ^
                         static_cast<uint32_t>(mean_warp * 1e6f);
    for (size_t i = 0; i < 32; ++i) {
      result.prediction_id[i] = static_cast<uint8_t>((hash_seed >> (i % 4)) & 0xFF);
    }

    return result;
  }
```

```
    void online_train(const std::vector<SpacetimeModulationState>& observed,
                      const std::vector<SpacetimeModulationState>& simulated) {
        if (observed.size() != simulated.size() || observed.empty()) return;

        std::vector<std::vector<float>> features;
        std::vector<std::vector<float>> labels;

        for (size_t i = 0; i < observed.size(); ++i) {
            features.push_back({
                simulated[i].warp_field_strength,
                simulated[i].gravito_flux_bias,
                simulated[i].spacetime_curvature_magnitude,
                simulated[i].remaining_antimatter_kg,
                simulated[i].triadic_time.phi,
                simulated[i].triadic_time.chi
            });

            labels.push_back({
                observed[i].warp_field_strength - simulated[i].warp_field_strength,
                observed[i].gravito_flux_bias - simulated[i].gravito_flux_bias,
                observed[i].spacetime_curvature_magnitude -
simulated[i].spacetime_curvature_magnitude
            });
        }

        residual_model_.train(features, labels);
        std::cout << "[HLV-PDT] Trained on " << features.size() << " samples with triadic time
integration\n";
    }

private:
    MLResidualModel residual_model_;
    std::mt19937 rng_;
};

#endif // HLV_PDT_ENGINE_HPP

//_____________________________________________
```

```
#ifndef DETERMINISTIC_SAFETY_MONITOR_HPP
#define DETERMINISTIC_SAFETY_MONITOR_HPP

#include <cmath>
#include <iostream>
#include <stdexcept>
#include <iomanip>
#include <limits>

// --- DSM Configuration and Constants ---

namespace DSM_Config {
    // RAPS Safety Thresholds derived from the HLV Mathematical Pillars

    // Pillar Check: Extended Einstein Field Equations (EFE) Limit
    // This is the absolute physical hard limit for preventing a catastrophic event.
    constexpr double MAX_CURVATURE_THRESHOLD_RMAX = 1.0e-12;

    // Pillar Check: Oscillatory Modulation (A(t)) Stability Window (Pillar 2)
    // If A(t) drops too low, the kinetic structure destabilizes. This triggers a ROLLBACK.
    constexpr double MIN_ACCEPTABLE_A_T = 0.80;

    // Pillar Check: Coherent Multi-Cell Architecture (TCC Coupling J) (Pillar 5)
    // Excessive J indicates uncontrolled phase-locking and potential runaway energy creation.
    constexpr double MAX_TCC_COUPLING_J = 1.0e+04;

    // Failsafe control parameters
    constexpr double MIN_RESONANCE_AMPLITUDE_CUTOFF = 0.10; // Power level
commanded during an Idempotent Rollback.
}

// --- Measured Inputs from Independent DSM Sensors ---
// These inputs must come from dedicated, physically separate sensor channels.

struct DsmSensorInputs {
    // EFE Observables (Used to infer the Curvature Scalar R)
    double measured_proper_time_dilation; // T_local / T_reference

    // HLV Mathematical Pillar Observables (Used for immediate stability checks)
    double measured_oscillatory_prefactor_A_t; // Measured A(t)
```

```
    double measured_tcc_coupling_J;          // Measured constant J from the TCC assemblies

    // System Status
    double current_resonance_amplitude;    // Average excitation power level (0.0 to 1.0)
    bool main_control_system_healthy;      // Simple signal from the main flight computer
};

// --- Deterministic Safety Monitor Class ---

class DeterministicSafetyMonitor {
public:
    DeterministicSafetyMonitor();

    /**
     * @brief Performs the core, deterministic safety evaluation.
     * Checks all defined safety bounds derived from HLV theory and EFE limits.
     * @param inputs The current sensor readings.
     * @return An integer representing the required safing action.
     */
    int evaluateSafety(const DsmSensorInputs& inputs);

    // Safing Action Enumerations
    enum SafingAction {
        ACTION_NONE = 0,         // No action required.
        ACTION_ROLLBACK = 1,     // Execute Idempotent Rollback (reduce amplitude,
decouple phase).
        ACTION_FULL_SHUTDOWN = 2 // Execute Emergency Collapse (remove all power).
    };

private:
    // Internal state variables for monitoring the safing procedure
    double last_estimated_Rmax_;
    bool safing_sequence_active_;

    /**
     * @brief Checks the stability conditions governed by A(t) and TCC Coupling J.
     */
    bool checkResonanceStability(double A_t, double J_coupling) const;

    /**
```

```
     * @brief Estimates the maximum local curvature scalar R from proper time dilation.
     * Uses a conservative, simplified analytical approximation (fast, independent, and always overestimates danger).
     */
    double estimateCurvatureScalar(double dilation) const;

    /**
     * @brief Checks if the estimated curvature exceeds the absolute hard threshold (EFE violation).
     */
    bool checkCurvatureViolation(double R_estimated) const;
};

// --- Function Implementations ---

DeterministicSafetyMonitor::DeterministicSafetyMonitor()
    : last_estimated_Rmax_(0.0), safing_sequence_active_(false) {}

double DeterministicSafetyMonitor::estimateCurvatureScalar(double dilation) const {
    // R_FACTOR is a calibrated constant (e.g., 10^-10)
    const double R_FACTOR = 1.0e-10;
    double time_stretch = 1.0 - dilation;

    if (time_stretch < 0) {
        // If proper time dilation is reversed, immediately report max danger.
        return std::numeric_limits<double>::infinity();
    }

    // Conservative estimation: proportional to the square of the time stretch.
    return R_FACTOR * time_stretch * time_stretch;
}

bool DeterministicSafetyMonitor::checkCurvatureViolation(double R_estimated) const {
    // Primary Defense: The absolute limit hard-wired from physics constraints.
    if (R_estimated >= DSM_Config::MAX_CURVATURE_THRESHOLD_RMAX) {
        return true;
    }
    return false;
}
```

```
bool DeterministicSafetyMonitor::checkResonanceStability(double A_t, double J_coupling)
const {
    // Secondary Defense 1: A(t) - Check stability window closure.
    if (A_t < DSM_Config::MIN_ACCEPTABLE_A_T) {
        std::cerr << "DSM FAILURE PREDICT: A(t) near minimum stable value (" << A_t << ")."
<< std::endl;
        return true;
    }

    // Secondary Defense 2: J - Check Tri-Cell Coupling limit.
    if (J_coupling > DSM_Config::MAX_TCC_COUPLING_J) {
        std::cerr << "DSM FAILURE PREDICT: TCC Coupling J exceeded safe limit (" <<
J_coupling << ")." << std::endl;
        return true;
    }

    return false;
}


int DeterministicSafetyMonitor::evaluateSafety(const DsmSensorInputs& inputs) {
    // 1. PRIMARY CHECK (FULL SHUTDOWN - Catastrophic Failure)
    double R_estimated = estimateCurvatureScalar(inputs.measured_proper_time_dilation);

    if (checkCurvatureViolation(R_estimated)) {
        safing_sequence_active_ = true;
        std::cerr << "DSM ALERT: ABSOLUTE CURVATURE THRESHOLD VIOLATION!
EXECUTING FULL SHUTDOWN." << std::endl;
        return ACTION_FULL_SHUTDOWN;
    }

    // 2. SECONDARY CHECKS (ROLLBACK - Pre-Failure Warning)
    // Checks based on HLV math pillars to prevent R_max from being violated.
    if (checkResonanceStability(inputs.measured_oscillatory_prefactor_A_t,
inputs.measured_tcc_coupling_J)) {
        if (!safing_sequence_active_) {
            safing_sequence_active_ = true;
            std::cerr << "DSM WARNING: HLV PILLAR INSTABILITY DETECTED.
EXECUTING ROLLBACK." << std::endl;
        }
```

```
        return ACTION_ROLLBACK;
    }

    // 3. TERTIARY CHECK (ROLLBACK - System Sanity)
    if (!inputs.main_control_system_healthy && inputs.current_resonance_amplitude >
DSM_Config::MIN_RESONANCE_AMPLITUDE_CUTOFF) {
        if (!safing_sequence_active_) {
            safing_sequence_active_ = true;
            std::cerr << "DSM WARNING: MAIN CONTROL FAILURE + POWER REQUEST.
EXECUTING ROLLBACK." << std::endl;
        }
        return ACTION_ROLLBACK;
    }



    // 4. Safing Deactivation - If system recovers
    if (safing_sequence_active_ && R_estimated <
DSM_Config::MAX_CURVATURE_THRESHOLD_RMAX * 0.5) {
        safing_sequence_active_ = false;
        std::cout << "DSM STATUS: Resuming normal operation. Safety margins re-established."
<< std::endl;
    }

    // 5. Default state
    return ACTION_NONE;
}

#endif // DETERMINISTIC_SAFETY_MONITOR_HPP

//_____________________________________________

// RAPS GOVERNOR WITH HLV

import time
import threading
import queue
import hashlib
import json
import random
import secrets
```

```
from typing import NamedTuple, Dict, Any, Optional

# --- CONFIG / CONSTANTS ---
DECISION_HORIZON_MS = 300
WATCHDOG_MS = 120
MAX_ACCEPTABLE_UNCERTAINTY = 0.25
MIN_CONFIDENCE_FOR_EXECUTION = 0.85
ITL_QUEUE_MAXSIZE = 8192
MERKLE_BATCH_SIZE = 32
SIM_DETERMINISTIC_SEED = None

# --- HLV MATHEMATICAL PILLAR CONSTANTS (Mirrored from DSM) ---
# These constants define the stability windows for the HLV framework.
HLV_CONSTANTS = {
    # Pillar Check 1 (EFE Limit): Used by DSM hardware for shutdown.
    "MAX_CURVATURE_THRESHOLD_RMAX": 1.0e-12,
    # Pillar Check 2 (A(t)): Must be above this for resonance stability.
    "MIN_ACCEPTABLE_A_T": 0.80,
    # Pillar Check 5 (TCC Coupling J): Must be below this to prevent runaway.
    "MAX_TCC_COUPLING_J": 1.0e+04,
}

# --- UTILITIES ---
def now_ms() -> int:
    """Returns current monotonic time in milliseconds."""
    return int(time.monotonic() * 1000)

def stable_json_hash(obj: Any) -> str:
    """Computes a stable SHA256 hash for logging and Merkle tree generation."""
    return hashlib.sha256(json.dumps(obj, sort_keys=True, separators=(',',
':')).encode()).hexdigest()

def seed_simulation(seed: Optional[int]):
    """Sets a deterministic seed for reproducible simulation runs."""
    global SIM_DETERMINISTIC_SEED
    SIM_DETERMINISTIC_SEED = seed
    if seed is not None:
        random.seed(seed)

# --- DATA STRUCTURES ---
```

```
class PredictionResult(NamedTuple):
    status: str
    mean_state: Dict[str, Any]
    cov: Dict[str, Any]
    model_version: str
    confidence: float
    id: str
    evidence: Dict[str, Any]

class Policy(NamedTuple):
    id: str
    command_set: Dict[str, Any]
    cost: float
    preconditions: Dict[str, Any]
    rollback: Dict[str, Any]

# --- OBSERVABILITY HOOKS ---
def metric_emit(name: str, value: Any, tags: Dict[str, str] = None):
    # Demo: structured print
    print(f"[METRIC] {name}={value} tags={tags}")

def audit_log(payload: Dict[str, Any]):
    print(f"[AUDIT] {json.dumps(payload)}")

#
================================================================================
========
# ITL: Thread-safe queue + Merkle batching (Implementation details omitted for brevity)
#
================================================================================
========
ITLQUEUE: "queue.Queue[Dict]" = queue.Queue(maxsize=ITL_QUEUE_MAXSIZE)
ITLSTORAGE: Dict[str, Dict] = {}
MERKLEBUFFER: list = []
ROLLBACKSTORE: Dict[str, Dict] = {}
ITL_LOCK = threading.Lock()
MERKLE_LOCK = threading.Lock()
FLUSHERSHUTDOWN = threading.Event()

def compute_merkle_root(ids: list) -> str:
```

```
    # Merkle tree implementation (unchanged)
    nodes = ids[:]
    while len(nodes) > 1:
        it = []
        for i in range(0, len(nodes), 2):
            a = nodes[i]
            b = nodes[i+1] if i+1 < len(nodes) else nodes[i]
            it.append(hashlib.sha256((a + b).encode()).hexdigest())
        nodes = it
    return nodes[0] if nodes else ""

def anchor_merkle_root(merkle_root: str, batch_ids: list):
    # Merkle anchor implementation (unchanged)
    signed_root = "SIGNED:" + merkle_root
    itl_commit({"type": "merkle_anchor", "merkle_root": merkle_root,
                "signed": signed_root, "batch_ids": batch_ids, "timestamp_ms": now_ms()})

def itl_background_flusher_loop():
    # Background thread for committing ITL entries (unchanged)
    global MERKLEBUFFER
    while not FLUSHERSHUTDOWN.is_set():
        try:
            entry = ITLQUEUE.get(timeout=0.1)
        except queue.Empty:
            continue

        entry_id = entry['id']
        payload = entry['payload']

        with ITL_LOCK:
            ITLSTORAGE[entry_id] = payload
        with MERKLE_LOCK:
            MERKLEBUFFER.append(entry_id)
            if len(MERKLEBUFFER) >= MERKLE_BATCH_SIZE:
                batch = MERKLEBUFFER[:MERKLE_BATCH_SIZE]
                MERKLEBUFFER = MERKLEBUFFER[MERKLE_BATCH_SIZE:]
                merkle_root = compute_merkle_root(batch)
                anchor_merkle_root(merkle_root, batch)
                metric_emit("itl.merkle_anchored", 1, tags={"batch_size": str(len(batch))})
        ITLQUEUE.task_done()
```

```
def start_itl_flusher():
    flusher = threading.Thread(target=itl_background_flusher_loop, daemon=True)
    flusher.start()
    return flusher

def shutdown():
    # Shutdown sequence (unchanged)
    stop_itl_flusher()
    STOPSUPERVISOR.set()
    with MERKLE_LOCK:
        if MERKLEBUFFER:
            merkle_root = compute_merkle_root(MERKLEBUFFER)
            anchor_merkle_root(merkle_root, MERKLEBUFFER)
            MERKLEBUFFER.clear()

def stop_itl_flusher():
    FLUSHERSHUTDOWN.set()
    try:
        ITLQUEUE.put_nowait({"id": "FLUSHER_STOP", "payload": {}})
    except Exception:
        pass

def itl_commit(payload: Dict[str, Any]) -> str:
    optimistic_id = stable_json_hash(payload)
    entry = {"id": optimistic_id, "payload": payload}
    try:
        ITLQUEUE.put_nowait(entry)
    except queue.Full:
        try:
            ITLQUEUE.put(entry, timeout=0.05)
        except queue.Full:
            metric_emit("itl.queue_full", 1)
            raise RuntimeError("ITL queue full: cannot commit telemetry")
    return optimistic_id

#
================================================================================
========
# HLV Mathematical Stubs (Placeholder for full PPE/PDT modeling)
```

```
#
================================================================================
========

def hlv_predict_At(current_state: Dict[str, Any], command_set: Dict[str, Any]) -> float:
    """
    STUB: Predicts the Oscillatory Prefactor A(t) based on the commanded state.
    In a full implementation, this uses the complex HLV dynamics (Pillars 1, 2, 3).
    A(t) is critical for kinetic stability; must be > MIN_ACCEPTABLE_A_T.
    """
    # Placeholder Logic: A(t) decreases with high throttle, increases with low temp
    throttle_influence = command_set.get("throttle_pct", 100.0) / 100.0
    baseline = 1.0 - (throttle_influence * 0.1)

    # Simulate a dangerous state if the valve is closed too much (e.g., A(t) drops)
    valve = command_set.get("valve_adjust", 0.0)
    if valve < -0.04:
        # Simulate a dangerous instability near the boundary
        return random.uniform(0.70, 0.85)

    return baseline + random.uniform(-0.02, 0.02)

def hlv_predict_J(current_state: Dict[str, Any], command_set: Dict[str, Any]) -> float:
    """
    STUB: Predicts the Tri-Cell Coupling constant J based on the commanded state.
    In a full implementation, this uses the TCC Lagrangian (Pillar 5).
    J is critical for preventing runaway coherent energy density; must be <
MAX_TCC_COUPLING_J.
    """
    # Placeholder Logic: J increases dramatically if throttle is high AND pressure is low
    throttle = command_set.get("throttle_pct", 100.0)

    if throttle > 99.0 and current_state.get("pressure_chamber_a", 0.0) < 2100.0:
        # Simulate a catastrophic TCC runaway when system stress is maximized
        return HLV_CONSTANTS["MAX_TCC_COUPLING_J"] * 1.5

    # Nominal J is low
    return random.uniform(50.0, 500.0)
```

```
#
================================================================================
========
# Stubs: platform-specific functionality
#
================================================================================
========
def fast_state_snapshot() -> Dict[str, Any]:
    """Retrieves the current measured state, including HLV sensor readings."""
    return {
        "pressure_chamber_a": random.uniform(2000.0, 2500.0),
        "temp_nozzle_b": random.uniform(800.0, 900.0),
        "thrust_command": random.uniform(50.0, 100.0),
        "valve_position": random.uniform(0.1, 0.9),
        # Current measured HLV parameters (used by the DSM hardware)
        "measured_proper_time_dilation": 1.0 + random.uniform(1e-15, 1e-13),
        "measured_oscillatory_prefactor_A_t": random.uniform(0.95, 1.05),
        "measured_tcc_coupling_J": random.uniform(100.0, 5000.0),
        "timestamp_ms": now_ms()
    }

def safety_monitor_validate(policy_preflight: Dict[str, Any], current_state: Dict[str, Any]) ->
bool:
    """
    Governor-level pre-flight safety check, enforcing standard bounds AND HLV stability
windows.
    """
    cmd = policy_preflight.get("command_set") or {}

    # --- 1. Standard Actuator Checks ---
    throttle = cmd.get("throttle_pct")
    if throttle is not None and not (0.0 <= throttle <= 100.0):
        itl_commit({"type": "safety_check_fail", "reason": "Throttle out of bounds", "value":
throttle})
        return False
    if policy_preflight.get("rollback") is None:
        itl_commit({"type": "safety_check_fail", "reason": "Missing rollback definition"})
        return False

    # --- 2. HLV Mathematical Pillar Constraint Validation ---
```

```
    # The Governor uses its Predictive Digital Twin (PDT) to forecast HLV parameters
    # and ensures the proposed command will not drive A(t) or J outside safe limits.

    predicted_A_t = hlv_predict_At(current_state, cmd)
    if predicted_A_t < HLV_CONSTANTS["MIN_ACCEPTABLE_A_T"]:
        itl_commit({
            "type": "safety_check_fail_hlv",
            "reason": "Predicted A(t) violates stability window (Pillar 2)",
            "predicted_A_t": predicted_A_t
        })
        return False

    predicted_J = hlv_predict_J(current_state, cmd)
    if predicted_J > HLV_CONSTANTS["MAX_TCC_COUPLING_J"]:
        itl_commit({
            "type": "safety_check_fail_hlv",
            "reason": "Predicted J violates TCC coupling limit (Pillar 5)",
            "predicted_J": predicted_J
        })
        return False

    return True

# Implementation of remaining functions (unchanged logic, only name changed for clarity)
# ... (pdt_infer, ape_get_best_policy, execute_command_on_actuators, etc.) ...

# --- Implementation of remaining functions (unchanged logic, only name changed for clarity) ---

def pdt_infer(snapshot: Dict[str, Any], horizon_ms: int) -> PredictionResult:
    if random.random() < 0.15:
        pr = PredictionResult(
            status="PREDICTED_ESE",
            mean_state={"pressure": 2650.0, "temp": 950.0},
            cov={'norm_sigma': 0.15},
            model_version="v2.1_stochastic",
            confidence=0.92,
            id=f"ESE_{now_ms()}",
            evidence={"sensor_code": "PC_A", "delta": 150}
        )
    else:
```

```
        pr = PredictionResult(
            status="NOMINAL",
            mean_state={"pressure": 2400.0, "temp": 860.0},
            cov={'norm_sigma': 0.05},
            model_version="v2.1_stochastic",
            confidence=0.99,
            id=f"NOM_{now_ms()}",
            evidence={}
        )
    metric_emit("pdt.infer_latency_ms", random.uniform(1.0, 10.0))
    return pr

def ape_generate_candidates(evidence: Dict[str, Any], mean_state: Dict[str, Any]) ->
list[Policy]:
    # Demo candidates; in production, derive from registry and current state
    return [
        Policy("POL_THROTTLE_ADJUST_001", {"throttle_pct": 98.5, "valve_adjust": -0.05},
1.5, {"min_thrust": 80.0}, {"throttle_pct": 100.0}),
        Policy("POL_VALVE_TRIM_002", {"valve_adjust": -0.08}, 1.2, {"min_thrust": 75.0},
{"valve_adjust": 0.0}),
        Policy("POL_REDUCE_THRUST_003", {"throttle_pct": 96.0}, 0.9, {"min_thrust": 70.0},
{"throttle_pct": 100.0}),
    ]

def ape_rank_and_select(evidence: Dict[str, Any], mean_state: Dict[str, Any]) -> Policy:
    candidates = ape_generate_candidates(evidence, mean_state)
    scored = []
    for pol in candidates:
        # Simple risk-aware scoring: cost + penalty for missing preconditions
        missing_preconds = 1.0 if mean_state.get("pressure", 0) < 2300.0 and
pol.preconditions.get("min_thrust", 0) > 90.0 else 0.0
        score = pol.cost + missing_preconds
        # Note: We skip safety_monitor_validate here and run it on the chosen policy later.
        valid = True # Placeholder for simplicity in ranking
        scored.append((score, valid, pol))
    # Prefer lowest cost
    scored.sort(key=lambda x: x[0])
    chosen = scored[0][2]
    itl_commit({"type": "policy_ranking", "candidates": [p.id for _, _, p in scored], "chosen":
chosen.id, "timestamp_ms": now_ms()})
```

```
    return chosen

def ape_get_best_policy(evidence: Dict[str, Any], mean_state: Dict[str, Any]) -> Policy:
    return ape_rank_and_select(evidence, mean_state)

APPLIEDTX: Dict[str, Dict[str, Any]] = {}

def execute_command_on_actuators(command_set: Dict[str, Any], tx_id: str, timeout_ms: int =
WATCHDOG_MS) -> bool:
    if tx_id in APPLIEDTX:
        metric_emit("actuator.idempotent_shortcircuit", 1)
        return True
    simulated_latency = max(0.001, random.uniform(0.003, 0.02))
    if simulated_latency * 1000 > timeout_ms:
        return False
    threading.Event().wait(simulated_latency)  # non-blocking simulation
    APPLIEDTX[tx_id] = command_set.copy()
    return True

def trigger_fallback_safe_state(reason: str = "safety_fallback"):
    payload = {"type": "fallback_triggered", "reason": reason, "timestamp_ms": now_ms()}
    itl_commit(payload)
    metric_emit("fallback.triggered", 1, tags={"reason": reason})

def uncertaintymetric(cov: Dict[str, Any]) -> float:
    return float(cov.get('norm_sigma', 0.0))

def shouldact(pred: PredictionResult) -> bool:
    if pred.status != "PREDICTED_ESE":
        return False
    conf_ok = pred.confidence >= MIN_CONFIDENCE_FOR_EXECUTION
    unc_ok = uncertaintymetric(pred.cov) <= MAX_ACCEPTABLE_UNCERTAINTY
    metric_emit("decision.conf_ok", int(conf_ok))
    metric_emit("decision.unc_ok", int(unc_ok))
    return conf_ok and unc_ok

#
================================================================================
========
# Rollback Execution & Recovery
```

```
#
================================================================================
========

def execute_rollback(policy_id: str) -> bool:
    """
    Attempt to revert system state using stored rollback metadata.
    """
    rollback = ROLLBACKSTORE.get(policy_id)
    if not rollback:
        itl_commit({"type": "rollback_missing", "policy_id": policy_id, "timestamp_ms":
now_ms()})
        trigger_fallback_safe_state(reason="rollback_missing")
        return False

    tx_id = secrets.token_hex(12)
    itl_commit({"type": "rollback_pending", "policy_id": policy_id, "tx_id": tx_id,
"timestamp_ms": now_ms()})

    success = execute_command_on_actuators(rollback, tx_id, timeout_ms=WATCHDOG_MS)
    if not success:
        itl_commit({"type": "rollback_failure", "policy_id": policy_id, "tx_id": tx_id,
"timestamp_ms": now_ms()})
        trigger_fallback_safe_state(reason="rollback_execution_failure")
        return False

    itl_commit({
        "type": "rollback_commit",
        "policy_id": policy_id,
        "tx_id": tx_id,
        "rollback_hash": stable_json_hash(rollback),
        "timestamp_ms": now_ms()
    })
    metric_emit("rollback.exec_success", 1, tags={"policy_id": policy_id})
    return True

#
================================================================================
========
# Governance loop
```

```
#
================================================================================
========
def raps_governance_cycle_once():
    loop_ts = now_ms()
    metric_emit("governance.cycle_start", 1)

    try:
        fast_state = fast_state_snapshot()
        snapshot_payload = {
            "type": "state_snapshot",
            "timestamp_ms": now_ms(),
            "fast_state_hash": stable_json_hash(fast_state),
            "fast_state_sample": fast_state
        }
        snap_id = itl_commit(snapshot_payload)

        pdt_start = now_ms()
        pred = pdt_infer(fast_state, DECISION_HORIZON_MS)
        pdt_latency = now_ms() - pdt_start

        itl_commit({
            "type": "prediction_commit",
            "prediction_id": pred.id,
            "model_version": pred.model_version,
            "confidence": pred.confidence,
            "uncertainty": uncertaintymetric(pred.cov),
            "evidence": pred.evidence,
            "ref_snapshot": snap_id,
            "timestamp_ms": now_ms()
        })
        metric_emit("pdt.latency_ms", pdt_latency)

        if shouldact(pred):
            itl_commit({"type": "ese_alert", "prediction_id": pred.id, "timestamp_ms": now_ms()})
            policy = ape_get_best_policy(pred.evidence, pred.mean_state)
            preflight = {
                "policy_id": policy.id,
                "command_set": policy.command_set,
                "rollback": policy.rollback,
```

```
            "cost": policy.cost,
            "timestamp_ms": now_ms(),
            "prediction_id": pred.id,
            "model_version": pred.model_version
        }
        itl_commit({"type": "policy_preflight", **preflight})

        # CRITICAL HLV SAFETY VALIDATION
        if not safety_monitor_validate(preflight, fast_state): # Pass fast_state for HLV prediction
            itl_commit({"type": "policy_rejected", "policy_id": policy.id, "reason":
"HLV_VIOLATION", "timestamp_ms": now_ms()})
            trigger_fallback_safe_state(reason="safety_monitor_reject_hlv")
            return

        tx_id = secrets.token_hex(12)
        itl_commit({"type": "command_pending", "policy_id": policy.id, "tx_id": tx_id,
"timestamp_ms": now_ms()})

        exec_start = now_ms()
        success = execute_command_on_actuators(policy.command_set, tx_id,
timeout_ms=WATCHDOG_MS)
        exec_elapsed = now_ms() - exec_start

        if not success or exec_elapsed > WATCHDOG_MS:
            itl_commit({
                "type": "execution_failure",
                "policy_id": policy.id,
                "tx_id": tx_id,
                "elapsed_ms": exec_elapsed,
                "timestamp_ms": now_ms()
            })
            rollback_success = execute_rollback(policy.id)
            if not rollback_success:
                trigger_fallback_safe_state(reason="execution_failure_or_timeout")
            return

        itl_commit({
            "type": "command_commit",
            "actor": "APE",
            "policy_id": policy.id,
```

```
            "tx_id": tx_id,
            "command_set_hash": stable_json_hash(policy.command_set),
            "reference_prediction_id": pred.id,
            "timestamp_ms": now_ms()
        })

        rollback_hash = stable_json_hash(policy.rollback)
        ROLLBACKSTORE[policy.id] = policy.rollback.copy()
        itl_commit({"type": "rollback_metadata", "policy_id": policy.id, "rollback_hash":
rollback_hash, "timestamp_ms": now_ms()})

        post_state = fast_state_snapshot()
        itl_commit({"type": "post_state_check", "policy_id": policy.id, "pre_hash":
stable_json_hash(fast_state), "post_hash": stable_json_hash(post_state), "timestamp_ms":
now_ms()})
        metric_emit("policy.exec_success", 1, tags={"policy_id": policy.id})

    else:
        itl_commit({"type": "nominal_trace", "timestamp_ms": now_ms()})
        metric_emit("governance.nominal", 1)

  except Exception as e:
    itl_commit({"type": "governance_exception", "error": str(e), "timestamp_ms": now_ms()})
    trigger_fallback_safe_state(reason="supervisor_exception")

  loop_elapsed = now_ms() - loop_ts
  metric_emit("governance.loop_elapsed_ms", loop_elapsed)
  if loop_elapsed > DECISION_HORIZON_MS:
    itl_commit({"type": "governance_budget_violation", "elapsed_ms": loop_elapsed,
"timestamp_ms": now_ms()})
    metric_emit("governance.budget_violation", 1)

# Supervisor
STOPSUPERVISOR = threading.Event()
def run_governance_supervisor(cycle_interval_ms: int = 100):
  while not STOPSUPERVISOR.is_set():
    start = now_ms()
    raps_governance_cycle_once()
    elapsed = now_ms() - start
    sleep_ms = max(0, cycle_interval_ms - elapsed)
```

```
        time.sleep(sleep_ms / 1000.0)

#
================================================================================
========
# Redundant Supervisors: A/B governance with cross-check + failover
#
================================================================================
========

SUPERVISOR_ERRORS = {"A": 0, "B": 0}
ACTIVE_SUPERVISOR = threading.Event()  # set => A active, clear => B active
ACTIVE_SUPERVISOR.set()  # start with A

def governance_cycle_guarded(label: str):
    try:
        raps_governance_cycle_once()
    except Exception as e:
        itl_commit({"type": "redundant_supervisor_exception", "who": label, "error": str(e),
"timestamp_ms": now_ms()})
        SUPERVISOR_ERRORS[label] += 1
        # failover threshold (tunable)
        if SUPERVISOR_ERRORS[label] >= 3:
            # switch active supervisor
            if label == "A":
                ACTIVE_SUPERVISOR.clear()
            else:
                ACTIVE_SUPERVISOR.set()
            itl_commit({"type": "supervisor_failover", "from": label, "to": "B" if label == "A" else
"A", "timestamp_ms": now_ms()})
        trigger_fallback_safe_state(reason=f"redundant_supervisor_exception_{label}")

def run_redundant_supervisors(interval_ms: int = 100):
    stop = threading.Event()

    def runner(label: str, active_when_set: bool):
        while not stop.is_set():
            # Only run if this thread is currently active
            if ACTIVE_SUPERVISOR.is_set() == active_when_set:
                start = now_ms()
```

```
            governance_cycle_guarded(label)
            elapsed = now_ms() - start
            sleep_ms = max(0, interval_ms - elapsed)
            time.sleep(sleep_ms / 1000.0)
        else:
            time.sleep(0.02)

    ta = threading.Thread(target=runner, args=("A", True), daemon=True)
    tb = threading.Thread(target=runner, args=("B", False), daemon=True)
    ta.start(); tb.start()
    return stop, ta, tb


#
================================================================================
========
# Ground Anchoring / Downlink (demo): mirror ITL entries to a file
#
================================================================================
========

DOWNLINK_FILE = "ground_anchor_manifest.log"
DOWNLINK_LOCK = threading.Lock()

def ground_anchor_emit(entry: Dict[str, Any]):
    line = json.dumps({"downlink_ts": now_ms(), "entry": entry}, separators=(',', ':'))
    with DOWNLINK_LOCK:
        with open(DOWNLINK_FILE, "a", encoding="utf-8") as f:
            f.write(line + "\n")

def ground_anchor_flusher_loop(poll_ms: int = 100):
    cursor_seen = set()
    while not FLUSHERSHUTDOWN.is_set():
        keys = list(ITLSTORAGE.keys())
        for k in keys:
            if k not in cursor_seen:
                ground_anchor_emit({"id": k, "payload": ITLSTORAGE[k]})
                cursor_seen.add(k)
        time.sleep(poll_ms / 1000.0)
```

```
def start_ground_anchor_flusher():
    t = threading.Thread(target=ground_anchor_flusher_loop, daemon=True)
    t.start()
    return t

#
================================================================================
========
# Demo Main: spin up ITL flusher, ground anchor, redundant supervisors,
#            inject a forced failure to exercise rollback + safing
#
================================================================================
========

def inject_forced_failure_once():
    """
    Monkey-patch execute_command_on_actuators to fail once,
    then restore original behavior. This forces the rollback path.
    """
    original = execute_command_on_actuators
    triggered = {"done": False}

    def failing_once(command_set: Dict[str, Any], tx_id: str, timeout_ms: int =
WATCHDOG_MS) -> bool:
        if not triggered["done"]:
            triggered["done"] = True
            # Simulate a timeout breach to force failure
            return False
        return original(command_set, tx_id, timeout_ms)

    globals()["execute_command_on_actuators"] = failing_once

    def restore():
        globals()["execute_command_on_actuators"] = original

    return restore

def demo_main(run_seconds: float = 3.0, interval_ms: int = 100):
    print("=== RAPS DEMO START ===")
    seed_simulation(42)
```

```
    # Start sinks
    flusher_thread = start_itl_flusher()
    ground_thread = start_ground_anchor_flusher()

    # Start redundant supervisors (A/B)
    stop_redundant, ta, tb = run_redundant_supervisors(interval_ms=interval_ms)

    # Inject a single forced failure to light up rollback path
    restore_exec = inject_forced_failure_once()

    # Run for a short period
    start_wall = time.time()
    while time.time() - start_wall < run_seconds:
        time.sleep(0.05)

    # Restore actuator path
    restore_exec()

    # Shutdown sequence
    stop_redundant.set()
    STOPSUPERVISOR.set()  # in case single supervisor is used elsewhere
    print("Stopping redundant supervisors...")
    time.sleep(0.2)

    print("Anchoring any remaining Merkle buffer and stopping flusher...")
    shutdown()  # flush + anchor remaining entries
    time.sleep(0.2)
    print("=== RAPS DEMO COMPLETE ===")

# If you want to run directly:
if __name__ == "__main__":
    demo_main(run_seconds=3.0, interval_ms=100)

// HLV Physics Engine Implementation

#include "PropulsionPhysicsEngine.hpp"
#include <cmath>       // For std::sqrt, std::fabs, std::sin, std::cos
#include <algorithm>   // For std::max, std::min
#include <numeric>     // For std::inner_product (dot product)
```

```
// Helper function for vector normalization
namespace {
    std::array<float, 3> normalize(const std::array<float, 3>& vec) {
        float magnitude = std::sqrt(vec[0] * vec[0] + vec[1] * vec[1] + vec[2] * vec[2]);
        if (magnitude < 1e-6f) { // Avoid division by zero
            return {0.0f, 0.0f, 0.0f};
        }
        return {vec[0] / magnitude, vec[1] / magnitude, vec[2] / magnitude};
    }
}

// Initializes the HLV engine
void PropulsionPhysicsEngine::init() {
    // Nothing to initialize for a stateless predictor.
}

// Calculates the net acceleration (m/s^2) due to all forces
std::array<float, 3> PropulsionPhysicsEngine::calculate_acceleration(
    const std::array<float, 3>& pos_m,
    const std::array<float, 3>& vel_m_s,
    float mass_kg,
    float thrust_mag_N,
    const std::array<float, 3>& thrust_dir_vec) const {

    std::array<float, 3> net_force = {0.0f, 0.0f, 0.0f};
    float r = std::sqrt(pos_m[0]*pos_m[0] + pos_m[1]*pos_m[1] + pos_m[2]*pos_m[2]);

    // 1. Gravity (Point Mass Earth Model)
    // F_gravity = -G * M_earth * mass / r^2 * (r_vec / r)
    if (r > RAPSConfig::R_EARTH_M / 2.0f) { // Sanity check to prevent singularity near center
        float mag_gravity = -(RAPSConfig::G_GRAVITATIONAL_CONSTANT *
RAPSConfig::M_EARTH_KG * mass_kg) / (r * r);
        std::array<float, 3> pos_norm = normalize(pos_m);
        net_force[0] += mag_gravity * pos_norm[0];
        net_force[1] += mag_gravity * pos_norm[1];
        net_force[2] += mag_gravity * pos_norm[2];
    }

    // 2. Thrust Force
```

```
    // F_thrust = Thrust_Mag * Thrust_Direction_Vector
    net_force[0] += thrust_mag_N * thrust_dir_vec[0];
    net_force[1] += thrust_mag_N * thrust_dir_vec[1];
    net_force[2] += thrust_mag_N * thrust_dir_vec[2];

    // 3. Simple Atmospheric Drag (Active only near Earth)
    if (r < RAPSConfig::R_EARTH_M + 100000.0f) { // Up to 100km altitude
        std::array<float, 3> vel_norm = normalize(vel_m_s);
        // Drag F_D = -0.5 * rho * v^2 * C_D * A, simplified to F_D = -k * v * v_norm
        float vel_mag = std::sqrt(vel_m_s[0]*vel_m_s[0] + vel_m_s[1]*vel_m_s[1] +
vel_m_s[2]*vel_m_s[2]);
        float drag_mag = -RAPSConfig::ATMOSPHERIC_DRAG_COEFF * vel_mag * vel_mag;
// Simple model
        net_force[0] += drag_mag * vel_norm[0];
        net_force[1] += drag_mag * vel_norm[1];
        net_force[2] += drag_mag * vel_norm[2];
    }

    // Acceleration = Net Force / Mass (A = F/m)
    return {
        net_force[0] / mass_kg,
        net_force[1] / mass_kg,
        net_force[2] / mass_kg
    };
}

// Predicts the future state based on current state and control inputs
PhysicsState PropulsionPhysicsEngine::predict_state(const PhysicsState& initial_state,
                                                    const PhysicsControlInput& control_input) const {
    PhysicsState next_state = initial_state; // Start with current state
    uint32_t remaining_time_ms = control_input.simulation_duration_ms;

    // Convert thrust from kN to N for calculations
    const float thrust_mag_N = control_input.thrust_magnitude_kN * 1000.0f;
    const float flow_rate = control_input.propellant_flow_kg_s;

    // Calculate the thrust direction vector (simplified spherical coordinates)
    // Assuming Z is the nominal thrust axis, and X/Y are gimbal planes.
    // The attitude quaternion update (d/dt Q) is omitted for this simple prediction,
    // assuming the vehicle attitude is perfectly controlled to the commanded gimbal angle.
```

```
    float theta = control_input.gimbal_theta_rad;
    float phi = control_input.gimbal_phi_rad;

    // Direction vector of thrust in the body frame (simplified)
    // Body frame is assumed aligned with inertial frame for simplicity in this step.
    std::array<float, 3> thrust_dir_vec = {
        std::sin(theta) * std::cos(phi), // X component
        std::sin(theta) * std::sin(phi), // Y component
        std::cos(theta)                  // Z component
    };
    thrust_dir_vec = normalize(thrust_dir_vec);

    // Simulate in small, fixed time steps using Euler integration (for simplicity and real-time
safety)
    while (remaining_time_ms > 0) {
        uint32_t dt_ms = std::min(remaining_time_ms, PHYSICS_DT_MS);
        float dt_s = static_cast<float>(dt_ms) / 1000.0f;

        // 1. Calculate Acceleration (A)
        std::array<float, 3> acc = calculate_acceleration(
            next_state.position_m,
            next_state.velocity_m_s,
            next_state.mass_kg,
            thrust_mag_N,
            thrust_dir_vec);

        // 2. Update Velocity (V_new = V_old + A * dt)
        next_state.velocity_m_s[0] += acc[0] * dt_s;
        next_state.velocity_m_s[1] += acc[1] * dt_s;
        next_state.velocity_m_s[2] += acc[2] * dt_s;

        // 3. Update Position (P_new = P_old + V_new * dt)
        next_state.position_m[0] += next_state.velocity_m_s[0] * dt_s;
        next_state.position_m[1] += next_state.velocity_m_s[1] * dt_s;
        next_state.position_m[2] += next_state.velocity_m_s[2] * dt_s;

        // 4. Update Mass (M_new = M_old - flow_rate * dt)
        float mass_loss = flow_rate * dt_s;
        next_state.mass_kg = std::max(MIN_MASS_KG, next_state.mass_kg - mass_loss);
```

```
        // 5. Update Time
        remaining_time_ms -= dt_ms;
        next_state.timestamp_ms += dt_ms;
    }

    return next_state;
}

// A simplified validation check based on physical limits
bool PropulsionPhysicsEngine::is_state_physically_plausible(const PhysicsState& state) const {
    // Check if mass is reasonable
    if (state.mass_kg < MIN_MASS_KG) return false;

    // Check if position is too close to the singularity (e.g., inside the Earth)
    float radius = std::sqrt(state.position_m[0]*state.position_m[0] +
                             state.position_m[1]*state.position_m[1] +
                             state.position_m[2]*state.position_m[2]);
    if (radius < RAPSConfig::R_EARTH_M * 0.9f) return false; // Vehicle is 10% or more into
the Earth

    // Velocity bounds
    // Note: Max velocity is very high (orbital speed), so min check is more relevant
    if (state.velocity_m_s[0] < MIN_VELOCITY_M_S || state.velocity_m_s[1] <
MIN_VELOCITY_M_S || state.velocity_m_s[2] < MIN_VELOCITY_M_S) return false;

    return true;
}

// RAPS Data Definitions

#ifndef RAPS_DEFINITIONS_HPP
#define RAPS_DEFINITIONS_HPP

#include <cstdint>
#include <cstring> // For std::memcmp, std::memset
#include <array>
#include <optional>
#include "PropulsionPhysicsEngine.hpp" // For PhysicsState, PhysicsControlInput
```

```
// ================================================================================
// Configuration Constants
// ================================================================================
namespace RAPSConfig {
    constexpr uint32_t DECISION_HORIZON_MS = 300;     // Supervisory horizon
    constexpr uint32_t WATCHDOG_MS = 120;             // Max allowed exec latency for APE exec path
    constexpr float MAX_ACCEPTABLE_UNCERTAINTY = 0.25f;
    constexpr float MIN_CONFIDENCE_FOR_EXECUTION = 0.85f;
    constexpr size_t ITL_QUEUE_SIZE = 128;            // Smaller for embedded
    constexpr size_t MERKLE_BATCH_SIZE = 32;
    constexpr size_t MAX_ROLLBACK_STORE = 16;
    constexpr float AILEE_CONFIDENCE_ACCEPTED = 0.90f;
    constexpr float AILEE_CONFIDENCE_BORDERLINE = 0.70f;
    constexpr float AILEE_GRACE_THRESHOLD = 0.72f;    // Slightly lower for grace

    // AILEE Consensus Layer Specifics (for HLV Dynamics)
    // Nominal targets are now related to trajectory endpoints
    static constexpr float NOMINAL_ALTITUDE_TARGET_M = 100000.0f; // 100km target
    static constexpr float NOMINAL_VELOCITY_TARGET_M_S = 7000.0f; // 7km/s target (near orbital)

    static constexpr float ACCEPT_POSITION_DEV_M = 500.0f; // Max deviation from predicted path
    static constexpr float ACCEPT_VELOCITY_DEV_M_S = 20.0f;
    static constexpr float ACCEPT_MASS_DEV_KG = 5.0f;
}

// ================================================================================
// Core Data Structures
// ================================================================================
```

```
// SHA256 hash representation
struct Hash256 {
    uint8_t data[32];

    bool operator==(const Hash256& other) const {
        return std::memcmp(data, other.data, 32) == 0;
    }
    bool operator!=(const Hash256& other) const {
        return !(*this == other);
    }
    // For invalid/null hash, all zeros
    static Hash256 null_hash() {
        Hash256 h{};
        std::memset(h.data, 0, 32);
        return h;
    }
    bool is_null() const {
        return *this == null_hash();
    }
};

// Prediction result from Digital Twin (PDT)
struct PredictionResult {
    enum class Status : uint8_t {
        NOMINAL,
        PREDICTED_ESE, // Estimated Safety Excursion
        INVALID
    };

    Status status;
    // Core prediction is the expected state at the end of the horizon
    PhysicsState predicted_end_state;
    float confidence;
    float uncertainty;
    uint32_t timestamp_ms;
    Hash256 prediction_id; // Hash of the prediction content
};

// Policy command set (from APE)
struct Policy {
```

```
    char id[32]; // Unique ID for the policy
    float thrust_magnitude_kN;
    float gimbal_theta_rad;
    float gimbal_phi_rad;
    float cost; // Lower cost is better (e.g., lower propellant usage)
    Hash256 policy_hash; // Hash of the policy content for integrity check
};

// Rollback metadata (for the Rollback Store)
struct RollbackPlan {
    char policy_id[32]; // Original policy this rollback is for
    float thrust_magnitude_kN;
    float gimbal_theta_rad;
    float gimbal_phi_rad;
    Hash256 rollback_hash; // Hash of the rollback command set
    bool valid; // If this rollback plan is considered valid
};

// --- AILEE Specific Data Structures (Unchanged, relies on confidence/status) ---

// Ailee validation statuses
enum class AileeStatus : uint8_t {
    UNDEFINED,
    ACCEPTED,
    BORDER_LINE,
    OUTRIGHT_REJECTED,
    GRACE_PASS,
    GRACE_FAIL,
    CONSENSUS_PASS,
    CONSENSUS_FAIL
};

// Data payload for Ailee layers
struct AileeDataPayload {
    PredictionResult  pred_result;
    std::optional<Policy> proposed_policy;
    float             current_raw_confidence;
};

// ITL Entry (compact embedded format - updated payloads)
```

```
struct ITLEntry {
    // --- PAYLOAD DEFINITIONS FOR ITLEntry ---
    // StateSnapshotPayload now references the full HLV state structure
    struct StateSnapshotPayload {
        Hash256 snapshot_hash;
        PhysicsState current_state; // Store the state directly
    };
    struct PredictionCommitPayload {
        Hash256 prediction_id;
        float confidence;
        float uncertainty;
        Hash256 ref_snapshot_id;
        PhysicsState end_state; // Store the predicted end state
    };
    struct ESEAlertPayload {
        Hash256 prediction_id;
        PhysicsState violating_state; // The state that caused the alert
    };
    struct PolicyPreflightPayload {
        Hash256 policy_hash;
        Hash256 prediction_id;
        float cost;
    };
    struct CommandExecutionPayload {
        Hash256 policy_id;
        char tx_id[24];
        Hash256 command_set_hash;
        Hash256 reference_prediction_id;
        uint32_t elapsed_ms;
    };
    struct RollbackMetadataPayload {
        Hash256 policy_id;
        Hash256 rollback_hash;
    };
    struct FallbackTriggeredPayload { char reason[32]; };
    struct MerkleAnchorPayload { Hash256 merkle_root; };
    struct GovernanceBudgetViolationPayload { uint32_t elapsed_ms; };
    struct NominalTracePayload { /* Empty payload */ };
    struct SupervisorExceptionPayload { char reason[32]; };
    struct AileeSafetyStatusPayload {
```

```
    AileeStatus status;
    float confidence_at_decision;
};
struct AileeGraceResultPayload {
    bool grace_pass;
    float confidence_after_grace;
};
struct AileeConsensusResultPayload {
    AileeStatus status;
};

union PayloadData {
    StateSnapshotPayload            state_snapshot;
    PredictionCommitPayload         prediction_commit;
    ESEAlertPayload                 ese_alert;
    PolicyPreflightPayload          policy_preflight;
    CommandExecutionPayload         command_execution;
    RollbackMetadataPayload         rollback_metadata;
    FallbackTriggeredPayload        fallback_triggered;
    MerkleAnchorPayload             merkle_anchor;
    GovernanceBudgetViolationPayload governance_budget_violation;
    NominalTracePayload             nominal_trace;
    SupervisorExceptionPayload      supervisor_exception;
    AileeSafetyStatusPayload        ailee_safety_status;
    AileeGraceResultPayload         ailee_grace_result;
    AileeConsensusResultPayload     ailee_consensus_result;
};

enum class Type : uint8_t {
    STATE_SNAPSHOT,
    PREDICTION_COMMIT,
    ESE_ALERT,
    POLICY_PREFLIGHT,
    COMMAND_PENDING,
    EXECUTION_FAILURE,
    COMMAND_COMMIT,
    ROLLBACK_METADATA,
    ROLLBACK_COMMIT,
    FALLBACK_TRIGGERED,
    MERKLE_ANCHOR,
```

```
        GOVERNANCE_BUDGET_VIOLATION,
        NOMINAL_TRACE,
        SUPERVISOR_EXCEPTION,
        AILEE_SAFETY_STATUS,
        AILEE_GRACE_RESULT,
        AILEE_CONSENSUS_RESULT
    };

    Type type;
    uint32_t timestamp_ms;
    Hash256 entry_id;
    PayloadData payload;
    uint16_t payload_len;

    ITLEntry() : type(Type::NOMINAL_TRACE), timestamp_ms(0), payload_len(0) {
        entry_id = Hash256::null_hash();
        std::memset(&payload, 0, sizeof(PayloadData));
    }
};

#endif // RAPS_DEFINITIONS_HPP

//

#ifndef PLATFORM_HAL_HPP
#define PLATFORM_HAL_HPP

#include <cstdint>
#include <cstddef> // For size_t
#include <string>  // For generate_tx_id return type
#include "RAPSDefinitions.hpp" // For Hash256

//
================================================================================
========
// Platform Abstraction Layer (INTERFACE FOR TARGET HARDWARE)
//
================================================================================
========
// This class provides a standardized interface to hardware-specific functions.
```

```
// All methods are static to simplify access and imply global system resources.
// In a real RTOS, thread-safe access to these resources must be ensured.
class PlatformHAL {
public:
    // Monotonic millisecond timestamp (replace with RTOS tick or hardware timer)
    static uint32_t now_ms();

    // Cryptographic operations (replace with HSM or certified crypto library)
    static Hash256 sha256(const void* data, size_t len);
    static bool ed25519_sign(const Hash256& msg, uint8_t signature[64]);

    // Flash/persistent storage (replace with robust, redundant flash drivers)
    static bool flash_write(uint32_t address, const void* data, size_t len);
    static bool flash_read(uint32_t address, void* data, size_t len);

    // Actuator interface (idempotent, transaction-aware, non-blocking)
    // tx_id is used for idempotency tracking on the actuator side.
    static bool actuator_execute(const char* tx_id, float throttle, float valve, uint32_t
timeout_ms);

    // Telemetry downlink queue (replace with flight data recorder / satcom interface)
    static bool downlink_queue(const void* data, size_t len);

    // Metric emission (replace with flight telemetry system)
    static void metric_emit(const char* name, float value);
    static void metric_emit(const char* name, float value, const char* tag_key, const char*
tag_value);

    // Random number generation for stubs (NOT for crypto or mission-critical logic)
    static void seed_rng_for_stubs(uint32_t seed);
    static float random_float(float min, float max);
    static std::string generate_tx_id(); // Generates a unique transaction ID

private:
    static std::mt19937_64 rng_; // Mersenne Twister RNG instance
    static bool rng_seeded_; // Track if RNG is seeded
};

#endif // PLATFORM_HAL_HPP
```

```
// Platform Abstraction Layer Implementation

#include "PlatformHAL.hpp"
#include <random>      // For std::mt19937_64, std::uniform_real_distribution
#include <iostream>    // For demo metric_emit, remove for production
#include <iomanip>     // For std::hex, std::setw
#include <string>      // For std::string usage
#include <chrono>      // For std::chrono in now_ms stub
#include <cstring>     // For std::memcpy

// Static members for RNG initialization
std::mt19937_64 PlatformHAL::rng_;
bool PlatformHAL::rng_seeded_ = false;

// Monotonic millisecond timestamp
uint32_t PlatformHAL::now_ms() {
   // Uses the system clock in a host environment, which is fine for the demo.
   // In a flight environment, this would read a hardware register.
   return std::chrono::duration_cast<std::chrono::milliseconds>(
            std::chrono::steady_clock::now().time_since_epoch()).count();
}

// Cryptographic operations (STUBS)
Hash256 PlatformHAL::sha256(const void* data, size_t len) {
   Hash256 h{};
   if (data && len > 0) {
      // Placeholder: returns a simple, non-cryptographic hash for simulation.
      uint64_t sum = 0;
      const uint8_t* byte_data = static_cast<const uint8_t*>(data);
      for (size_t i = 0; i < len; ++i) {
         sum += byte_data[i];
      }
      // Use sum and len to create a somewhat unique (for demo) hash
      std::memcpy(h.data, &sum, std::min(sizeof(sum), sizeof(h.data)));
      h.data[8] = static_cast<uint8_t>(len & 0xFF); // Mix in length
      h.data[9] = static_cast<uint8_t>((len >> 8) & 0xFF);
      // Fill remaining with some deterministic-ish value for uniqueness in demo
      for(size_t i = 16; i < 32; ++i) h.data[i] = static_cast<uint8_t>(i + (sum % 100) + (len %
50));
   }
```

```
    return h;
}

bool PlatformHAL::ed25519_sign(const Hash256& msg, uint8_t signature[64]) {
    // Placeholder: always succeeds in demo.
    std::memset(signature, 0xDE, 64); // Dummy signature
    return true;
}

// Flash/persistent storage (STUBS)
bool PlatformHAL::flash_write(uint32_t address, const void* data, size_t len) {
    if (!rng_seeded_) { seed_rng_for_stubs(1); } // Seed on first use if needed
    static std::uniform_real_distribution<float> dist(0.0f, 1.0f);
    if (dist(rng_) < 0.005f) { // 0.5% chance of simulated write failure
        return false;
    }
    return true;
}

bool PlatformHAL::flash_read(uint32_t address, void* data, size_t len) {
    std::memset(data, 0, len); // Dummy read
    return true;
}

// Actuator interface (STUB) - Assumed to be simplified for the HLV
bool PlatformHAL::actuator_execute(const char* tx_id, float thrust_kN, float gimbal_angle_rad,
uint32_t timeout_ms) {
    if (!rng_seeded_) { seed_rng_for_stubs(1); }
    static std::uniform_real_distribution<float> dist(0.003f, 0.02f);
    uint32_t simulated_latency_ms = static_cast<uint32_t>(dist(rng_) * 1000.0f);

    if (simulated_latency_ms > timeout_ms) {
        return false; // Simulated timeout
    }
    return true;
}

// Telemetry downlink queue (STUB)
bool PlatformHAL::downlink_queue(const void* data, size_t len) {
    return true;
```

```
}

// Metric emission (STUB)
void PlatformHAL::metric_emit(const char* name, float value) {
   // std::cout << "[METRIC] " << name << "=" << value << "\n";
}

void PlatformHAL::metric_emit(const char* name, float value, const char* tag_key, const char*
tag_value) {
   // std::cout << "[METRIC] " << name << "=" << value << ", " << tag_key << "=" <<
tag_value << "\n";
}

// Random number generation for stubs (NOT for crypto or mission-critical logic)
void PlatformHAL::seed_rng_for_stubs(uint32_t seed) {
   rng_.seed(seed);
   rng_seeded_ = true;
}

float PlatformHAL::random_float(float min, float max) {
   if (!rng_seeded_) { seed_rng_for_stubs(1); }
   return std::uniform_real_distribution<float>(min, max)(rng_);
}

std::string PlatformHAL::generate_tx_id() {
   if (!rng_seeded_) { seed_rng_for_stubs(1); }
   std::string hex_chars = "0123456789abcdef";
   std::string tx_id_str;
   tx_id_str.reserve(24);
   for (int i = 0; i < 24; ++i) {
      tx_id_str += hex_chars[std::uniform_int_distribution<int>(0, 15)(rng_)];
   }
   return tx_id_str;
}

//ITL Manager Interface

#ifndef ITL_MANAGER_HPP
#define ITL_MANAGER_HPP
```

```
#include <cstdint>
#include <cstddef> // For size_t
#include "RAPSDefinitions.hpp"
#include "PlatformHAL.hpp" // For PlatformHAL::sha256, metric_emit etc.
#include <vector> // Required for Merkle calculation in implementation

//
================================================================================
========
// Immutable Telemetry Ledger (ITL) Manager
//
================================================================================
========
// Manages a thread-safe (conceptual) queue for ITL entries and Merkle batching.
class ITLManager {
private:
    // Static allocation for queue (avoids heap fragmentation)
    ITLEntry queue_[RAPSConfig::ITL_QUEUE_SIZE];
    size_t queue_head_ = 0;
    size_t queue_tail_ = 0;
    size_t queue_count_ = 0;

    Hash256 merkle_buffer_[RAPSConfig::MERKLE_BATCH_SIZE];
    size_t merkle_count_ = 0;

    uint32_t flash_write_cursor_ = 0; // Track flash position

    // NOTE ON CONCURRENCY:
    // In a multi-threaded/multi-tasking RTOS environment, all accesses to
    // internal state MUST be protected by an RTOS-specific mutex.

    Hash256 compute_merkle_root(const Hash256* ids, size_t count) const;
    void anchor_merkle_root(const Hash256& root);

public:
    // Initializes the ITL Manager
    void init();

    // Non-blocking commit (returns optimistic ID).
    // Returns null_hash() if queue is full.
```

```
    Hash256 commit(const ITLEntry& entry);

    // Background processing (call from low-priority task)
    void flush_pending();

    // Merkle batch processing (called by flush_pending when batch is full)
    void process_merkle_batch();
};

#endif // ITL_MANAGER_HPP

// ITL Manager Implementation

#include "ITLManager.hpp"
#include <algorithm> // For std::min
#include <iostream>  // For debug prints, remove for production
#include <vector>    // Required for Merkle calculation
#include <cstring>   // For std::memcpy

void ITLManager::init() {
    queue_head_ = 0;
    queue_tail_ = 0;
    queue_count_ = 0;
    merkle_count_ = 0;
    flash_write_cursor_ = 0;
    // In a real system, you might read flash_write_cursor_ from NVM to resume.
}

Hash256 ITLManager::commit(const ITLEntry& entry_template) {
    // CRITICAL: In a multi-threaded environment, this section needs a mutex.
    if (queue_count_ >= RAPSConfig::ITL_QUEUE_SIZE) {
        PlatformHAL::metric_emit("itl.queue_full", 1.0f);
        return Hash256::null_hash(); // Signal failure
    }

    // Create a mutable copy to set the ID and payload length correctly
    ITLEntry entry = entry_template;

    // Determine actual payload length based on type for hashing
    // CRITICAL: This implementation is complex due to the union.
```

```
    size_t effective_payload_len = 0;
    switch (entry.type) {
        case ITLEntry::Type::STATE_SNAPSHOT: effective_payload_len =
sizeof(ITLEntry::StateSnapshotPayload); break;
        case ITLEntry::Type::PREDICTION_COMMIT: effective_payload_len =
sizeof(ITLEntry::PredictionCommitPayload); break;
        case ITLEntry::Type::ESE_ALERT: effective_payload_len =
sizeof(ITLEntry::ESEAlertPayload); break;
        case ITLEntry::Type::POLICY_PREFLIGHT: effective_payload_len =
sizeof(ITLEntry::PolicyPreflightPayload); break;
        case ITLEntry::Type::COMMAND_PENDING:
        case ITLEntry::Type::EXECUTION_FAILURE:
        case ITLEntry::Type::COMMAND_COMMIT:
        case ITLEntry::Type::ROLLBACK_COMMIT:
            effective_payload_len = sizeof(ITLEntry::CommandExecutionPayload); break;
        case ITLEntry::Type::ROLLBACK_METADATA: effective_payload_len =
sizeof(ITLEntry::RollbackMetadataPayload); break;
        case ITLEntry::Type::FALLBACK_TRIGGERED: effective_payload_len =
sizeof(ITLEntry::FallbackTriggeredPayload); break;
        case ITLEntry::Type::MERKLE_ANCHOR: effective_payload_len =
sizeof(ITLEntry::MerkleAnchorPayload); break;
        case ITLEntry::Type::GOVERNANCE_BUDGET_VIOLATION: effective_payload_len =
sizeof(ITLEntry::GovernanceBudgetViolationPayload); break;
        case ITLEntry::Type::NOMINAL_TRACE: effective_payload_len =
sizeof(ITLEntry::NominalTracePayload); break;
        case ITLEntry::Type::SUPERVISOR_EXCEPTION: effective_payload_len =
sizeof(ITLEntry::SupervisorExceptionPayload); break;
        case ITLEntry::Type::AILEE_SAFETY_STATUS: effective_payload_len =
sizeof(ITLEntry::AileeSafetyStatusPayload); break;
        case ITLEntry::Type::AILEE_GRACE_RESULT: effective_payload_len =
sizeof(ITLEntry::AileeGraceResultPayload); break;
        case ITLEntry::Type::AILEE_CONSENSUS_RESULT: effective_payload_len =
sizeof(ITLEntry::AileeConsensusResultPayload); break;
        default: effective_payload_len = 0; break;
    }
    entry.payload_len = static_cast<uint16_t>(effective_payload_len);

    // Generate entry ID: Hash the relevant parts of the entry for integrity
    std::array<uint8_t, sizeof(entry.type) + sizeof(entry.timestamp_ms) +
sizeof(ITLEntry::PayloadData)> hash_input_buffer;
```

```
    size_t offset = 0;
    std::memcpy(hash_input_buffer.data() + offset, &entry.type, sizeof(entry.type)); offset +=
sizeof(entry.type);
    std::memcpy(hash_input_buffer.data() + offset, &entry.timestamp_ms,
sizeof(entry.timestamp_ms)); offset += sizeof(entry.timestamp_ms);
    if (effective_payload_len > 0) {
        std::memcpy(hash_input_buffer.data() + offset, &entry.payload, effective_payload_len); //
Copy active union data
        offset += effective_payload_len;
    }
    entry.entry_id = PlatformHAL::sha256(hash_input_buffer.data(), offset);

    // Queue the entry
    queue_[queue_tail_] = entry; // Copy the fully prepared entry
    queue_tail_ = (queue_tail_ + 1) % RAPSConfig::ITL_QUEUE_SIZE;
    queue_count_++;

    PlatformHAL::metric_emit("itl.commit_count", (float)queue_count_);
    return entry.entry_id;
}

// Computes the Merkle root hash for a given batch of entry IDs
Hash256 ITLManager::compute_merkle_root(const Hash256* ids, size_t count) const {
    if (count == 0) return Hash256::null_hash();
    if (count == 1) return ids[0];

    // Merkle tree construction: Pair up hashes and hash them together recursively.
    std::vector<Hash256> current_level(ids, ids + count);

    while (current_level.size() > 1) {
        std::vector<Hash256> next_level;
        for (size_t i = 0; i < current_level.size(); i += 2) {
            Hash256 left = current_level[i];
            // If odd number, the last node is hashed with itself
            Hash256 right = (i + 1 < current_level.size()) ? current_level[i + 1] : left;

            // Concatenate the two hashes and hash the result
            std::array<uint8_t, 64> combined_data;
            std::memcpy(combined_data.data(), left.data, 32);
            std::memcpy(combined_data.data() + 32, right.data, 32);
```

```
            next_level.push_back(PlatformHAL::sha256(combined_data.data(), 64));
        }
        current_level = std::move(next_level);
    }

    return current_level[0];
}

// Creates an ITL entry for the Merkle root and writes it to flash
void ITLManager::anchor_merkle_root(const Hash256& root) {
    // 1. Create the ITL entry for the anchor
    ITLEntry anchor_entry;
    anchor_entry.type = ITLEntry::Type::MERKLE_ANCHOR;
    anchor_entry.timestamp_ms = PlatformHAL::now_ms();
    anchor_entry.payload.merkle_anchor.merkle_root = root;
    anchor_entry.payload_len = sizeof(ITLEntry::MerkleAnchorPayload);

    // Hash the anchor entry itself (simplified hash generation for ITL entry)
    std::array<uint8_t, sizeof(anchor_entry.type) + sizeof(anchor_entry.timestamp_ms) +
sizeof(ITLEntry::MerkleAnchorPayload)> hash_input_buffer;
    size_t offset = 0;
    std::memcpy(hash_input_buffer.data() + offset, &anchor_entry.type,
sizeof(anchor_entry.type)); offset += sizeof(anchor_entry.type);
    std::memcpy(hash_input_buffer.data() + offset, &anchor_entry.timestamp_ms,
sizeof(anchor_entry.timestamp_ms)); offset += sizeof(anchor_entry.timestamp_ms);
    std::memcpy(hash_input_buffer.data() + offset, &anchor_entry.payload,
anchor_entry.payload_len); offset += anchor_entry.payload_len;
    anchor_entry.entry_id = PlatformHAL::sha256(hash_input_buffer.data(), offset);

    // 2. Write the anchor entry to flash
    if (PlatformHAL::flash_write(flash_write_cursor_, &anchor_entry, sizeof(ITLEntry))) {
        flash_write_cursor_ += sizeof(ITLEntry);
        PlatformHAL::downlink_queue(&anchor_entry, sizeof(ITLEntry));
        PlatformHAL::metric_emit("itl.merkle_anchored", 1.0f);
    } else {
        PlatformHAL::metric_emit("itl.flash_write_fail", 1.0f);
    }
}

// Merkle batch processing (called by flush_pending when batch is full)
```

```
void ITLManager::process_merkle_batch() {
    if (merkle_count_ == 0) return;

    // 1. Compute Merkle Root
    Hash256 root = compute_merkle_root(merkle_buffer_, merkle_count_);
    PlatformHAL::metric_emit("itl.merkle_root_computed", 1.0f);

    // 2. Anchor the root in the ITL flash
    anchor_merkle_root(root);

    // 3. Reset the Merkle buffer
    merkle_count_ = 0;
}

// Background processing (call from low-priority task)
void ITLManager::flush_pending() {
    // CRITICAL: This section needs a mutex protecting queue access.
    while (queue_count_ > 0) {
        // Read the entry from the head
        ITLEntry entry_to_process = queue_[queue_head_];

        // 1. Write the entry to flash
        if (PlatformHAL::flash_write(flash_write_cursor_, &entry_to_process, sizeof(ITLEntry))) {
            flash_write_cursor_ += sizeof(ITLEntry);

            // 2. Add entry ID to the current Merkle batch
            if (merkle_count_ < RAPSConfig::MERKLE_BATCH_SIZE) {
                merkle_buffer_[merkle_count_] = entry_to_process.entry_id;
                merkle_count_++;
            } else {
                // Merkle buffer is full, process the batch immediately before adding the new item
                process_merkle_batch();
                merkle_buffer_[merkle_count_] = entry_to_process.entry_id;
                merkle_count_++;
            }

            // 3. Downlink queue the entry
            PlatformHAL::downlink_queue(&entry_to_process, sizeof(ITLEntry));

            // 4. Update queue indices (consume the entry)
```

```
            queue_head_ = (queue_head_ + 1) % RAPSConfig::ITL_QUEUE_SIZE;
            queue_count_--;

        } else {
            // Flash write failed. Stop processing and retry later.
            PlatformHAL::metric_emit("itl.flash_write_stop", 1.0f);
            break;
        }
    }
    // Check if a partial batch remains and flush if needed (optional optimization)
    // For simplicity, we only flush when full or explicitly triggered externally.
}

// Digital Twin Prediction Engine Interface

#include "PDTEngine.hpp"
#include "PlatformHAL.hpp"
#include <cmath>
#include <numeric>
#include <iostream>

void PDTEngine::init() {
    core_physics_model_.init();
    // Initialize the snapshot to a safe, default state (e.g., pre-launch state)
    current_snapshot_.position_m = {RAPSConfig::R_EARTH_M, 0.0f, 0.0f};
    current_snapshot_.velocity_m_s = {0.0f, 0.0f, 0.0f};
    current_snapshot_.attitude_q = {1.0f, 0.0f, 0.0f, 0.0f}; // Identity quaternion
    current_snapshot_.mass_kg = 250000.0f; // Mock launch mass
    current_snapshot_.timestamp_ms = 0;
}

// Updates the internal state snapshot from current sensor readings
void PDTEngine::update_state_snapshot(const PhysicsState& new_state) {
    current_snapshot_ = new_state;

    // Commit state snapshot to ITL (mock ITL commit)
    // NOTE: In RAPSController, this will be handled after validation.
}

// Generates a mock Policy (HLV command) that aims for a nominal state.
```

```
PhysicsControlInput PDTEngine::generate_nominal_control(const PhysicsState& current_state)
const {
    PhysicsControlInput nominal_input;

    // Simple nominal: Apply maximum thrust and zero gimbal
    nominal_input.thrust_magnitude_kN = PropulsionPhysicsEngine::MAX_THRUST_kN;
    nominal_input.gimbal_theta_rad = 0.0f;
    nominal_input.gimbal_phi_rad = 0.0f;
    nominal_input.propellant_flow_kg_s = 100.0f; // Mock high flow rate
    nominal_input.simulation_duration_ms = RAPSConfig::DECISION_HORIZON_MS;

    // In a real system, this would come from a Guidance, Navigation, and Control (GNC) loop
    // that calculates the required thrust vector to meet trajectory targets.

    return nominal_input;
}


// Runs the prediction simulation over the DECISION_HORIZON_MS.
PredictionResult PDTEngine::predict(const PhysicsControlInput& control_input) const {
    // 1. Run the core physics model prediction
    PhysicsState end_state = core_physics_model_.predict_state(current_snapshot_,
control_input);

    // 2. Mock uncertainty and confidence based on deviation from a perfect path (simplified)
    float current_radius = std::sqrt(std::inner_product(current_snapshot_.position_m.begin(),
current_snapshot_.position_m.end(), current_snapshot_.position_m.begin(), 0.0f));
    float end_radius = std::sqrt(std::inner_product(end_state.position_m.begin(),
end_state.position_m.end(), end_state.position_m.begin(), 0.0f));

    // Deviation is proportional to how much the altitude changed unexpectedly fast
    float altitude_diff = end_radius - current_radius;
    float expected_change = 1000.0f; // Expected altitude change in 300ms (mock value)
    float deviation_factor = std::fabs(altitude_diff - expected_change) / expected_change;

    // Confidence decreases with high deviation
    float confidence = std::max(0.5f, 1.0f - deviation_factor * 0.5f);
    float uncertainty = 1.0f - confidence;

    PredictionResult result;
```

```
    result.predicted_end_state = end_state;
    result.confidence = confidence;
    result.uncertainty = uncertainty;
    result.timestamp_ms = PlatformHAL::now_ms();

    // 3. Determine status
    if (end_radius < RAPSConfig::R_EARTH_M) { // Simulated Earth impact
        result.status = PredictionResult::Status::PREDICTED_ESE;
    } else {
        result.status = PredictionResult::Status::NOMINAL;
    }

    // 4. Hash the result for integrity
    // Hashing: Status + Confidence + End State (simplified hash)
    size_t hash_len = sizeof(result.status) + sizeof(result.confidence) + sizeof(PhysicsState);
    std::vector<uint8_t> hash_input(hash_len);
    std::memcpy(hash_input.data(), &result.status, sizeof(result.status));
    std::memcpy(hash_input.data() + sizeof(result.status), &result.confidence,
sizeof(result.confidence));
    std::memcpy(hash_input.data() + sizeof(result.status) + sizeof(result.confidence), &end_state,
sizeof(PhysicsState));
    result.prediction_id = PlatformHAL::sha256(hash_input.data(), hash_len);

    PlatformHAL::metric_emit("pdt.confidence", result.confidence);
    PlatformHAL::metric_emit("pdt.uncertainty", result.uncertainty);

    return result;
}

// Autonomous Policy Engine Interface

#ifndef APE_ENGINE_HPP
#define APE_ENGINE_HPP

#include "RAPSDefinitions.hpp"
#include "PropulsionPhysicsEngine.hpp"
#include <vector>
```

```
//
================================================================================
========
// Autonomous Policy Engine (APE)
//
================================================================================
========
// The APE generates optimized control policies based on mission objectives.
// It is the 'AI' layer responsible for proposing actions.
class APEEngine {
private:
    // This could house a pre-trained reinforcement learning model or complex optimization
routines
public:
    void init();

    // Generates a set of potential policies (thrust/gimbal commands) to meet mission goals
    // Policy generation must be constrained by operational limits.
    std::vector<Policy> generate_policies(const PhysicsState& current_state);

    // Helper to compute the cost of a policy (e.g., fuel usage, time to target, risk)
    float compute_policy_cost(const Policy& policy, const PhysicsState& current_state) const;

    // Helper to select the 'best' policy from a list based on lowest cost
    std::optional<Policy> select_best_policy(const std::vector<Policy>& candidates) const;
};

#endif // APE_ENGINE_HPP

// Safety Monitor Interface

#ifndef SAFETY_MONITOR_HPP
#define SAFETY_MONITOR_HPP

#include "RAPSDefinitions.hpp"
#include "PropulsionPhysicsEngine.hpp"
#include <optional>
```

```
//
================================================================================
========
// Safety Monitor (AILEE - Autonomous Integrity and Execution Limiting Engine)
//
================================================================================
========
// The SafetyMonitor validates policies and predictions against hard flight rules (safety
envelope).
class SafetyMonitor {
private:
    PropulsionPhysicsEngine physics_engine_; // Used to run micro-simulations
    PDTEngine pdt_engine_; // Dependency for current state/reference

    // A simple rollback store (for demonstration)
    std::array<RollbackPlan, RAPSConfig::MAX_ROLLBACK_STORE> rollback_store_;
    size_t rollback_count_ = 0;

    // Checks the predicted state against hard safety bounds (e.g., orbital decay, G-limits)
    bool check_safety_bounds(const PhysicsState& state) const;

public:
    void init(const PDTEngine& pdt);

    // Core AILEE logic: Validates a policy by simulating it against the PDT's current state
    AileeDataPayload validate_policy(const Policy& policy) const;

    // Attempts a 'grace period' re-evaluation if the initial confidence is borderline
    AileeStatus run_grace_period(const Policy& policy, const AileeDataPayload&
initial_payload) const;

    // Checks for execution integrity during active command phase
    bool monitor_execution_integrity(const Policy& executed_policy, const PhysicsState&
current_state) const;

    // Rollback management
    void commit_rollback_plan(const Policy& policy, const Policy& safe_fallback_policy);
    std::optional<RollbackPlan> get_last_safe_rollback() const;
};
```

```
#endif // SAFETY_MONITOR_HPP

// Safety Monitor Implementation

#include "SafetyMonitor.hpp"
#include "PlatformHAL.hpp"
#include <cmath>
#include <algorithm>

void SafetyMonitor::init(const PDTEngine& pdt) {
    physics_engine_.init();
    pdt_engine_ = pdt; // Copy the PDT instance or hold a reference (using copy for simple demo)
    rollback_count_ = 0;
    // Pre-populate one safe fallback for the Redundant Supervisor to use immediately
    Policy fallback_policy;
    std::snprintf(fallback_policy.id, sizeof(fallback_policy.id), "SAFE_FALLBACK_P");
    fallback_policy.thrust_magnitude_kN = 0.0f; // Safe state: Shut down
    fallback_policy.gimbal_theta_rad = 0.0f;
    fallback_policy.gimbal_phi_rad = 0.0f;

    // Commit the default safe fallback plan
    commit_rollback_plan(fallback_policy, fallback_policy);
}

// Checks the predicted state against hard safety bounds
bool SafetyMonitor::check_safety_bounds(const PhysicsState& state) const {
    // HLV Safety Bounds:

    // 1. Structural Mass Limit
    if (state.mass_kg < PropulsionPhysicsEngine::MIN_MASS_KG) {
        PlatformHAL::metric_emit("safety.mass_fail", state.mass_kg);
        return false;
    }

    // 2. Trajectory Bounds (too close to Earth is bad)
    float radius = std::sqrt(state.position_m[0]*state.position_m[0] +
                             state.position_m[1]*state.position_m[1] +
                             state.position_m[2]*state.position_m[2]);
    // Safety Margin: Must be above 95% of Earth's radius
    if (radius < RAPSConfig::R_EARTH_M * 0.95f) {
```

```
        PlatformHAL::metric_emit("safety.trajectory_fail", radius);
        return false;
    }

    // 3. Simple Velocity Limit (e.g., must be accelerating or maintaining orbit)
    // This check is highly context-dependent, but we use a mock check:
    float velocity_mag = std::sqrt(state.velocity_m_s[0]*state.velocity_m_s[0] +
                                   state.velocity_m_s[1]*state.velocity_m_s[1] +
                                   state.velocity_m_s[2]*state.velocity_m_s[2]);
    if (velocity_mag < 0.0f && state.mass_kg > 100000.0f) { // Cannot be negative at high mass
        PlatformHAL::metric_emit("safety.velocity_fail", velocity_mag);
        return false;
    }

    return true;
}


// Core AILEE logic: Validates a policy
AileeDataPayload SafetyMonitor::validate_policy(const Policy& policy) const {
    const PhysicsState& current_state = pdt_engine_.get_current_state();

    // Convert Policy to Control Input (assuming constant flow rate for the horizon)
    PhysicsControlInput input;
    input.thrust_magnitude_kN = policy.thrust_magnitude_kN;
    input.gimbal_theta_rad = policy.gimbal_theta_rad;
    input.gimbal_phi_rad = policy.gimbal_phi_rad;
    input.propellant_flow_kg_s = 100.0f; // Mock flow rate
    input.simulation_duration_ms = RAPSConfig::DECISION_HORIZON_MS;

    // 1. Predict the outcome of the policy
    PhysicsState end_state = physics_engine_.predict_state(current_state, input);

    // 2. Check hard safety limits on the predicted state
    PredictionResult mock_pred = pdt_engine_.predict(input); // Use PDT mock confidence
    AileeDataPayload payload = {mock_pred, policy, mock_pred.confidence};

    if (!check_safety_bounds(end_state)) {
        payload.current_raw_confidence = 0.0f; // Immediately fail confidence
        PlatformHAL::metric_emit("safety.policy_rejected", 0.0f, "reason", "safety_violation");
```

```
        return payload;
    }

    // 3. Assign AILEE status based on confidence
    if (payload.current_raw_confidence >= RAPSConfig::AILEE_CONFIDENCE_ACCEPTED)
{
        PlatformHAL::metric_emit("ailee.status", 1.0f, "status", "ACCEPTED");
        // Status will be set in RAPSController based on the final decision flow
    } else if (payload.current_raw_confidence >=
RAPSConfig::AILEE_CONFIDENCE_BORDERLINE) {
        PlatformHAL::metric_emit("ailee.status", 2.0f, "status", "BORDERLINE");
        // Status will be set in RAPSController based on the final decision flow
    } else {
        PlatformHAL::metric_emit("ailee.status", 3.0f, "status", "OUTRIGHT_REJECTED");
    }

    return payload;
}

// Attempts a 'grace period' re-evaluation if the initial confidence is borderline
AileeStatus SafetyMonitor::run_grace_period(const Policy& policy, const AileeDataPayload&
initial_payload) const {
    if (initial_payload.current_raw_confidence >=
RAPSConfig::AILEE_GRACE_THRESHOLD) {
        // Mock a successful grace period re-validation (e.g., using a second, simpler model)
        PlatformHAL::metric_emit("ailee.grace_pass", 1.0f);
        return AileeStatus::GRACE_PASS;
    } else {
        PlatformHAL::metric_emit("ailee.grace_fail", 1.0f);
        return AileeStatus::GRACE_FAIL;
    }
}

// Checks for execution integrity during active command phase
bool SafetyMonitor::monitor_execution_integrity(const Policy& executed_policy, const
PhysicsState& current_state) const {
    // 1. Check current state against safety bounds
    if (!physics_engine_.is_state_physically_plausible(current_state)) {
        PlatformHAL::metric_emit("safety.realtime_violation", 1.0f);
        return false;
```

```
    }

    // 2. Check if current state deviates too much from the expected path
    // This would require referencing the prediction result for the *current* execution window.
    // Simplified: Check if mass is rapidly dropping (unexpected engine shutdown)
    if (current_state.mass_kg < pdt_engine_.get_current_state().mass_kg * 0.99f &&
executed_policy.thrust_magnitude_kN > 0.0f) {
        // If mass is dropping unexpectedly fast while thrust is requested, something is wrong.
        PlatformHAL::metric_emit("safety.mass_anomaly", 1.0f);
        return false;
    }

    return true;
}

// Rollback management
void SafetyMonitor::commit_rollback_plan(const Policy& policy, const Policy&
safe_fallback_policy) {
    if (rollback_count_ >= RAPSConfig::MAX_ROLLBACK_STORE) {
        // Overwrite the oldest one (simple circular buffer or throw error)
        rollback_count_ = 0;
    }

    RollbackPlan plan;
    std::strncpy(plan.policy_id, policy.id, sizeof(plan.policy_id) - 1);
    plan.thrust_magnitude_kN = safe_fallback_policy.thrust_magnitude_kN;
    plan.gimbal_theta_rad = safe_fallback_policy.gimbal_theta_rad;
    plan.gimbal_phi_rad = safe_fallback_policy.gimbal_phi_rad;
    plan.valid = true;

    // Hash the rollback command set
    float rollback_data[] = {plan.thrust_magnitude_kN, plan.gimbal_theta_rad,
plan.gimbal_phi_rad};
    plan.rollback_hash = PlatformHAL::sha256(rollback_data, sizeof(rollback_data));

    rollback_store_[rollback_count_++] = plan;
}

std::optional<RollbackPlan> SafetyMonitor::get_last_safe_rollback() const {
    if (rollback_count_ > 0) {
```

```
        return rollback_store_[rollback_count_ - 1]; // Return the most recent valid one
    }
    return std::nullopt;
}

// RAPS CONTROLLER INTERFACE

#ifndef RAPS_CONTROLLER_HPP
#define RAPS_CONTROLLER_HPP

#include "RAPSDefinitions.hpp"
#include "ITLManager.hpp"
#include "PDTEngine.hpp"
#include "APEEngine.hpp"
#include "SafetyMonitor.hpp"

// Forward declaration of the Supervisor (for redundancy management)
class RedundantSupervisor;

//
================================================================================
========
// RAPS Controller (Main Governance Orchestrator)
//
================================================================================
========
// Manages the overall decision cycle: Sense -> Predict -> Plan -> Act -> Audit.
class RAPSController {
private:
    ITLManager itl_manager_;
    PDTEngine pdt_engine_;
    APEEngine ape_engine_;
    SafetyMonitor safety_monitor_;

    // Reference to the supervising authority (e.g., the A/B redundancy manager)
    RedundantSupervisor* supervisor_ = nullptr;

    // Internal state tracking
    bool is_thrusting_ = false;
    uint32_t last_command_timestamp_ = 0;
```

```
    // Private Orchestration Steps
    void step_sense_and_audit(const PhysicsState& current_state);
    void step_predict_and_plan();
    void step_validate_and_execute(const Policy& policy_to_execute);
    void step_execute_command(const Policy& policy);

public:
    void init(RedundantSupervisor* supervisor);

    // Main control loop iteration (called by the RTOS thread/task)
    void run_cycle(const PhysicsState& current_state);

    // Fallback handler (called by RedundantSupervisor on primary failure)
    void trigger_fallback(const char* reason);
};

#endif // RAPS_CONTROLLER_HPP

// RAPS CONTROLLER IMPLEMENTATION

#include "RAPSController.hpp"
#include "PlatformHAL.hpp"
#include "RedundantSupervisor.hpp"
#include <iostream>

void RAPSController::init(RedundantSupervisor* supervisor) {
    supervisor_ = supervisor;
    itl_manager_.init();
    pdt_engine_.init();
    ape_engine_.init();
    // Safety Monitor needs PDT for its initial state reference
    safety_monitor_.init(pdt_engine_);

    // Initial state snapshot (mock initial sensor reading)
    PhysicsState initial_state = pdt_engine_.get_current_state();
    step_sense_and_audit(initial_state);
}

// Main control loop iteration
```

```
void RAPSController::run_cycle(const PhysicsState& current_state) {
    uint32_t start_time = PlatformHAL::now_ms();

    // 1. SENSE & AUDIT (Update state, check execution integrity)
    step_sense_and_audit(current_state);

    // 2. PREDICT & PLAN (Generate control policy)
    step_predict_and_plan();

    // 3. AUDIT: Flush ITL queue in background
    itl_manager_.flush_pending();

    uint32_t elapsed_time = PlatformHAL::now_ms() - start_time;
    PlatformHAL::metric_emit("raps.cycle_time_ms", (float)elapsed_time);

    if (elapsed_time > RAPSConfig::WATCHDOG_MS) {
        // Budget violation is a critical event
        PlatformHAL::metric_emit("raps.budget_violation", (float)elapsed_time);
        ITLEntry budget_entry;
        budget_entry.type = ITLEntry::Type::GOVERNANCE_BUDGET_VIOLATION;
        budget_entry.timestamp_ms = start_time;
        budget_entry.payload.governance_budget_violation.elapsed_ms = elapsed_time;
        itl_manager_.commit(budget_entry);
    }
}

void RAPSController::step_sense_and_audit(const PhysicsState& current_state) {
    // 1. Update Digital Twin with current sensor/estimator state
    pdt_engine_.update_state_snapshot(current_state);

    // 2. Log the new state snapshot to the ITL
    ITLEntry state_entry;
    state_entry.type = ITLEntry::Type::STATE_SNAPSHOT;
    state_entry.timestamp_ms = PlatformHAL::now_ms();
    state_entry.payload.state_snapshot.current_state = current_state;
    // The hash of the state is complex, for simplicity, we rely on the state being inside the
payload.
    // In a real system, the snapshot hash would be calculated securely.
    itl_manager_.commit(state_entry);
```

```
    // 3. Real-time execution integrity monitoring (if a command is active)
    if (is_thrusting_ && !safety_monitor_.monitor_execution_integrity(Policy{}, current_state)) {
        // Critical: Command execution is going off rails!
        trigger_fallback("Execution Integrity Failed");
    }
}

void RAPSController::step_predict_and_plan() {
    // 1. Get current state from PDT
    const PhysicsState& current_state = pdt_engine_.get_current_state();

    // 2. Generate optimal policies
    std::vector<Policy> candidates = ape_engine_.generate_policies(current_state);

    // 3. Select the best policy candidate
    std::optional<Policy> best_policy = ape_engine_.select_best_policy(candidates);

    if (!best_policy) {
        trigger_fallback("No Policy Generated");
        return;
    }

    // 4. Run Policy Validation and Execution Flow
    step_validate_and_execute(*best_policy);
}

void RAPSController::step_validate_and_execute(const Policy& policy_to_execute) {
    // 1. Initial AILEE Validation
    AileeDataPayload validation_result = safety_monitor_.validate_policy(policy_to_execute);

    AileeStatus final_status;

    if (validation_result.current_raw_confidence >=
RAPSConfig::AILEE_CONFIDENCE_ACCEPTED) {
        final_status = AileeStatus::ACCEPTED;
    } else if (validation_result.current_raw_confidence >=
RAPSConfig::AILEE_CONFIDENCE_BORDERLINE) {
        // Borderline Confidence: Run Grace Period
        final_status = safety_monitor_.run_grace_period(policy_to_execute, validation_result);
    } else {
```

```
        // Outright Rejected
        final_status = AileeStatus::OUTRIGHT_REJECTED;
    }

    // 2. Commit AILEE status to ITL
    ITLEntry ailee_entry;
    ailee_entry.type = ITLEntry::Type::AILEE_SAFETY_STATUS;
    ailee_entry.timestamp_ms = PlatformHAL::now_ms();
    ailee_entry.payload.ailee_safety_status.status = final_status;
    ailee_entry.payload.ailee_safety_status.confidence_at_decision =
validation_result.current_raw_confidence;
    itl_manager_.commit(ailee_entry);

    // 3. Decision Tree
    if (final_status == AileeStatus::ACCEPTED || final_status == AileeStatus::GRACE_PASS ||
final_status == AileeStatus::CONSENSUS_PASS) {
        // Policy is safe and approved: Execute!
        step_execute_command(policy_to_execute);
    } else {
        // Policy is rejected: Fallback is required
        trigger_fallback("Policy Rejected by AILEE");
    }
}

void RAPSController::step_execute_command(const Policy& policy) {
    std::string tx_id = PlatformHAL::generate_tx_id();

    // 1. Commit Command Pending to ITL
    ITLEntry pending_entry;
    pending_entry.type = ITLEntry::Type::COMMAND_PENDING;
    pending_entry.timestamp_ms = PlatformHAL::now_ms();
    std::strncpy(pending_entry.payload.command_execution.tx_id, tx_id.c_str(),
sizeof(pending_entry.payload.command_execution.tx_id) - 1);
    itl_manager_.commit(pending_entry);

    // 2. Execute command via HAL (using Gimbal Theta as the single simplified valve/gimbal
actuator in the stub)
    uint32_t timeout = RAPSConfig::WATCHDOG_MS / 2; // Half the cycle time for execution
    bool success = PlatformHAL::actuator_execute(
        tx_id.c_str(),
```

```
        policy.thrust_magnitude_kN,
        policy.gimbal_theta_rad, // Using one gimbal angle for actuator stub
        timeout);

    // 3. Handle result
    if (success) {
        is_thrusting_ = (policy.thrust_magnitude_kN > 0.0f);
        last_command_timestamp_ = PlatformHAL::now_ms();

        // Command Commit to ITL
        ITLEntry commit_entry;
        commit_entry.type = ITLEntry::Type::COMMAND_COMMIT;
        commit_entry.timestamp_ms = last_command_timestamp_;
        std::strncpy(commit_entry.payload.command_execution.tx_id, tx_id.c_str(),
sizeof(commit_entry.payload.command_execution.tx_id) - 1);
        itl_manager_.commit(commit_entry);

        // Update rollback store with the executed policy (for next cycle's safe abort)
        Policy safe_abort;
        std::snprintf(safe_abort.id, sizeof(safe_abort.id), "ABORT_%s", policy.id);
        safe_abort.thrust_magnitude_kN = 0.0f; // Immediate engine shutdown is the safest default
abort
        safety_monitor_.commit_rollback_plan(policy, safe_abort);

        PlatformHAL::metric_emit("raps.command_executed", 1.0f);
    } else {
        // Execution Failure
        PlatformHAL::metric_emit("raps.execution_failure", 1.0f);
        ITLEntry failure_entry;
        failure_entry.type = ITLEntry::Type::EXECUTION_FAILURE;
        failure_entry.timestamp_ms = PlatformHAL::now_ms();
        std::strncpy(failure_entry.payload.command_execution.tx_id, tx_id.c_str(),
sizeof(failure_entry.payload.command_execution.tx_id) - 1);
        failure_entry.payload.command_execution.elapsed_ms = PlatformHAL::now_ms() -
pending_entry.timestamp_ms;
        itl_manager_.commit(failure_entry);

        // On execution failure, transition to fallback
        trigger_fallback("Actuator Execution Timeout/Failure");
    }
```

```
}

// Fallback handler (called by RedundantSupervisor or internally)
void RAPSController::trigger_fallback(const char* reason) {
    PlatformHAL::metric_emit("raps.fallback_triggered", 1.0f, "reason", reason);

    // 1. Log fallback trigger to ITL
    ITLEntry fallback_entry;
    fallback_entry.type = ITLEntry::Type::FALLBACK_TRIGGERED;
    fallback_entry.timestamp_ms = PlatformHAL::now_ms();
    std::strncpy(fallback_entry.payload.fallback_triggered.reason, reason,
sizeof(fallback_entry.payload.fallback_triggered.reason) - 1);
    itl_manager_.commit(fallback_entry);

    // 2. Retrieve last safe rollback plan
    std::optional<RollbackPlan> rollback = safety_monitor_.get_last_safe_rollback();

    if (rollback) {
        // 3. Execute Rollback Command (e.g., Immediate Engine Shutdown)
        std::string tx_id = PlatformHAL::generate_tx_id();

        bool success = PlatformHAL::actuator_execute(
            tx_id.c_str(),
            rollback->thrust_magnitude_kN,
            rollback->gimbal_theta_rad, // Using one gimbal angle for stub
            RAPSConfig::WATCHDOG_MS / 4); // Quick, aggressive timeout

        if (success) {
            is_thrusting_ = false;
            // Rollback Commit to ITL
            ITLEntry commit_entry;
            commit_entry.type = ITLEntry::Type::ROLLBACK_COMMIT;
            commit_entry.timestamp_ms = PlatformHAL::now_ms();
            std::strncpy(commit_entry.payload.command_execution.tx_id, tx_id.c_str(),
sizeof(commit_entry.payload.command_execution.tx_id) - 1);
            itl_manager_.commit(commit_entry);
            PlatformHAL::metric_emit("raps.rollback_success", 1.0f);
        } else {
            // Catastrophic failure: Rollback failed. Engage higher-level system failure.
            PlatformHAL::metric_emit("raps.critical_failure", 1.0f, "reason", "RollbackFailed");
```

```
            // Notify supervisor to potentially switch to the other channel
            if (supervisor_) {

supervisor_->notify_failure(RedundantSupervisor::FailureMode::CRITICAL_ROLLBACK_FAI
L);
            }
        }
    } else {
        // No valid rollback plan, fail safe
        PlatformHAL::metric_emit("raps.no_rollback_plan", 1.0f);
        if (supervisor_) {

supervisor_->notify_failure(RedundantSupervisor::FailureMode::CRITICAL_NO_ROLLBACK
);
        }
    }
}

// Redundant Supervisor Interface

#ifndef REDUNDANT_SUPERVISOR_HPP
#define REDUNDANT_SUPERVISOR_HPP

#include "RAPSController.hpp"
#include "RAPSDefinitions.hpp"

//
================================================================================
========
// Redundant Supervisor
//
================================================================================
========
// Manages the A/B redundancy channels (RAPSController A and RAPSController B).
// Determines which controller is the active 'governor'.
class RedundantSupervisor {
public:
    enum class FailureMode {
        CRITICAL_ROLLBACK_FAIL,
        CRITICAL_NO_ROLLBACK,
```

```
        PRIMARY_CHANNEL_LOCKUP,
        MISMATED_PREDICTION
    };

private:
    RAPSController controller_a_;
    RAPSController controller_b_;

    // State of redundancy
    bool is_channel_a_active_ = true;
    uint32_t last_sync_timestamp_ = 0;

    // For MISMATED_PREDICTION: Store the last good prediction from the active channel
    PredictionResult last_active_prediction_;

    // Internal actions
    void switch_to_channel_b();
    void switch_to_channel_a();
    void log_supervisor_exception(FailureMode mode);

public:
    void init();

    // Main loop for the Supervisor (runs the active controller cycle)
    void run_cycle(const PhysicsState& current_state);

    // Called by the active RAPSController on catastrophic failure
    void notify_failure(FailureMode mode);

    // Synchronizes the inactive channel's state with the active one
    void synchronize_inactive_controller(const PhysicsState& current_state);

    // Checks for consistency between A/B channels (if both are running predictions)
    bool check_a_b_prediction_mismatch(const PredictionResult& result_a, const
PredictionResult& result_b) const;
};

#endif // REDUNDANT_SUPERVISOR_HPP

// Redundant Supervisor Implementation
```

```
#include "RedundantSupervisor.hpp"
#include "PlatformHAL.hpp"
#include <iostream>

void RedundantSupervisor::init() {
    controller_a_.init(this); // Pass self reference for failure notification
    controller_b_.init(this);

    // Initial sync
    const PhysicsState& initial_state = controller_a_.get_current_state(); // Assume A holds initial
state
    synchronize_inactive_controller(initial_state);
    last_sync_timestamp_ = PlatformHAL::now_ms();

    PlatformHAL::metric_emit("supervisor.active_channel", is_channel_a_active_ ? 0.0f : 1.0f,
"channel", is_channel_a_active_ ? "A" : "B");
}

void RedundantSupervisor::run_cycle(const PhysicsState& current_state) {
    // 1. Run the Active Controller
    if (is_channel_a_active_) {
        controller_a_.run_cycle(current_state);
        // last_active_prediction_ = controller_a_.get_last_prediction(); // Needs implementation in
RAPSController
    } else {
        controller_b_.run_cycle(current_state);
        // last_active_prediction_ = controller_b_.get_last_prediction();
    }

    // 2. Periodically synchronize the inactive controller
    if (PlatformHAL::now_ms() - last_sync_timestamp_ > 1000) { // Sync every 1s
        synchronize_inactive_controller(current_state);
        last_sync_timestamp_ = PlatformHAL::now_ms();
    }

    // NOTE: For full dual-mode redundancy, both A and B would run prediction,
    // and the Supervisor would compare them using `check_a_b_prediction_mismatch`
}
```

```
void RedundantSupervisor::synchronize_inactive_controller(const PhysicsState& current_state)
{
    // In a real system, this involves reading the full state, ITL cursor, and configuration
    // from the active channel's memory/flash and writing it to the inactive channel.
    if (is_channel_a_active_) {
        controller_b_.update_state_snapshot(current_state);
    } else {
        controller_a_.update_state_snapshot(current_state);
    }
    PlatformHAL::metric_emit("supervisor.sync_complete", 1.0f);
}

// Checks for consistency between A/B channels
bool RedundantSupervisor::check_a_b_prediction_mismatch(const PredictionResult& result_a,
const PredictionResult& result_b) const {
    // Simple check: Mismatch if the predicted end position differs by more than 100m
    float pos_a = result_a.predicted_end_state.position_m[0] +
result_a.predicted_end_state.position_m[1] + result_a.predicted_end_state.position_m[2];
    float pos_b = result_b.predicted_end_state.position_m[0] +
result_b.predicted_end_state.position_m[1] + result_b.predicted_end_state.position_m[2];

    if (std::fabs(pos_a - pos_b) > RAPSConfig::ACCEPT_POSITION_DEV_M) {
        log_supervisor_exception(FailureMode::MISMATED_PREDICTION);
        return true;
    }
    return false;
}

void RedundantSupervisor::log_supervisor_exception(FailureMode mode) {
    const char* reason = "Unknown";
    switch(mode) {
        case FailureMode::CRITICAL_ROLLBACK_FAIL: reason =
"CRITICAL_ROLLBACK_FAIL"; break;
        case FailureMode::CRITICAL_NO_ROLLBACK: reason =
"CRITICAL_NO_ROLLBACK"; break;
        case FailureMode::PRIMARY_CHANNEL_LOCKUP: reason =
"PRIMARY_CHANNEL_LOCKUP"; break;
        case FailureMode::MISMATED_PREDICTION: reason = "MISMATED_PREDICTION";
break;
    }
```

```
    ITLEntry entry;
    entry.type = ITLEntry::Type::SUPERVISOR_EXCEPTION;
    entry.timestamp_ms = PlatformHAL::now_ms();
    std::strncpy(entry.payload.supervisor_exception.reason, reason,
sizeof(entry.payload.supervisor_exception.reason) - 1);
    // In a real system, the Supervisor would have its own ITL manager
    // For this demo, we can't commit, but we would log the event securely.
    PlatformHAL::metric_emit("supervisor.exception", 1.0f, "mode", reason);
}

void RedundantSupervisor::switch_to_channel_b() {
    is_channel_a_active_ = false;
    // Log switch and perform immediate sync
    synchronize_inactive_controller(is_channel_a_active_ ? controller_a_.get_current_state() :
controller_b_.get_current_state());
    PlatformHAL::metric_emit("supervisor.active_channel", 1.0f, "channel", "B");
    PlatformHAL::metric_emit("supervisor.switch_count", 1.0f);
}

void RedundantSupervisor::switch_to_channel_a() {
    is_channel_a_active_ = true;
    synchronize_inactive_controller(is_channel_a_active_ ? controller_b_.get_current_state() :
controller_a_.get_current_state());
    PlatformHAL::metric_emit("supervisor.active_channel", 0.0f, "channel", "A");
    PlatformHAL::metric_emit("supervisor.switch_count", 1.0f);
}

void RedundantSupervisor::notify_failure(FailureMode mode) {
    log_supervisor_exception(mode);

    // Logic: If the active channel fails critically, switch immediately.
    if (mode == FailureMode::CRITICAL_ROLLBACK_FAIL || mode ==
FailureMode::CRITICAL_NO_ROLLBACK || mode ==
FailureMode::PRIMARY_CHANNEL_LOCKUP) {
        if (is_channel_a_active_) {
            PlatformHAL::metric_emit("supervisor.failover", 1.0f, "from", "A");
            switch_to_channel_b();
            // Trigger the new active channel to execute the fallback immediately
            controller_b_.trigger_fallback("Failover Switch");
```

```
        } else {
            // Both A and B failed? Catastrophic system halt.
            PlatformHAL::metric_emit("supervisor.fatal_system_halt", 1.0f);
        }
    }
}

// Demonstration Entry Point

#include "RedundantSupervisor.hpp"
#include "PlatformHAL.hpp"
#include <iostream>
#include <unistd.h> // For usleep in Linux/macOS, use <windows.h> or RTOS calls for other
targets

// Mock sensor reading function for demonstration purposes
PhysicsState mock_read_sensors(const PhysicsState& last_state) {
    PhysicsState new_state = last_state;
    // Simulate slight drift in state based on the last known state
    new_state.timestamp_ms = PlatformHAL::now_ms();

    // Apply a mock velocity change
    new_state.velocity_m_s[0] += PlatformHAL::random_float(-0.5f, 0.5f);
    new_state.velocity_m_s[1] += PlatformHAL::random_float(-0.5f, 0.5f);
    new_state.velocity_m_s[2] += PlatformHAL::random_float(-0.5f, 0.5f);

    // Apply mock position change based on new velocity
    float dt_s = (new_state.timestamp_ms - last_state.timestamp_ms) / 1000.0f;
    new_state.position_m[0] += new_state.velocity_m_s[0] * dt_s;
    new_state.position_m[1] += new_state.velocity_m_s[1] * dt_s;
    new_state.position_m[2] += new_state.velocity_m_s[2] * dt_s;

    // Simulate mass change (always decreasing)
    new_state.mass_kg -= PlatformHAL::random_float(0.0f, 10.0f); // Mock fuel burn

    return new_state;
}

int main() {
```

```
    std::cout <<
"============================================================\n";
    std::cout << " RAPS Kernel HLV Demonstration (RTOS Concepts)\n";
    std::cout <<
"============================================================\n";

    // Initialize Platform HAL (RNG seed)
    PlatformHAL::seed_rng_for_stubs(static_cast<uint32_t>(std::time(nullptr)));

    // Initialize the Redundant Supervisor (which initializes both RAPS Controllers)
    RedundantSupervisor supervisor;
    supervisor.init();

    // Mock initial state (on the launch pad or in stable orbit reference)
    PhysicsState current_state;
    current_state.position_m = {RAPSConfig::R_EARTH_M, 0.0f, 0.0f};
    current_state.velocity_m_s = {0.0f, 0.0f, 0.0f};
    current_state.attitude_q = {1.0f, 0.0f, 0.0f, 0.0f};
    current_state.mass_kg = 250000.0f;
    current_state.timestamp_ms = PlatformHAL::now_ms();

    // --- Main RTOS Loop Simulation ---
    // The main loop should run faster than the Decision Horizon
    constexpr uint32_t RTOS_CYCLE_MS = 50; // 20Hz loop
    int cycle_count = 0;

    while (cycle_count < 20) { // Run for 20 cycles (1 second in this demo)
        uint32_t cycle_start = PlatformHAL::now_ms();

        // 1. Mock Sensor Reading (Simulate I/O)
        current_state = mock_read_sensors(current_state);

        // 2. Run Supervisor (which runs the active RAPS Controller A/B)
        supervisor.run_cycle(current_state);

        // --- Mock a failure event at cycle 10 to test redundancy/fallback ---
        if (cycle_count == 10) {
            std::cout << "\n[MAIN] *** SIMULATING PRIMARY CHANNEL LOCKUP ***\n";
```

```
supervisor.notify_failure(RedundantSupervisor::FailureMode::PRIMARY_CHANNEL_LOCKU
P);
            std::cout << "[MAIN] *** FAILOVER SHOULD HAVE OCCURRED ***\n\n";
        }

        // 3. Throttle loop to maintain RTOS frequency
        uint32_t cycle_time_ms = PlatformHAL::now_ms() - cycle_start;
        if (cycle_time_ms < RTOS_CYCLE_MS) {
            // Use usleep to wait (in a real RTOS, this would be a task_delay or yield)
            usleep((RTOS_CYCLE_MS - cycle_time_ms) * 1000);
        }

        std::cout << "[MAIN] Cycle " << cycle_count++
                  << " | Radius: " << (std::sqrt(current_state.position_m[0]*current_state.position_m[0]
+ current_state.position_m[1]*current_state.position_m[1] +
current_state.position_m[2]*current_state.position_m[2]) - RAPSConfig::R_EARTH_M) /
1000.0f
                  << " km | Mass: " << current_state.mass_kg << " kg\n";
    }

    std::cout <<
"\n============================================================\n";
    std::cout << "Demo Complete. Check metrics for execution trace.\n";
    return 0;
}

// APCU Header with HLV Math Constants

#pragma once

#include "PlatformHAL.hpp"
#include <array>
#include <cstdint>
#include <cstring>

// --- Configuration Structs (Assuming RAPSConfig is defined elsewhere) ---
// For the purpose of this file, we assume RAPSConfig is available.
namespace RAPSConfig {
    // Critical Limits
```

```
    static constexpr float CRITICAL_ANTIMATTER_KG = 10.0f;
    static constexpr float EMERGENCY_ANTIMATTER_RESERVE_KG = 50.0f;
    static constexpr float CRITICAL_QUANTUM_FLUID_LITERS = 100.0f;
    static constexpr float EMERGENCY_QUANTUM_FLUID_LITERS = 500.0f;
    static constexpr float CRITICAL_FIELD_COUPLING_THRESHOLD = 0.85f;
    static constexpr float MAX_SUBSPACE_EFFICIENCY = 95.0f;
}

// --- Physical Constants ---
static constexpr float INITIAL_ANTIMATTER_KG = 1000.0f;
static constexpr float INITIAL_QUANTUM_FLUID_LITERS = 10000.0f;
static constexpr float MIN_POWER_DRAW_GW = 0.5f;

// Field Limits
static constexpr float MAX_WARP_FIELD_STRENGTH = 10.0f;        // Unitless, max stable W field
static constexpr float MAX_GRAVITO_FLUX_BIAS = 5.0f;           // Unitless, max stable flux bias
static constexpr float MAX_TIME_DILATION_FACTOR = 50.0f;       // Time factor (max ship time / external time)
static constexpr float MAX_INDUCED_GRAVITY_G = 2.5f;           // Max induced gravity in G's
static constexpr float MAX_SYSTEM_POWER_DRAW_GW = 500.0f;
static constexpr float MAX_SPACETIME_CURVATURE_MAGNITUDE = 100.0f; // Max curvature unit

// --- HLV Propulsion Math Constants ---

// 1. Curvature & Dilation Physics
static constexpr float WARP_CURVATURE_CUBIC_SCALAR = 0.8f;      // C scales with W^3
static constexpr float FLUX_CURVATURE_QUADRATIC_SCALAR = 0.4f;  // C scales with Phi_g^2
static constexpr float CURVATURE_TIME_DILATION_EXPONENT_BASE = 0.05f; // Exponential rate of Dilation from Curvature
static constexpr float FLUID_LEVEL_DAMPING_FACTOR = 0.1f;       // Fluid non-linear dampening

// 2. Gravity Physics
static constexpr float FLUX_GRAVITY_BASE_SCALAR = 0.5f;         // G scales linearly with Phi_g
```

```
static constexpr float WARP_GRAVITY_MODULATION_SCALAR = 0.1f;   // W field modulates G effect

// 3. Coupling Stress & Stability
static constexpr float COUPLING_STRESS_EXPONENT_SCALAR = 0.2f;  // Stress (Sigma) exponential growth rate
static constexpr float STABILITY_STRESS_QUADRATIC_SCALAR = 0.5f; // Stability penalty from Sigma^2
static constexpr float STABILITY_CURVATURE_CUBIC_SCALAR = 0.005f; // Stability penalty from C^3

// 4. Power & Efficiency
static constexpr float POWER_WARP_CUBIC_SCALAR = 0.2f;          // Power scales with W^3
static constexpr float POWER_CURVATURE_QUADRATIC_SCALAR = 0.01f; // Power scales with C^2
static constexpr float POWER_FLUX_LINEAR_SCALAR = 5.0f;         // Power scales linearly with Phi_g
static constexpr float POWER_SLEW_PENALTY_SCALE = 1000.0f;
static constexpr float POWER_SLEW_PENALTY_EXPONENT = 2.0f;      // Quadratic penalty for rapid change

static constexpr float EFFICIENCY_WARP_QUADRATIC_SCALAR = 10.0f; // Base efficiency factor
static constexpr float EFFICIENCY_FLUID_EXPONENT = 0.5f;         // Fluid level efficiency power factor
static constexpr float EFFICIENCY_POWER_PEAK_GW = 250.0f;        // Target power draw for max efficiency
static constexpr float EFFICIENCY_POWER_VARIANCE_GW = 50.0f;     // Variance for Gaussian power curve

// 5. Resource Consumption & Displacement
static constexpr float ANTIMATTER_BURN_RATE_GW_TO_KG_PER_MS = 1.0e-12f; // Very small burn rate
static constexpr float QUANTUM_FLUID_BASE_CONSUMPTION_RATE = 1.0e-5f; // L/ms
static constexpr float QUANTUM_FLUID_CONSUMPTION_CURVATURE_EXPONENT = 1.5f; // Non-linear curvature cost
static constexpr float QUANTUM_FLUID_CONSUMPTION_PER_CURVATURE_UNIT_MS = 5.0e-6f; // L/C/ms
static constexpr float WARP_TO_DISPLACEMENT_FACTOR_KM_PER_S = 1000.0f; // Displacement per unit warp
```

```
// --- PID Control Parameters (Placeholder values used in original .cpp) ---
static constexpr float WARP_KP = 0.5f, WARP_KI = 0.001f, WARP_KD = 0.1f,
WARP_INTEGRAL_LIMIT = 50.0f;
static constexpr float FLUX_KP = 0.8f, FLUX_KI = 0.005f, FLUX_KD = 0.15f,
FLUX_INTEGRAL_LIMIT = 20.0f;
static constexpr float DILATION_KP = 0.2f, DILATION_KI = 0.0005f, DILATION_KD =
0.05f;
static constexpr float GRAVITY_KP = 0.4f, GRAVITY_KI = 0.002f, GRAVITY_KD = 0.1f;

// Control Response Limits
static constexpr float WARP_FIELD_RESPONSE_RATE_PER_MS = 0.01f;
static constexpr float GRAVITO_FLUX_RESPONSE_RATE_PER_MS = 0.01f;
static constexpr float TIME_DILATION_RESPONSE_RATE_PER_MS = 0.005f;
static constexpr float GRAVITY_RESPONSE_RATE_PER_MS = 0.005f;

// Emergency Limits
static constexpr float EMERGENCY_RESPONSE_DAMPING_FACTOR = 0.2f;
static constexpr uint32_t RESONANCE_SAMPLE_COUNT = 50;

// --- Data Structures (from original code context) ---
struct SpacetimeModulationState {
    float power_draw_GW;
    float warp_field_strength;
    float gravito_flux_bias;
    float spacetime_curvature_magnitude;
    float time_dilation_factor;
    float induced_gravity_g;
    float subspace_efficiency_pct;
    double total_displacement_km;
    float remaining_antimatter_kg;
    float quantum_fluid_level;
    float field_coupling_stress;
    float spacetime_stability_index;
    float control_authority_remaining;
    bool emergency_mode_active;
    uint64_t timestamp_ms;
    Hash256 state_hash;
};
```

```
struct SpacetimeModulationCommand {
    float target_warp_field_strength;
    float target_gravito_flux_bias;
    float target_time_dilation_factor;
    float target_artificial_gravity_g;
    float target_quantum_fluid_flow_rate; // L/s
    float target_power_budget_GW;
    bool enable_emergency_damping;
    bool enable_resonance_suppression;
    bool enable_time_dilation_coupling;
};

// --- Control Unit Class ---
class AdvancedPropulsionControlUnit {
public:
    void init();
    void update_internal_state(uint32_t elapsed_ms);
    bool receive_and_execute_spacetime_command(
        const SpacetimeModulationCommand& command,
        const char* directive_id);
    SpacetimeModulationState get_current_state() const;
    bool execute_controlled_shutdown();
    bool initiate_emergency_spacetime_collapse();
    bool restore_from_safe_state(const SpacetimeModulationState& safe_state);

private:
    // State
    SpacetimeModulationState current_propulsion_state_;
    SpacetimeModulationCommand active_spacetime_command_;
    char active_directive_id_[64];
    SpacetimeModulationState last_safe_state_;
    uint64_t last_safe_state_timestamp_ms_;
    bool emergency_mode_active_;

    // PID State
    float warp_error_integral_;
    float warp_error_previous_;
    float flux_error_integral_;
    float flux_error_previous_;
    float dilation_error_integral_;
```

```
    float dilation_error_previous_;
    float gravity_error_integral_;
    float gravity_error_previous_;
    float fluid_error_integral_; // Placeholder for future fluid control
    float fluid_error_previous_;

    // Resonance Detection
    std::array<float, RESONANCE_SAMPLE_COUNT> field_coupling_history_;
    uint32_t coupling_history_index_;

    // Core Physics and Control Logic
    Hash256 calculate_state_hash(const SpacetimeModulationState& state) const;
    float compute_pid_output(float error, float& integral, float& previous_error, float kp, float ki,
float kd, float integral_limit, float elapsed_ms);

    // HLV Math Models (Updated)
    float compute_capability_scale() const;
    float compute_spacetime_curvature() const;
    float compute_derived_time_dilation() const;
    float compute_derived_gravity() const;
    float compute_field_coupling_stress() const;
    float compute_power_draw(float warp_slew, float flux_slew) const;
    float compute_subspace_efficiency(const SpacetimeModulationState& state) const;
    float compute_stability_index() const;
    float compute_control_authority() const;

    void consume_resources(uint32_t elapsed_ms);

    // Safety & Resilience
    bool detect_resonance_instability();
    void apply_resonance_suppression(float& warp_change, float& flux_change);
    bool is_operational_state_safe() const;
    bool is_state_safe_to_save(const SpacetimeModulationState& state) const;
    void save_safe_state();
    void enter_emergency_mode();
    void apply_emergency_limits(SpacetimeModulationCommand& command);
};

// APCU Implementation with HLV Math
```

```
#include "AdvancedPropulsionControlUnit.hpp"
#include <cmath>
#include <algorithm>

// Define the mathematical constant Pi for use in the Gaussian function
static constexpr float PI = 3.14159265358979323846f;

void AdvancedPropulsionControlUnit::init() {
    current_propulsion_state_ = {
        MIN_POWER_DRAW_GW,
        0.0f,  // warp_field_strength
        0.0f,  // gravito_flux_bias
        0.0f,  // spacetime_curvature_magnitude
        1.0f,  // time_dilation_factor
        0.0f,  // induced_gravity_g
        0.0f,  // subspace_efficiency_pct
        0.0f,  // total_displacement_km
        INITIAL_ANTIMATTER_KG,
        INITIAL_QUANTUM_FLUID_LITERS,
        0.0f,  // field_coupling_stress
        1.0f,  // spacetime_stability_index
        1.0f,  // control_authority_remaining
        false, // emergency_mode_active
        PlatformHAL::now_ms(),
        Hash256::null_hash()
    };
    current_propulsion_state_.state_hash = calculate_state_hash(current_propulsion_state_);

    active_spacetime_command_ = {
        0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
        MIN_POWER_DRAW_GW,
        false, false, true  // Conservative defaults
    };
    std::strncpy(active_directive_id_, "INIT_NEUTRAL", sizeof(active_directive_id_) - 1);
    active_directive_id_[sizeof(active_directive_id_) - 1] = '\0';

    // Initialize PID state
    warp_error_integral_ = 0.0f;
    warp_error_previous_ = 0.0f;
    flux_error_integral_ = 0.0f;
```

```
    flux_error_previous_ = 0.0f;
    dilation_error_integral_ = 0.0f;
    dilation_error_previous_ = 0.0f;
    gravity_error_integral_ = 0.0f;
    gravity_error_previous_ = 0.0f;
    fluid_error_integral_ = 0.0f;
    fluid_error_previous_ = 0.0f;

    // Initialize resonance detection
    field_coupling_history_.fill(0.0f);
    coupling_history_index_ = 0;

    emergency_mode_active_ = false;
    last_safe_state_ = current_propulsion_state_;
    last_safe_state_timestamp_ms_ = PlatformHAL::now_ms();

    PlatformHAL::metric_emit("apcu.initialized", 1.0f,
        "antimatter_kg", std::to_string(INITIAL_ANTIMATTER_KG).c_str());
    PlatformHAL::metric_emit("apcu.initialized", 1.0f,
        "quantum_fluid_L", std::to_string(INITIAL_QUANTUM_FLUID_LITERS).c_str());
}

Hash256 AdvancedPropulsionControlUnit::calculate_state_hash(const
SpacetimeModulationState& state) const {
    std::array<float, 10> hash_input = {
        state.power_draw_GW,
        state.warp_field_strength,
        state.gravito_flux_bias,
        state.spacetime_curvature_magnitude,
        state.time_dilation_factor,
        state.induced_gravity_g,
        state.subspace_efficiency_pct,
        (float)state.total_displacement_km, // Cast double to float for hashing consistency
        state.remaining_antimatter_kg,
        state.quantum_fluid_level
    };
    return PlatformHAL::sha256(hash_input.data(), sizeof(hash_input));
}

bool AdvancedPropulsionControlUnit::receive_and_execute_spacetime_command(
```

```
    const SpacetimeModulationCommand& command,
    const char* directive_id) {

    // Validate command bounds
    if (command.target_warp_field_strength < 0.0f ||
        command.target_warp_field_strength > MAX_WARP_FIELD_STRENGTH ||
        command.target_gravito_flux_bias < -MAX_GRAVITO_FLUX_BIAS ||
        command.target_gravito_flux_bias > MAX_GRAVITO_FLUX_BIAS ||
        command.target_time_dilation_factor < 1.0f ||
        command.target_time_dilation_factor > MAX_TIME_DILATION_FACTOR ||
        command.target_artificial_gravity_g < -MAX_INDUCED_GRAVITY_G ||
        command.target_artificial_gravity_g > MAX_INDUCED_GRAVITY_G ||
        command.target_power_budget_GW < MIN_POWER_DRAW_GW ||
        command.target_power_budget_GW > MAX_SYSTEM_POWER_DRAW_GW ||
        command.target_quantum_fluid_flow_rate < 0.0f) {

        PlatformHAL::metric_emit("apcu.command_rejected_oob", 1.0f,
            "directive_id", directive_id);
        return false;
    }

    // If in emergency mode, apply additional constraints
    SpacetimeModulationCommand validated_command = command;
    if (emergency_mode_active_) {
        apply_emergency_limits(validated_command);
    }

    active_spacetime_command_ = validated_command;
    std::strncpy(active_directive_id_, directive_id, sizeof(active_directive_id_) - 1);
    active_directive_id_[sizeof(active_directive_id_) - 1] = '\0';

    PlatformHAL::metric_emit("apcu.command_received", 1.0f,
        "directive_id", directive_id);
    return true;
}

float AdvancedPropulsionControlUnit::compute_pid_output(
    float error,
    float& integral,
    float& previous_error,
```

```
    float kp, float ki, float kd,
    float integral_limit,
    float elapsed_ms) {

    // Update integral with anti-windup
    integral += error * elapsed_ms;
    integral = std::max(-integral_limit, std::min(integral_limit, integral));

    // Compute derivative
    float derivative = 0.0f;
    if (elapsed_ms > 0.0f) {
        // Use time in seconds for a more stable derivative term if elapsed_ms is large
        float dt_s = elapsed_ms / 1000.0f;
        derivative = (error - previous_error) / dt_s;
    }

    // PID output
    float output = (kp * error) + (ki * integral) + (kd * derivative);

    // Update state
    previous_error = error;

    return output;
}

float AdvancedPropulsionControlUnit::compute_capability_scale() const {
    float scale = 1.0f;

    // Antimatter constraints (non-linear penalty below 10% capacity)
    if (current_propulsion_state_.remaining_antimatter_kg <
        INITIAL_ANTIMATTER_KG * 0.1f) {
        scale *= std::pow(current_propulsion_state_.remaining_antimatter_kg /
(INITIAL_ANTIMATTER_KG * 0.1f), 2.0f);
        scale = std::max(0.05f, scale); // Minimum scale to prevent zero division
    }

    // Quantum fluid constraints (similar non-linear penalty)
    if (current_propulsion_state_.quantum_fluid_level <
        INITIAL_QUANTUM_FLUID_LITERS * 0.1f) {
```

```
        scale *= std::pow(current_propulsion_state_.quantum_fluid_level /
(INITIAL_QUANTUM_FLUID_LITERS * 0.1f), 2.0f);
        scale = std::max(0.05f, scale);
    }

    return std::min(1.0f, scale);
}

// HLV Math Update: Curvature is a non-linear function of fields (W^3 and Flux^2)
float AdvancedPropulsionControlUnit::compute_spacetime_curvature() const {
    float W = current_propulsion_state_.warp_field_strength;
    float Phi_g = current_propulsion_state_.gravito_flux_bias;

    // C = (W)^3 * C_W + 0.5 * (Phi_g)^2 * C_Phi
    float curvature = (std::pow(W, 3.0f) * WARP_CURVATURE_CUBIC_SCALAR) +
                      (0.5f * std::pow(std::fabs(Phi_g), 2.0f) *
FLUX_CURVATURE_QUADRATIC_SCALAR);

    return std::min(MAX_SPACETIME_CURVATURE_MAGNITUDE, std::max(0.0f,
curvature));
}

// HLV Math Update: Dilation is an exponential function of curvature, modulated by fluid level
float AdvancedPropulsionControlUnit::compute_derived_time_dilation() const {
    float C = current_propulsion_state_.spacetime_curvature_magnitude;
    float L_fluid = current_propulsion_state_.quantum_fluid_level;

    // Fluid efficiency (non-linear saturation)
    float fluid_ratio = L_fluid / INITIAL_QUANTUM_FLUID_LITERS;
    float fluid_efficiency_mod = 1.0f - std::exp(-fluid_ratio *
FLUID_LEVEL_DAMPING_FACTOR);

    // D = 1.0 + (exp(C * C_dil) - 1) * fluid_efficiency_mod
    float base_dilation_from_C = 1.0f + (std::exp(C *
CURVATURE_TIME_DILATION_EXPONENT_BASE) - 1.0f);

    return 1.0f + ((base_dilation_from_C - 1.0f) * fluid_efficiency_mod);
}
```

```
// HLV Math Update: Gravity is primarily linear with flux, but modulated by the warp field magnitude
float AdvancedPropulsionControlUnit::compute_derived_gravity() const {
    float W = current_propulsion_state_.warp_field_strength;
    float Phi_g = current_propulsion_state_.gravito_flux_bias;

    // G = Phi_g * G_Phi * (1.0 + W * G_W)
    float gravity = Phi_g * FLUX_GRAVITY_BASE_SCALAR * (1.0f + W * WARP_GRAVITY_MODULATION_SCALAR);

    return gravity; // Bounded in update_internal_state
}

// HLV Math Update: Stress is an exponential function of combined high fields and dilation deviation, penalized by instability
float AdvancedPropulsionControlUnit::compute_field_coupling_stress() const {
    float W = current_propulsion_state_.warp_field_strength;
    float Phi_g = current_propulsion_state_.gravito_flux_bias;
    float D = current_propulsion_state_.time_dilation_factor;
    float S = current_propulsion_state_.spacetime_stability_index;

    // Prevent division by zero if stability is low
    float stability_penalty_factor = (S > 0.05f) ? (1.0f / S) : (1.0f / 0.05f);

    // Sigma = exp( (W * |Phi_g| * (D - 1.0) * Sigma_C) / S) - 1.0
    float stress_term = W * std::fabs(Phi_g) * (D - 1.0f) * COUPLING_STRESS_EXPONENT_SCALAR;

    float coupling_stress = std::exp(stress_term * stability_penalty_factor) - 1.0f;

    return std::max(0.0f, coupling_stress);
}

bool AdvancedPropulsionControlUnit::detect_resonance_instability() {
    // Update coupling history
    field_coupling_history_[coupling_history_index_] =
        current_propulsion_state_.field_coupling_stress;
    coupling_history_index_ = (coupling_history_index_ + 1) % RESONANCE_SAMPLE_COUNT;
```

```
    // Compute variance to detect oscillations
    float mean = 0.0f;
    for (uint32_t i = 0; i < RESONANCE_SAMPLE_COUNT; ++i) {
        mean += field_coupling_history_[i];
    }
    mean /= (float)RESONANCE_SAMPLE_COUNT;

    float variance = 0.0f;
    for (uint32_t i = 0; i < RESONANCE_SAMPLE_COUNT; ++i) {
        float diff = field_coupling_history_[i] - mean;
        variance += diff * diff;
    }
    variance /= (float)RESONANCE_SAMPLE_COUNT;

    // High variance + high coupling = resonance
    // The variance threshold is relative to the coupling stress
    bool resonant = (variance > (mean * 0.05f) && mean > 0.5f);

    if (resonant) {
        PlatformHAL::metric_emit("apcu.resonance_detected", 1.0f,
            "coupling_stress", std::to_string(mean).c_str());
    }

    return resonant;
}

void AdvancedPropulsionControlUnit::apply_resonance_suppression(
    float& warp_change,
    float& flux_change) {

    // Dampen changes when resonance is active
    float suppression_factor = 0.3f;
    warp_change *= suppression_factor;
    flux_change *= suppression_factor;

    PlatformHAL::metric_emit("apcu.resonance_suppression_active", 1.0f);
}

// HLV Math Update: Power scales non-linearly with fields and quadratically with slew rate
float AdvancedPropulsionControlUnit::compute_power_draw(
```

```
    float warp_slew,
    float flux_slew) const {

    float W = current_propulsion_state_.warp_field_strength;
    float Phi_g = current_propulsion_state_.gravito_flux_bias;
    float C = current_propulsion_state_.spacetime_curvature_magnitude;

    // Base power from field magnitudes: P_W * W^3 + P_C * C^2 + P_Phi * |Phi_g|
    float base_power = MIN_POWER_DRAW_GW +
                       (std::pow(W, 3.0f) * POWER_WARP_CUBIC_SCALAR) +
                       (std::pow(C, 2.0f) * POWER_CURVATURE_QUADRATIC_SCALAR) +
                       (std::fabs(Phi_g) * POWER_FLUX_LINEAR_SCALAR);

    // Quadratic penalty for rapid field changes (slew rate in unit/ms)
    float slew_penalty = POWER_SLEW_PENALTY_SCALE * (std::pow(std::fabs(warp_slew),
POWER_SLEW_PENALTY_EXPONENT) +
                         std::pow(std::fabs(flux_slew), POWER_SLEW_PENALTY_EXPONENT));

    return base_power + slew_penalty;
}

// HLV Math Update: Efficiency scales with W^2/P and is penalized by Gaussian power draw
and fluid depletion
float AdvancedPropulsionControlUnit::compute_subspace_efficiency(
    const SpacetimeModulationState& state) const {

    float W = state.warp_field_strength;
    float P = state.power_draw_GW;

    // 1. Base Efficiency: Scales with W^2 and inversely with Power (W^2/P)
    // Ensures efficiency is only high when W is high AND power usage is reasonable
    float base_efficiency = (W > 0.1f && P > MIN_POWER_DRAW_GW) ?
                            (EFFICIENCY_WARP_QUADRATIC_SCALAR * std::pow(W, 2.0f) / P) :
0.0f;

    // 2. Power Penalty (Gaussian Curve): Efficiency peaks at a specific power level
    float power_diff = state.power_draw_GW - EFFICIENCY_POWER_PEAK_GW;
    float power_penalty = std::exp(-0.5f * std::pow(power_diff /
EFFICIENCY_POWER_VARIANCE_GW, 2.0f));
```

```
    // 3. Fluid Penalty: Non-linear penalty for low fluid level (Fluid_Ratio^Exponent)
    float fluid_ratio = state.quantum_fluid_level / INITIAL_QUANTUM_FLUID_LITERS;
    float fluid_modulation = std::pow(std::min(1.0f, fluid_ratio),
EFFICIENCY_FLUID_EXPONENT);

    // 4. Stability Bonus: Quadratic boost for high stability
    float stability_bonus = state.spacetime_stability_index * state.spacetime_stability_index *
10.0f; // Max 10% boost

    float efficiency = base_efficiency * power_penalty * fluid_modulation + stability_bonus;

    return std::max(0.0f, std::min(RAPSConfig::MAX_SUBSPACE_EFFICIENCY, efficiency));
}

// HLV Math Update: Stability is inversely related to Curvature and quadratically penalized by
Stress
float AdvancedPropulsionControlUnit::compute_stability_index() const {
    float C = current_propulsion_state_.spacetime_curvature_magnitude;
    float Sigma = current_propulsion_state_.field_coupling_stress;

    // Curvature Penalty: S_C * C^3
    float curvature_penalty = STABILITY_CURVATURE_CUBIC_SCALAR * std::pow(C, 3.0f);

    // Stress Penalty: S_Sigma * Sigma^2
    float stress_penalty = STABILITY_STRESS_QUADRATIC_SCALAR * std::pow(Sigma,
2.0f);

    // Base Stability (1.0) - penalties
    float base_stability = 1.0f - curvature_penalty - stress_penalty;

    // Mismatch Penalty (Original logic is good for penalizing active control error)
    float warp_mismatch = std::fabs(current_propulsion_state_.warp_field_strength -
                                    active_spacetime_command_.target_warp_field_strength) /
MAX_WARP_FIELD_STRENGTH;
    float flux_mismatch = std::fabs(current_propulsion_state_.gravito_flux_bias -
                                    active_spacetime_command_.target_gravito_flux_bias) /
MAX_GRAVITO_FLUX_BIAS;

    float mismatch_penalty = (warp_mismatch + flux_mismatch) * 0.1f; // Small penalty for
active hunting
```

```
    float final_stability = base_stability - mismatch_penalty;

    return std::max(0.0f, std::min(1.0f, final_stability));
}

float AdvancedPropulsionControlUnit::compute_control_authority() const {
    // How much control margin remains before hitting limits
    float warp_margin = 1.0f - (current_propulsion_state_.warp_field_strength /
MAX_WARP_FIELD_STRENGTH);
    float flux_margin = 1.0f - (std::fabs(current_propulsion_state_.gravito_flux_bias) /
MAX_GRAVITO_FLUX_BIAS);
    float power_margin = 1.0f - (current_propulsion_state_.power_draw_GW /
MAX_SYSTEM_POWER_DRAW_GW);

    // Combine margins, stability, and resource capability for overall authority
    float resource_capability = compute_capability_scale();
    float stability_factor = current_propulsion_state_.spacetime_stability_index;

    // Authority is the average of resource/stability factors and physical margins
    float authority = (resource_capability * 0.3f) + (stability_factor * 0.3f) +
                      ((warp_margin + flux_margin + power_margin) / 3.0f * 0.4f);

    return std::max(0.0f, std::min(1.0f, authority));
}

void AdvancedPropulsionControlUnit::consume_resources(uint32_t elapsed_ms) {
    // Antimatter consumption based on power draw
    float antimatter_consumed = current_propulsion_state_.power_draw_GW *
ANTIMATTER_BURN_RATE_GW_TO_KG_PER_MS * elapsed_ms;
    current_propulsion_state_.remaining_antimatter_kg -= antimatter_consumed;
    current_propulsion_state_.remaining_antimatter_kg =
        std::max(0.0f, current_propulsion_state_.remaining_antimatter_kg);

    // HLV Math Update: Fluid consumption scales non-linearly with curvature (C^1.5)
    float C = current_propulsion_state_.spacetime_curvature_magnitude;
    float fluid_consumed_base = QUANTUM_FLUID_BASE_CONSUMPTION_RATE *
elapsed_ms;
```

```
    float fluid_consumed_curvature = std::pow(C,
QUANTUM_FLUID_CONSUMPTION_CURVATURE_EXPONENT) *
QUANTUM_FLUID_CONSUMPTION_PER_CURVATURE_UNIT_MS * elapsed_ms;

    // Fluid injection from command
    float fluid_injected = active_spacetime_command_.target_quantum_fluid_flow_rate *
(elapsed_ms / 1000.0f);

    current_propulsion_state_.quantum_fluid_level +=
        (fluid_injected - fluid_consumed_base - fluid_consumed_curvature);
    current_propulsion_state_.quantum_fluid_level =
        std::max(0.0f, std::min(INITIAL_QUANTUM_FLUID_LITERS * 1.2f,
                                current_propulsion_state_.quantum_fluid_level));
}

void AdvancedPropulsionControlUnit::update_internal_state(uint32_t elapsed_ms) {
    if (elapsed_ms == 0) return;

    float dt_s = static_cast<float>(elapsed_ms) / 1000.0f;

    // Compute resource constraints
    float capability_scale = compute_capability_scale();
    float effective_power_budget = active_spacetime_command_.target_power_budget_GW *
capability_scale;

    // Apply emergency damping if active
    float response_scale = emergency_mode_active_ ?
EMERGENCY_RESPONSE_DAMPING_FACTOR : 1.0f;

    //
================================================================================
====
    // 1. Warp Field Control (Full PID)
    //
================================================================================
====
    float warp_error = active_spacetime_command_.target_warp_field_strength -
                       current_propulsion_state_.warp_field_strength;

    float warp_pid_output = compute_pid_output(
```

```
        warp_error,
        warp_error_integral_,
        warp_error_previous_,
        WARP_KP, WARP_KI, WARP_KD,
        WARP_INTEGRAL_LIMIT,
        elapsed_ms
    );

    float warp_change_request = warp_pid_output * capability_scale * response_scale;
    warp_change_request = std::max(-WARP_FIELD_RESPONSE_RATE_PER_MS *
elapsed_ms,
                            std::min(WARP_FIELD_RESPONSE_RATE_PER_MS * elapsed_ms,
                                warp_change_request));

    //
==========================================================================
====
    // 2. Gravito-Flux Control (Full PID)
    //
==========================================================================
====
    float flux_error = active_spacetime_command_.target_gravito_flux_bias -
                    current_propulsion_state_.gravito_flux_bias;

    float flux_pid_output = compute_pid_output(
        flux_error,
        flux_error_integral_,
        flux_error_previous_,
        FLUX_KP, FLUX_KI, FLUX_KD,
        FLUX_INTEGRAL_LIMIT,
        elapsed_ms
    );

    float flux_change_request = flux_pid_output * capability_scale * response_scale;
    flux_change_request = std::max(-GRAVITO_FLUX_RESPONSE_RATE_PER_MS *
elapsed_ms,
                            std::min(GRAVITO_FLUX_RESPONSE_RATE_PER_MS * elapsed_ms,
                                flux_change_request));
```

```
    //
================================================================================
    // 3. Resonance Detection & Suppression
    //
================================================================================
    if (active_spacetime_command_.enable_resonance_suppression &&
detect_resonance_instability()) {
        apply_resonance_suppression(warp_change_request, flux_change_request);
    }

    // Apply field changes
    current_propulsion_state_.warp_field_strength += warp_change_request;
    current_propulsion_state_.warp_field_strength =
        std::max(0.0f, std::min(MAX_WARP_FIELD_STRENGTH,
                        current_propulsion_state_.warp_field_strength));

    current_propulsion_state_.gravito_flux_bias += flux_change_request;
    current_propulsion_state_.gravito_flux_bias =
        std::max(-MAX_GRAVITO_FLUX_BIAS, std::min(MAX_GRAVITO_FLUX_BIAS,
                                        current_propulsion_state_.gravito_flux_bias));

    //
================================================================================
    // 4. Derived Spacetime Curvature (Physics Simulation)
    //
================================================================================
    // Update curvature immediately based on new field states
    float new_curvature = compute_spacetime_curvature();

    // Spacetime inertia: Curvature changes gradually towards the value derived from fields
    float curvature_error = new_curvature -
current_propulsion_state_.spacetime_curvature_magnitude;
    float curvature_change = curvature_error * 0.1f * dt_s;

    current_propulsion_state_.spacetime_curvature_magnitude += curvature_change;
    current_propulsion_state_.spacetime_curvature_magnitude =
```

```
        std::max(0.0f, std::min(MAX_SPACETIME_CURVATURE_MAGNITUDE,
                        current_propulsion_state_.spacetime_curvature_magnitude));

    //
================================================================================
    // 5. Time Dilation Control
    //
================================================================================
    if (active_spacetime_command_.enable_time_dilation_coupling) {
        // Active control mode: Try to achieve target via quantum field modulation
        float dilation_error = active_spacetime_command_.target_time_dilation_factor -
                        current_propulsion_state_.time_dilation_factor;

        float dilation_pid_output = compute_pid_output(
            dilation_error,
            dilation_error_integral_,
            dilation_error_previous_,
            DILATION_KP, DILATION_KI, DILATION_KD,
            0.5f,  // Integral limit
            elapsed_ms
        );

        float dilation_change = dilation_pid_output * capability_scale * response_scale;
        dilation_change = std::max(-TIME_DILATION_RESPONSE_RATE_PER_MS *
elapsed_ms,
                            std::min(TIME_DILATION_RESPONSE_RATE_PER_MS * elapsed_ms,
                                dilation_change));

        current_propulsion_state_.time_dilation_factor += dilation_change;
    } else {
        // Passive mode: Dilation is purely derived from curvature and fluid
        current_propulsion_state_.time_dilation_factor = compute_derived_time_dilation();
    }

    current_propulsion_state_.time_dilation_factor =
        std::max(1.0f, std::min(MAX_TIME_DILATION_FACTOR,
                        current_propulsion_state_.time_dilation_factor));
```

```
    //
================================================================================
    // 6. Artificial Gravity Control (Full PID)
    //
================================================================================
    float gravity_error = active_spacetime_command_.target_artificial_gravity_g -
                          current_propulsion_state_.induced_gravity_g;

    float gravity_pid_output = compute_pid_output(
        gravity_error,
        gravity_error_integral_,
        gravity_error_previous_,
        GRAVITY_KP, GRAVITY_KI, GRAVITY_KD,
        0.5f,
        elapsed_ms
    );

    float gravity_change = gravity_pid_output * capability_scale * response_scale;
    gravity_change = std::max(-GRAVITY_RESPONSE_RATE_PER_MS * elapsed_ms,
                              std::min(GRAVITY_RESPONSE_RATE_PER_MS * elapsed_ms,
                                       gravity_change));

    // Gravity is primarily derived from flux, but PID fine-tunes the derived value
    float derived_gravity = compute_derived_gravity();
    // Gravity output is a blend of the derived physics value and the PID adjustment
    current_propulsion_state_.induced_gravity_g = derived_gravity + gravity_change;

    current_propulsion_state_.induced_gravity_g =
        std::max(-MAX_INDUCED_GRAVITY_G, std::min(MAX_INDUCED_GRAVITY_G,
                                                  current_propulsion_state_.induced_gravity_g));

    //
================================================================================
    // 7. Power Draw & Resource Consumption
    //
================================================================================
```

```
    // Slew rates for power calculation are in unit/ms
    current_propulsion_state_.power_draw_GW = compute_power_draw(
        warp_change_request / elapsed_ms,
        flux_change_request / elapsed_ms
    );

    // Apply effective power budget (constrained by resource capability)
    current_propulsion_state_.power_draw_GW =
        std::min(effective_power_budget, current_propulsion_state_.power_draw_GW);
    current_propulsion_state_.power_draw_GW =
        std::max(MIN_POWER_DRAW_GW, current_propulsion_state_.power_draw_GW);

    consume_resources(elapsed_ms);

    //
================================================================================
====
    // 8. Efficiency & Displacement
    //
================================================================================
====
    current_propulsion_state_.subspace_efficiency_pct =
        compute_subspace_efficiency(current_propulsion_state_);

    current_propulsion_state_.total_displacement_km +=
        (current_propulsion_state_.warp_field_strength *
(current_propulsion_state_.subspace_efficiency_pct / 100.0f) *
WARP_TO_DISPLACEMENT_FACTOR_KM_PER_S * dt_s);

    //
================================================================================
====
    // 9. Diagnostic Metrics (Must be computed AFTER core state changes)
    //
================================================================================
====
    current_propulsion_state_.field_coupling_stress = compute_field_coupling_stress();
    current_propulsion_state_.spacetime_stability_index = compute_stability_index();
    current_propulsion_state_.control_authority_remaining = compute_control_authority();
    current_propulsion_state_.emergency_mode_active = emergency_mode_active_;
```

```
    //
================================================================================
    // 10. State Management & Safety
    //
================================================================================
    current_propulsion_state_.timestamp_ms += elapsed_ms;
    current_propulsion_state_.state_hash = calculate_state_hash(current_propulsion_state_);

    // Save safe state periodically
    if (is_state_safe_to_save(current_propulsion_state_) &&
        (current_propulsion_state_.timestamp_ms - last_safe_state_timestamp_ms_ > 1000)) {
        save_safe_state();
    }

    // Check for emergency conditions
    if (!is_operational_state_safe() && !emergency_mode_active_) {
        enter_emergency_mode();
    }

    // Emit comprehensive metrics
    PlatformHAL::metric_emit("apcu.power_draw_GW",
current_propulsion_state_.power_draw_GW);
    PlatformHAL::metric_emit("apcu.warp_strength",
current_propulsion_state_.warp_field_strength);
    PlatformHAL::metric_emit("apcu.flux_bias", current_propulsion_state_.gravito_flux_bias);
    PlatformHAL::metric_emit("apcu.curvature_mag",
current_propulsion_state_.spacetime_curvature_magnitude);
    PlatformHAL::metric_emit("apcu.time_dilation_factor",
current_propulsion_state_.time_dilation_factor);
    PlatformHAL::metric_emit("apcu.induced_gravity_g",
current_propulsion_state_.induced_gravity_g);
    PlatformHAL::metric_emit("apcu.subspace_efficiency_pct",
current_propulsion_state_.subspace_efficiency_pct);
    PlatformHAL::metric_emit("apcu.total_displacement_km",
(float)current_propulsion_state_.total_displacement_km);
    PlatformHAL::metric_emit("apcu.antimatter_kg",
current_propulsion_state_.remaining_antimatter_kg);
```

```
    PlatformHAL::metric_emit("apcu.quantum_fluid_L",
current_propulsion_state_.quantum_fluid_level);
    PlatformHAL::metric_emit("apcu.coupling_stress",
current_propulsion_state_.field_coupling_stress);
    PlatformHAL::metric_emit("apcu.stability_index",
current_propulsion_state_.spacetime_stability_index);
    PlatformHAL::metric_emit("apcu.control_authority",
current_propulsion_state_.control_authority_remaining);
}

void AdvancedPropulsionControlUnit::save_safe_state() {
    last_safe_state_ = current_propulsion_state_;
    last_safe_state_timestamp_ms_ = current_propulsion_state_.timestamp_ms;
    PlatformHAL::metric_emit("apcu.safe_state_saved", 1.0f);
}

bool AdvancedPropulsionControlUnit::is_state_safe_to_save(
    const SpacetimeModulationState& state) const {
    return state.remaining_antimatter_kg >
RAPSConfig::EMERGENCY_ANTIMATTER_RESERVE_KG &&
           state.quantum_fluid_level >
RAPSConfig::EMERGENCY_QUANTUM_FLUID_LITERS &&
           state.field_coupling_stress <
RAPSConfig::CRITICAL_FIELD_COUPLING_THRESHOLD &&
           state.spacetime_stability_index > 0.6f;
}

void AdvancedPropulsionControlUnit::enter_emergency_mode() {
    emergency_mode_active_ = true;
    current_propulsion_state_.emergency_mode_active = true;

    // Reset PID integrals to prevent windup in emergency
    warp_error_integral_ = 0.0f;
    warp_error_previous_ = 0.0f;
    flux_error_integral_ = 0.0f;
    flux_error_previous_ = 0.0f;
    dilation_error_integral_ = 0.0f;
    dilation_error_previous_ = 0.0f;
    gravity_error_integral_ = 0.0f;
    gravity_error_previous_ = 0.0f;
```

```
    fluid_error_integral_ = 0.0f;
    fluid_error_previous_ = 0.0f;

    PlatformHAL::metric_emit("apcu.emergency_mode_activated", 1.0f);
}

void AdvancedPropulsionControlUnit::apply_emergency_limits(
    SpacetimeModulationCommand& command) {

    // Reduce all targets to conservative values
    command.target_warp_field_strength *= 0.5f;
    command.target_gravito_flux_bias *= 0.3f;
    command.target_time_dilation_factor = 1.0f +
        (command.target_time_dilation_factor - 1.0f) * 0.3f;
    command.target_artificial_gravity_g *= 0.5f;
    command.target_power_budget_GW = std::min(command.target_power_budget_GW,
                                      MAX_SYSTEM_POWER_DRAW_GW * 0.6f);

    // Force conservative control modes
    command.enable_emergency_damping = true;
    command.enable_resonance_suppression = true;

    PlatformHAL::metric_emit("apcu.emergency_limits_applied", 1.0f);
}

bool AdvancedPropulsionControlUnit::initiate_emergency_spacetime_collapse() {
    PlatformHAL::metric_emit("apcu.emergency_collapse_initiated", 1.0f);

    // Create emergency shutdown command
    SpacetimeModulationCommand emergency_command = {
        0.0f,  // Collapse warp field
        0.0f,  // Neutralize flux
        1.0f,  // Return to normal time
        0.0f,  // Remove artificial gravity
        0.0f,  // Stop fluid flow
        MIN_POWER_DRAW_GW,
        true, true, false // Force damping, suppression, disable active dilation
    };

    // Override current command
```

```
    active_spacetime_command_ = emergency_command;
    std::strncpy(active_directive_id_, "EMERGENCY_COLLAPSE", sizeof(active_directive_id_)
- 1);
    active_directive_id_[sizeof(active_directive_id_) - 1] = '\0';

    enter_emergency_mode();

    return true;
}

bool AdvancedPropulsionControlUnit::execute_controlled_shutdown() {
    PlatformHAL::metric_emit("apcu.controlled_shutdown_initiated", 1.0f);

    // Similar to emergency collapse but more gradual
    SpacetimeModulationCommand shutdown_command = {
        0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
        MIN_POWER_DRAW_GW,
        false, false, false  // Allow smooth transition
    };

    return receive_and_execute_spacetime_command(shutdown_command,
"CONTROLLED_SHUTDOWN");
}

bool AdvancedPropulsionControlUnit::restore_from_safe_state(
    const SpacetimeModulationState& safe_state) {

    // Validate that the provided state is actually safe
    if (!is_state_safe_to_save(safe_state)) {
        PlatformHAL::metric_emit("apcu.restore_rejected_unsafe_state", 1.0f);
        return false;
    }

    // Check resource availability for restoration
    if (safe_state.remaining_antimatter_kg > current_propulsion_state_.remaining_antimatter_kg
||
        safe_state.quantum_fluid_level > current_propulsion_state_.quantum_fluid_level) {
        PlatformHAL::metric_emit("apcu.restore_rejected_insufficient_resources", 1.0f);
        return false;
    }
```

```
    // Restore state (physics properties only, not resources)
    current_propulsion_state_.warp_field_strength = safe_state.warp_field_strength;
    current_propulsion_state_.gravito_flux_bias = safe_state.gravito_flux_bias;
    current_propulsion_state_.spacetime_curvature_magnitude =
safe_state.spacetime_curvature_magnitude;
    current_propulsion_state_.time_dilation_factor = safe_state.time_dilation_factor;
    current_propulsion_state_.induced_gravity_g = safe_state.induced_gravity_g;

    // Reset PID controllers to prevent instability
    warp_error_integral_ = 0.0f;
    warp_error_previous_ = 0.0f;
    flux_error_integral_ = 0.0f;
    flux_error_previous_ = 0.0f;
    dilation_error_integral_ = 0.0f;
    dilation_error_previous_ = 0.0f;
    gravity_error_integral_ = 0.0f;
    gravity_error_previous_ = 0.0f;

    // Exit emergency mode if we successfully restored to safe state
    if (emergency_mode_active_) {
        emergency_mode_active_ = false;
        current_propulsion_state_.emergency_mode_active = false;
        PlatformHAL::metric_emit("apcu.emergency_mode_deactivated", 1.0f);
    }

    PlatformHAL::metric_emit("apcu.state_restored", 1.0f);
    return true;
}

SpacetimeModulationState AdvancedPropulsionControlUnit::get_current_state() const {
    return current_propulsion_state_;
}

bool AdvancedPropulsionControlUnit::is_operational_state_safe() const {
    bool safe = true;

    // Critical resource checks
    if (current_propulsion_state_.remaining_antimatter_kg <
RAPSConfig::CRITICAL_ANTIMATTER_KG) {
```

```
        PlatformHAL::metric_emit("apcu.safety_fuel_critical", 1.0f);
        safe = false;
    }

    if (current_propulsion_state_.quantum_fluid_level <
RAPSConfig::CRITICAL_QUANTUM_FLUID_LITERS) {
        PlatformHAL::metric_emit("apcu.safety_quantum_fluid_critical", 1.0f);
        safe = false;
    }

    // Power system checks
    if (current_propulsion_state_.power_draw_GW > MAX_SYSTEM_POWER_DRAW_GW *
0.98f) {
        PlatformHAL::metric_emit("apcu.safety_power_critical", 1.0f);
        safe = false;
    }

    // Spacetime distortion limits
    if (current_propulsion_state_.spacetime_curvature_magnitude >
MAX_SPACETIME_CURVATURE_MAGNITUDE * 0.98f) {
        PlatformHAL::metric_emit("apcu.safety_curvature_critical", 1.0f);
        safe = false;
    }

    if (current_propulsion_state_.time_dilation_factor > MAX_TIME_DILATION_FACTOR *
0.98f) {
        PlatformHAL::metric_emit("apcu.safety_time_dilation_critical", 1.0f);
        safe = false;
    }

    if (std::fabs(current_propulsion_state_.induced_gravity_g) > MAX_INDUCED_GRAVITY_G
* 0.98f) {
        PlatformHAL::metric_emit("apcu.safety_gravity_critical", 1.0f);
        safe = false;
    }

    // Field integrity checks (critical - prevents singularities)
    if (current_propulsion_state_.warp_field_strength > MAX_WARP_FIELD_STRENGTH *
1.01f ||
```

```
        std::fabs(current_propulsion_state_.gravito_flux_bias) > MAX_GRAVITO_FLUX_BIAS *
1.01f) {
        PlatformHAL::metric_emit("apcu.safety_field_oob_critical", 1.0f);
        safe = false;
    }

    // Resonance/coupling stress
    if (current_propulsion_state_.field_coupling_stress >
RAPSConfig::CRITICAL_FIELD_COUPLING_THRESHOLD) {
        PlatformHAL::metric_emit("apcu.safety_coupling_stress_critical", 1.0f);
        safe = false;
    }

    // Stability index
    if (current_propulsion_state_.spacetime_stability_index < 0.3f) {
        PlatformHAL::metric_emit("apcu.safety_stability_critical", 1.0f);
        safe = false;
    }

    // Control authority (if we're losing control)
    if (current_propulsion_state_.control_authority_remaining < 0.1f) {
        PlatformHAL::metric_emit("apcu.safety_control_authority_low", 1.0f);
        safe = false;
    }

    return safe;
};
```