TOSCA Architecture

Design Principles

Safety First: Multiple independent safety layers ensure the laser cannot fire unless all checks pass. Hardware interlocks work even if software crashes.

Complete Audit Trail: Every action is logged permanently with who did what, when, and why. Logs are append-only and tamper-evident.

Hardware Independence: The software works with both real devices and simulated ones, enabling thorough testing without expensive equipment.

Thread Safety: PyQt6's signal/slot pattern keeps hardware operations and UI updates safely separated without manual locking.

Regulatory Context

TOSCA is designed to meet these standards:

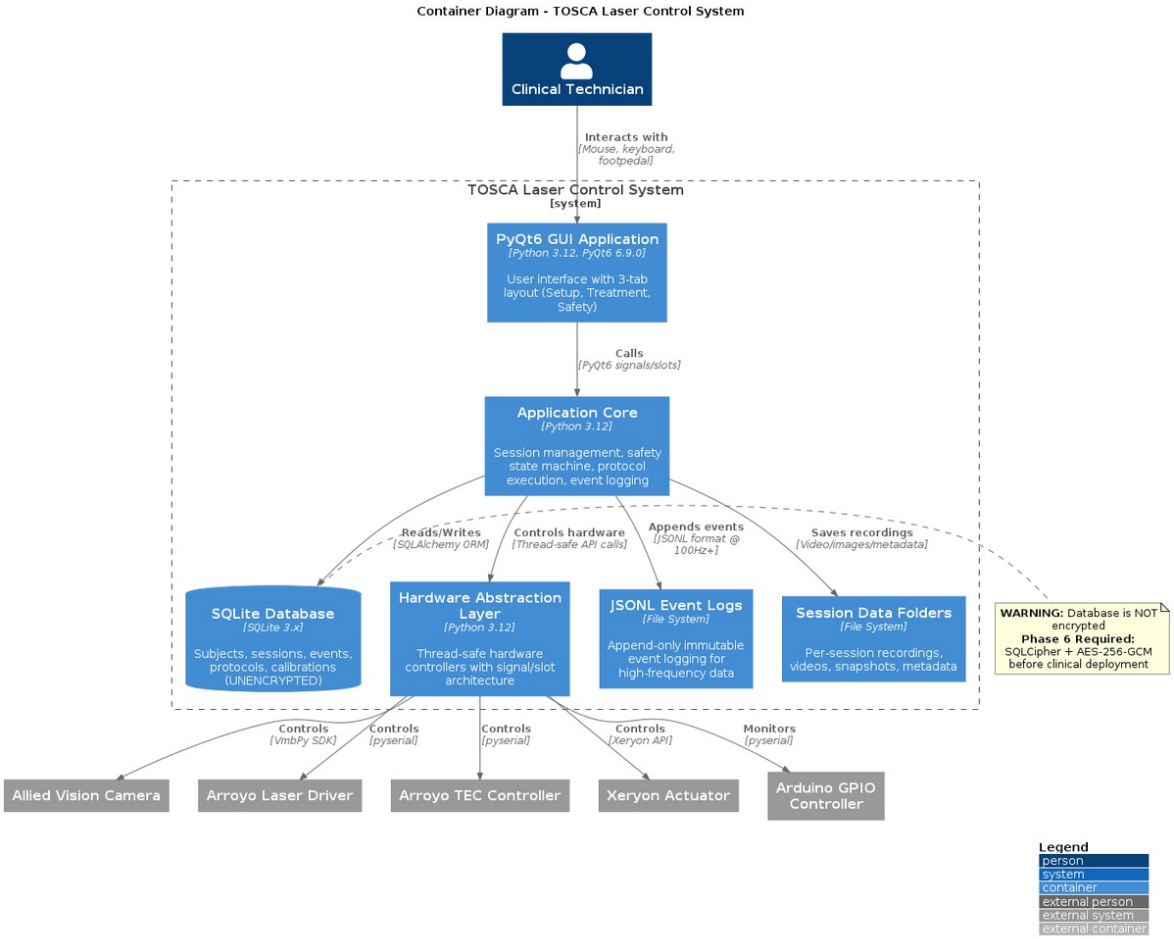
- **IEC 62304:** Software development lifecycle for medical devices
- **21 CFR Part 820:** Quality system requirements (design controls)
- **21 CFR Part 11:** Electronic records must be tamper-evident
- **IEC 60601-1:** Medical equipment safety (hardware interlocks, watchdog)
- **ISO 14971:** Risk management (identify hazards, implement controls)

These standards shape every architectural decision. They're not just documentation requirements - they influence code structure, testing strategy, and deployment architecture.

System Architecture Overview

Four-Layer Architecture

TOSCA uses a layered architecture that separates concerns and enables independent testing of each layer.



TOSCA Container Diagram

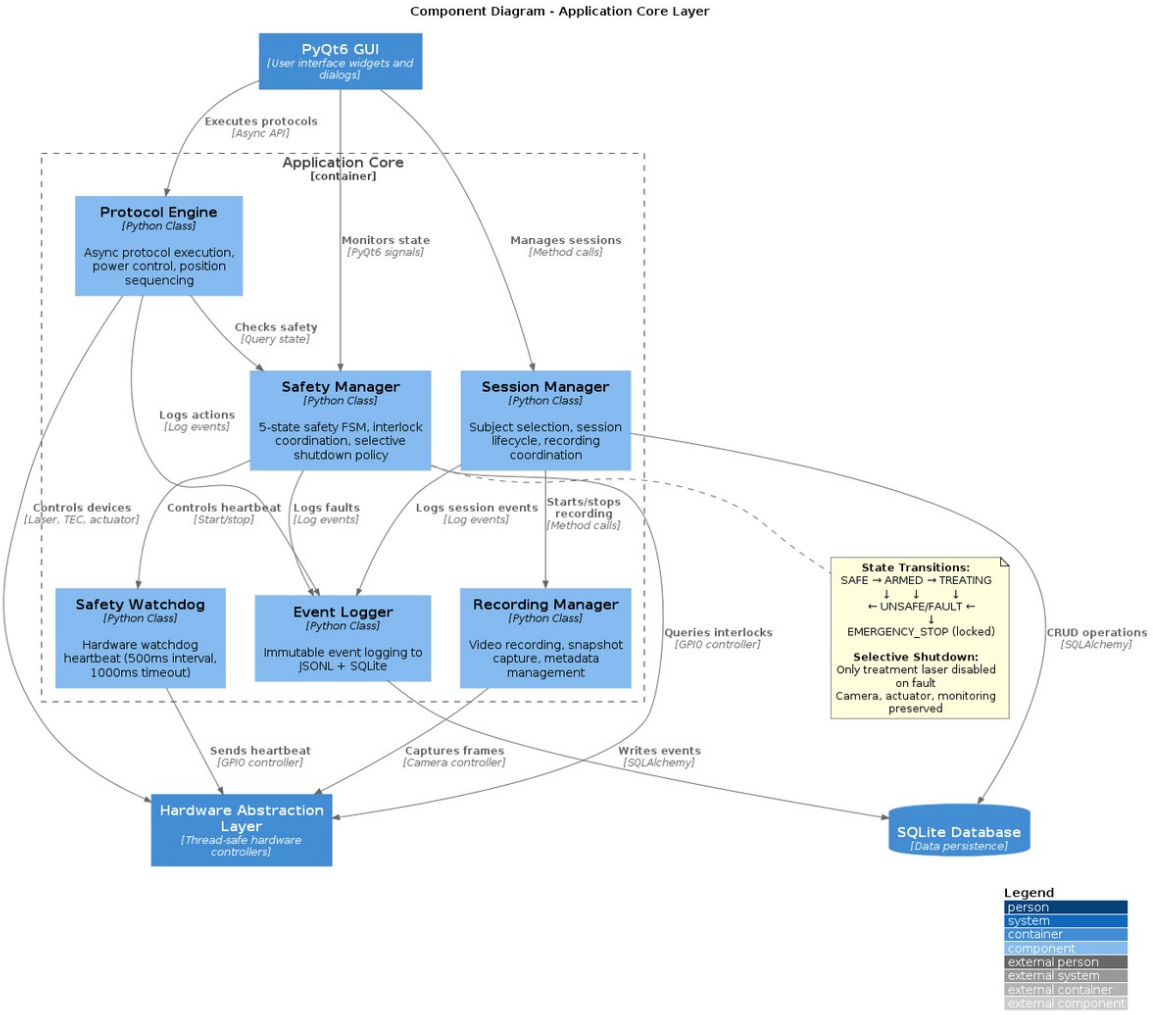
Layer 1: User Interface (PyQt6)

The operator interface provides three main tabs: - **Hardware Tab:** Device connection, diagnostics, initialization sequence - **Treatment Tab:** Protocol loading, execution monitoring, active treatment dashboard - **Safety Tab:** Interlock status, event log viewer, safety state display

The emergency stop button appears on all screens with highest visual priority (60px red button, always visible).

Layer 2: Core Application Logic

The core layer contains:

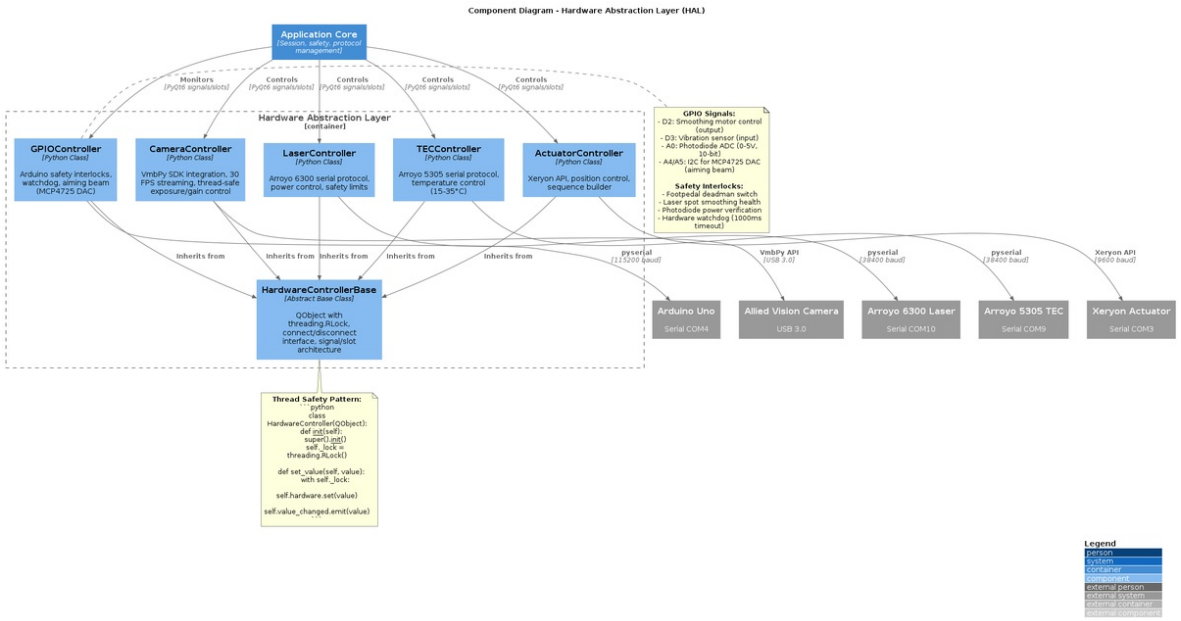


TOSCA Component Diagram - Application Core

- **Safety Manager** (safety.py): Aggregates all safety interlocks and makes laser enable/disable decisions. Implements 5-state safety state machine.
- **Protocol Engine** (protocol_engine.py): Executes treatment protocols asynchronously. Coordinates laser power, actuator position, and timing.
- **Session Manager** (session_manager.py): Manages treatment sessions, links events to sessions, validates operator permissions.
- **Event Logger** (event_logger.py): Writes all events to SQLite database and JSONL files. Implements append-only logging with checksums.

Layer 3: Hardware Abstraction Layer

Every hardware device has a dedicated controller:



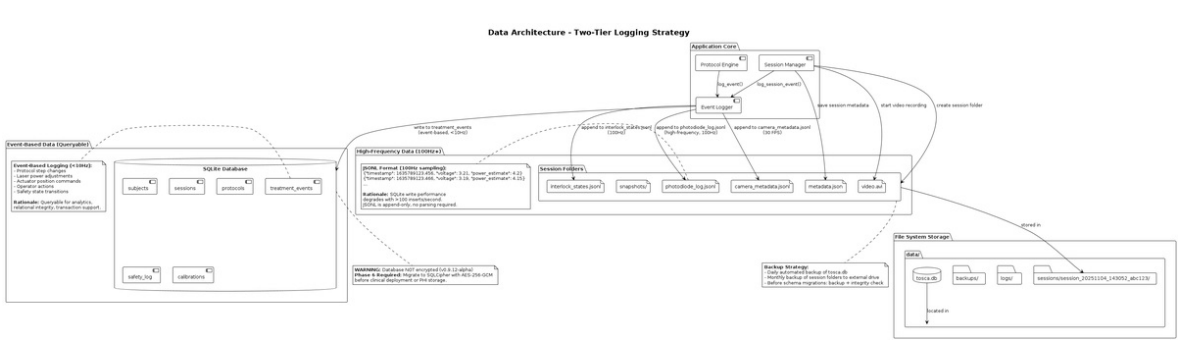
TOSCA Component Diagram - Hardware Abstraction Layer

- **Laser Controller** (laser_controller.py): Arroyo 6300 driver, serial communication (COM10, 38400 baud), power control with safety limits
- **TEC Controller** (tec_controller.py): Arroyo 5305 temperature controller (COM9, 38400 baud)
- **Actuator Controller** (actuator_controller.py): Xeryon linear stage (COM3, 9600 baud), 45mm travel
- **Camera Controller** (camera_controller.py): Allied Vision 1800 U-158c (USB 3.0, VmbPy SDK)
- **GPIO Controller** (gpio_controller.py): Arduino Uno (COM4, 115200 baud), safety interlocks, watchdog

All controllers inherit from HardwareControllerBase, providing consistent connect/disconnect interface and PyQt6 signal emission.

Layer 4: Data Persistence

Data storage uses multiple formats for different purposes:



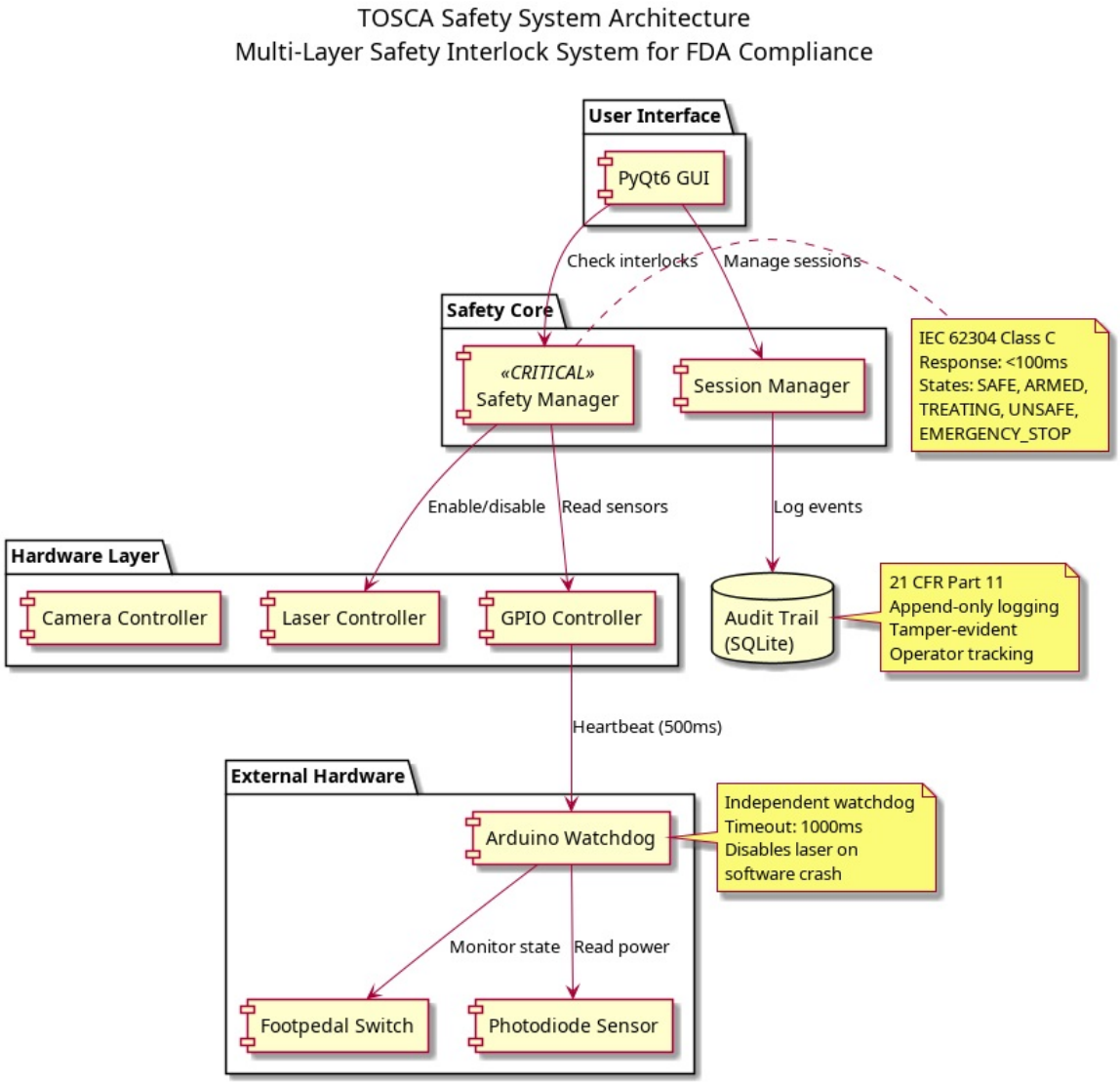
TOSCA Data Architecture

- **SQLite database** (data/tosca.db): Relational storage, indexed queries, ACID compliance via WAL mode
- **JSONL files** (data/logs/events_YYYYMMDD.jsonl): Human-readable backup, survives database corruption
- **Protocol files** (protocols/*.json): Treatment definitions, version-controlled

Both event logging methods are append-only for FDA 21 CFR Part 11 compliance.

Safety Architecture: Defense in Depth

The safety system implements four independent layers. Each layer can independently disable the laser if it detects a fault.



Simple Safety Architecture

Layer 1: Hardware Interlocks (Primary Safety)

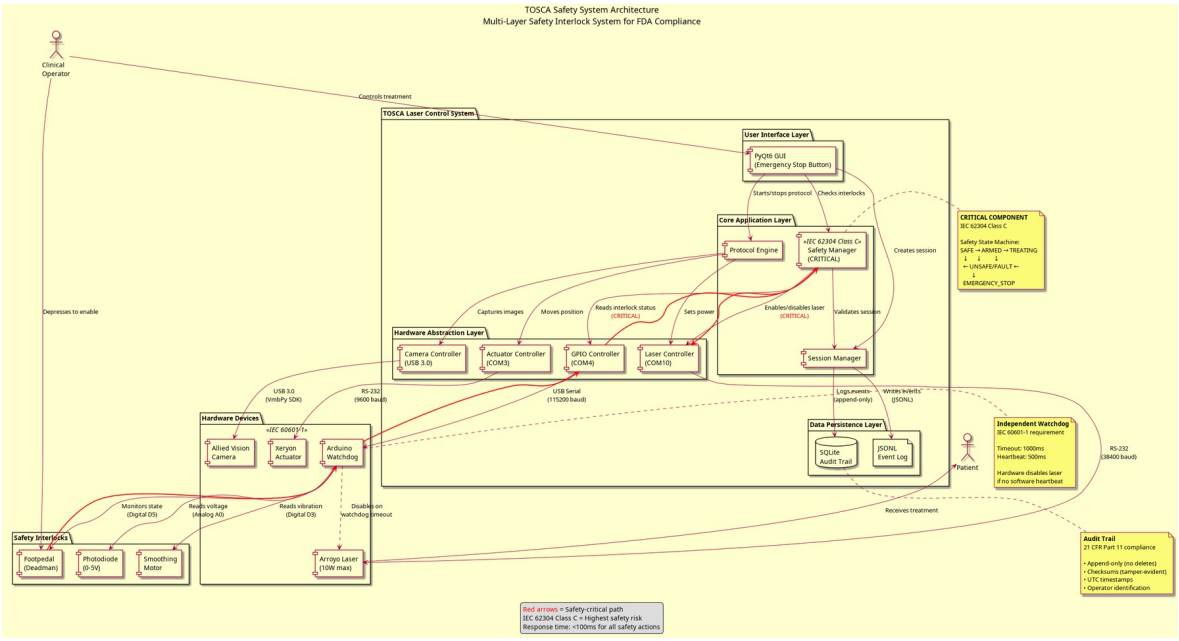
These hardware interlocks operate independently of software:

Footpedal Deadman Switch (Arduino D5, active-high) - Operator must continuously depress footpedal for laser operation - Release immediately disables laser at hardware level - No software required - Arduino firmware directly controls laser enable line - Response time: <50ms from release to laser disable

Photodiode Power Verification (Arduino A0, 0-5V analog) - Continuous monitoring of actual laser output via pickoff - 10-bit ADC resolution (0-1023 counts) - Validates commanded power matches actual output - Tolerance: ±5% deviation triggers safety fault - Sampling rate: 2 Hz (500ms polling)

Smoothing Motor Health Monitoring (Arduino D2 motor control, D3 vibration sense) - Dual-signal validation: motor drive active AND vibration detected - MPU6050 accelerometer on I2C bus (address 0x68) - If motor commanded ON but no vibration detected → safety fault - Ensures laser spot smoothing device operational before treatment

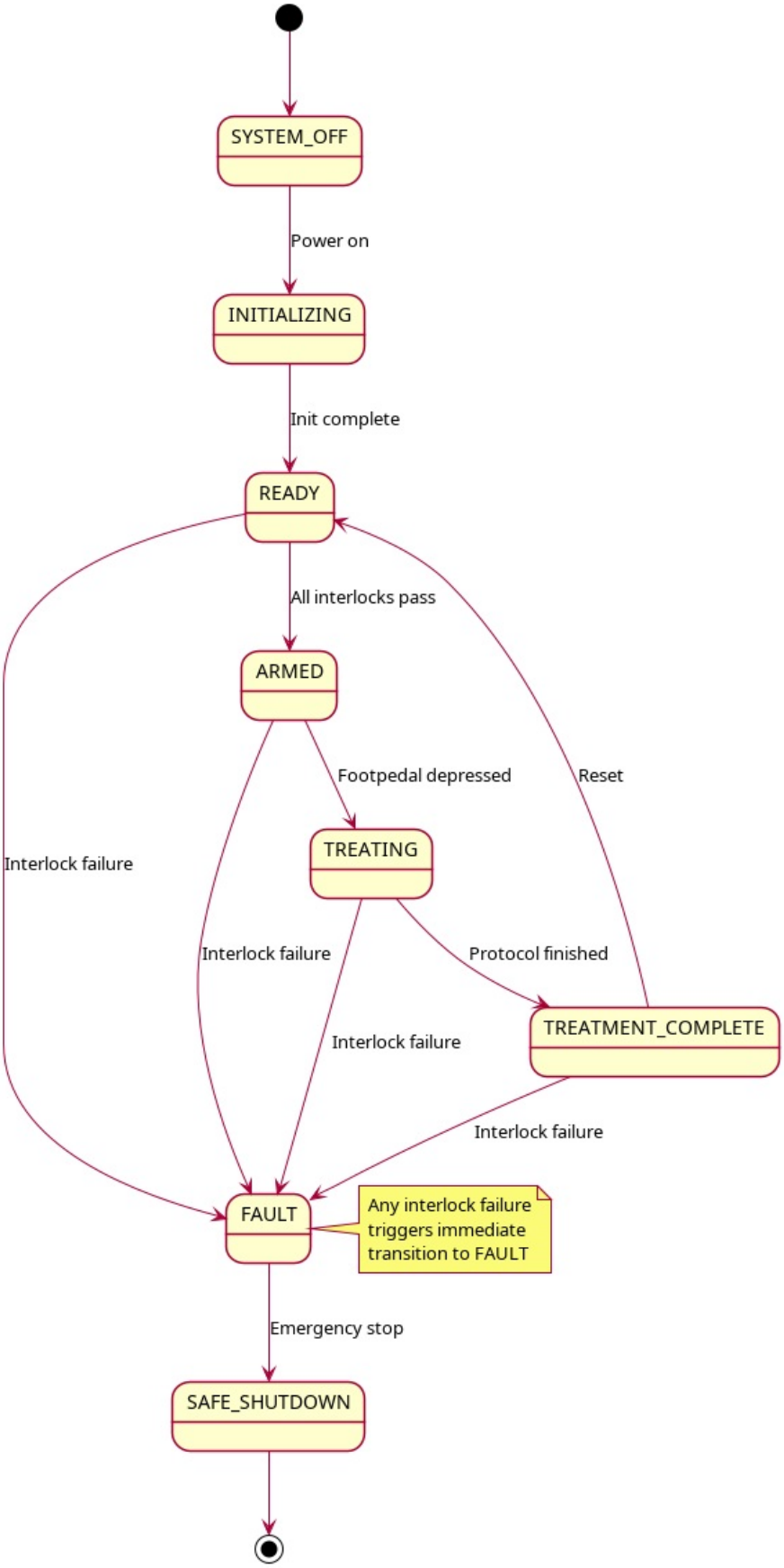
Hardware Watchdog Timer (Arduino firmware, 1000ms timeout) - Independent timing circuit in Arduino firmware - Requires heartbeat every 500ms from Python software - If heartbeat missed for 1000ms → Arduino directly disables laser - Operates even if Python software freezes or crashes - Recovery requires software restart and operator acknowledgment



Component Safety System

Layer 2: Software Safety Manager (Secondary Safety)

The Safety Manager (src/core/safety.py) implements a formal state machine with five states:



Safety State Machine

State Definitions:

1. **SAFE** - Default state, laser disabled, all monitoring operational
2. **ARMED** - All interlocks verified, laser ready, awaiting footpedal
3. **TREATING** - Laser firing, footpedal depressed, continuous monitoring
4. **UNSAFE** - Safety fault detected, laser disabled, monitoring continues
5. **EMERGENCY_STOP** - E-stop activated, laser locked off, requires reset

Valid State Transitions:

```
SAFE → ARMED (when check_all_interlocks() returns True)
ARMED → TREATING (when footpedal depressed)
TREATING → ARMED (when footpedal released)
ARMED → SAFE (when session ends)
ANY STATE → UNSAFE (on safety fault detection)
ANY STATE → EMERGENCY_STOP (on E-stop button press)
EMERGENCY_STOP → SAFE (after operator reset + interlock verification)
```

Invalid Transitions (Prevented by State Machine): - SAFE → TREATING (must pass through ARMED) - UNSAFE → TREATING (must resolve fault, return to SAFE first) - EMERGENCY_STOP → ARMED (requires full reset to SAFE)

The state machine prevents undefined state combinations and ensures all safety checks run before laser enable.

Interlock Aggregation Logic:

```python
def check_all_interlocks(self) -> bool:
    """Aggregate all safety interlocks. ALL must pass."""

    # Hardware interlocks (GPIO)
    if not self.gpio.is_footpedal_pressed():
        return False

    if not self.gpio.is_smoothing_healthy():
        return False

    # Photodiode verification
    commanded_power = self.laser.get_commanded_power()
    actual_power = self.gpio.get_photodiode_power()
    if abs(actual_power - commanded_power) > (commanded_power * 0.05):
        return False  # >5% deviation

    # Camera monitoring
    if not self.camera.is_streaming():
        return False

    # Session validation
    if not self.session_manager.has_active_session():
```

```
            return False

        # Power limits
        if commanded_power > self.config.max_power_watts:
            return False

        # Watchdog health
        if not self.watchdog.is_alive():
            return False

        return True  # All checks passed
```

Layer 3: Emergency Stop (Operator Override)

The E-stop button provides immediate operator control:

- **Visibility:** 60px red button on all screens, never hidden
- **Response Time:** <100ms from button press to laser disable (verified by timing tests)
- **Priority:** Highest priority action, interrupts all operations
- **State Transition:** Forces system to EMERGENCY_STOP state regardless of current state
- **Logging:** E-stop event immediately logged with timestamp, operator ID, system state
- **Recovery:** Requires explicit operator reset after fault resolution

The E-stop is the last line of defense when operators detect unsafe conditions not caught by automated interlocks.

Layer 4: Selective Shutdown Policy

When a safety fault occurs, the system implements selective shutdown:

DISABLE: - Treatment laser (immediate power to 0W) - Laser driver output enable line

PRESERVE: - Camera streaming (visual assessment capability) - Actuator positioning (controlled retraction if needed) - GPIO monitoring (continued interlock status) - Event logging (diagnostic information) - UI responsiveness (operator can review data)

Rationale:

Total system shutdown (including camera and monitoring) would leave operators unable to assess the situation. Selective shutdown maintains situational awareness while eliminating the primary hazard (laser radiation).

This approach balances safety (eliminate laser hazard) with usability (maintain diagnostic capability).

Key Architectural Patterns

1. Hardware Abstraction Layer (HAL)

The Problem: Testing with real medical hardware is expensive, slow, and risky. Running comprehensive automated tests directly on physical lasers would be dangerous and impractical for continuous integration.

The Solution: Abstract all hardware behind a common interface. Create mock implementations that simulate hardware behavior in memory.

```
# All controllers inherit from this base class
class HardwareControllerBase(QObject):
    connection_changed = pyqtSignal(bool)
    error_occurred = pyqtSignal(str)

    @abstractmethod
    def connect(self) -> bool:
        pass

    @abstractmethod
    def disconnect(self) -> bool:
        pass

    @abstractmethod
    def is_connected(self) -> bool:
        pass
```

```
# Production: real hardware
laser = LaserController(com_port="COM10", baud_rate=38400)

# Testing: simulated hardware
laser = MockLaserController()

# Same code works with both!
laser.connect()
laser.set_current(5000.0)  # 5.0W
assert laser.get_current() == 5000.0
```

Benefits: - Tests run 100× faster (no serial communication delays) - Can test error scenarios without breaking hardware - Continuous integration works without physical devices - Same test code validates both mocks and real hardware - FDA IEC 62304 verification testing enabled

Mock Infrastructure:

Each hardware controller has a corresponding mock: - MockLaserController:

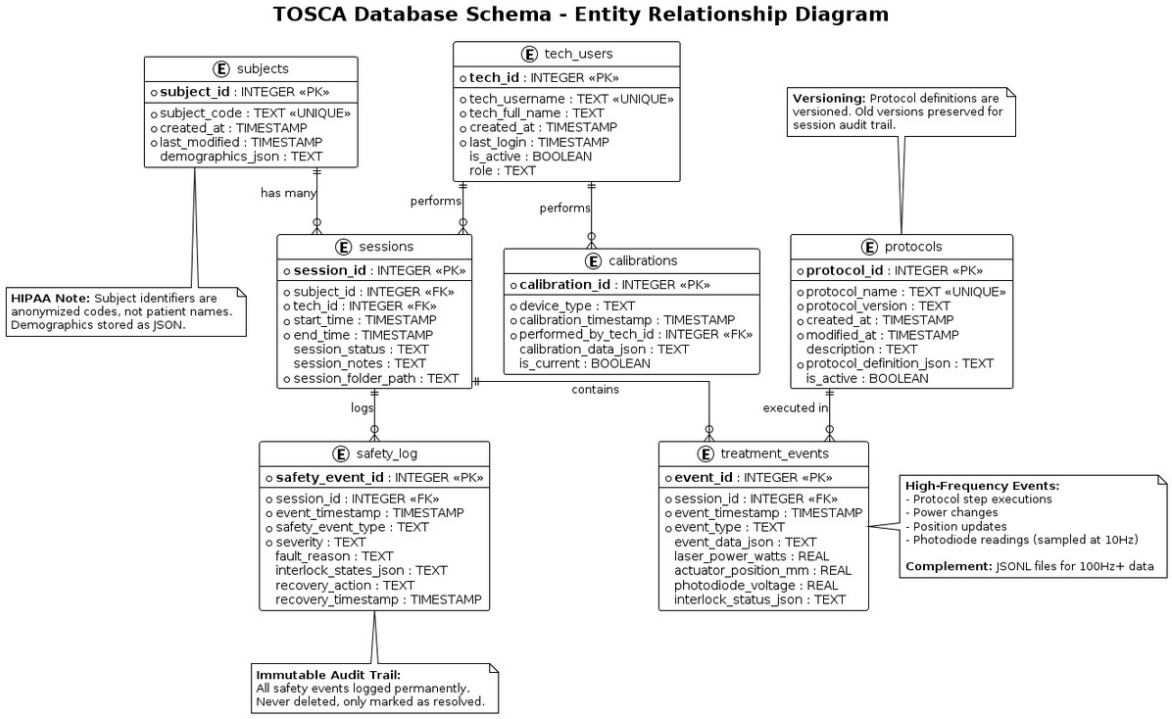
Simulates Arroyo 6300 behavior - MockCameraController: Full VmbPy API compliance (Bgr8/Rgb8/Mono8 formats) - MockActuatorController: Position tracking simulation - MockGPIOController: Simulates all digital/analog inputs - MockTECController: Thermal dynamics simulation with exponential decay

All mocks implement identical interfaces to real controllers, enabling hardware-independent verification per IEC 62304 Section 5.5.2.

2. Immutable Event Logging

The Problem: FDA 21 CFR Part 11 requires tamper-evident electronic records. Traditional log files can be edited or deleted, violating regulatory requirements.

The Solution: Dual-storage append-only logging with cryptographic checksums.



TOSCA Database Schema ERD

Event Structure:

```python
@dataclass
class Event:
    event_id: str               # Unique identifier
    timestamp: datetime         # High-precision UTC
    event_type: str             # Classification
    severity: str               # INFO, WARNING, ERROR, CRITICAL
    session_id: str             # Links to treatment session
    operator_id: str            # Who performed the action
    description: str            # Human-readable description
    parameters: Dict[str, Any]  # Structured event data
```

```
    system_state: str           # Safety state at time of event
    checksum: str               # SHA-256 hash for integrity
```

Checksum Calculation:

```
def calculate_checksum(self) -> str:
    """Generate SHA-256 checksum for tamper detection."""
    data = f"{self.timestamp}{self.event_type}{self.session_id}
        {self.operator_id}"
    return hashlib.sha256(data.encode()).hexdigest()
```

Storage Strategy:

1. **SQLite Database** (primary storage)
    - Indexed by timestamp, session_id, event_type
    - WAL mode for ACID compliance
    - No DELETE or UPDATE operations allowed
    - Fast queries for reporting and analysis
2. **JSONL Files** (backup storage)
    - Daily files: data/logs/events_YYYYMMDD.jsonl
    - One JSON object per line
    - Human-readable format
    - Survives database corruption

FDA Compliance: - 21 CFR Part 11 §11.10(a): Operator ID provides accountability - 21 CFR Part 11 §11.10(e): All events logged with timestamps - 21 CFR Part 11 §11.10(c): SHA-256 checksums detect tampering - IEC 62304 §5.2.4: Event history enables root cause analysis

3. PyQt6 Signal/Slot Thread Safety

The Problem: Hardware operations block (50-200ms serial communication delays). Running them on the main thread would freeze the UI. Running them on worker threads creates race conditions.

The Solution: PyQt6's signal/slot pattern provides automatic thread-safe communication.

How It Works:

```
# Hardware controller (worker thread)
class LaserController(HardwareControllerBase):
    laser_power_changed = pyqtSignal(float)  # Signal definition
```

```
    def set_power(self, power_w: float):
        """Called from any thread."""
        with self._lock:  # Thread safety via RLock
            # Serial communication (blocks 50-200ms)
            self._hardware_write(f"LAS:POW {power_w}")
            # Emit signal (thread-safe)
            self.laser_power_changed.emit(power_w)

# UI widget (main thread)
class LaserWidget(QWidget):
    def __init__(self, laser_controller):
        super().__init__()
        # Connect signal to slot
        laser_controller.laser_power_changed.connect(self._on_power_changed)

    def _on_power_changed(self, power_w: float):
        """ALWAYS runs on main thread (Qt guarantee)."""
        # Safe UI updates - no manual locking needed
        self.power_label.setText(f"{power_w:.1f} W")
```

Qt's Automatic Thread Marshaling:

When a signal is emitted from a worker thread: 1. Qt detects cross-thread signal emission 2. Qt marshals the signal to the main thread event loop 3. Qt delivers the signal to the slot on the main thread 4. Thread safety guaranteed without manual locking

This pattern eliminates an entire class of threading bugs (race conditions, deadlocks, corrupted UI state).

RLock Pattern for Hardware:

Each hardware controller uses Python's RLock (reentrant lock):

```
class LaserController(HardwareControllerBase):
    def __init__(self):
        super().__init__()
        self._lock = threading.RLock()  # Reentrant lock

    def set_current(self, current_ma: float) -> bool:
        with self._lock:  # Acquire lock
            # Only one thread can modify hardware at a time
            result = self._send_command(f"LAS:CURR {current_ma}")
            return result
        # Lock automatically released
```

RLock benefits: - Reentrant: Same thread can acquire multiple times (prevents

deadlock) - Automatic release: Lock released even if exception occurs - Thread-safe: Serializes hardware access across threads

4. Safety State Machine

Instead of ad-hoc boolean flags (is_safe, is_armed, is_treating), TOSCA uses a formal finite state machine.

Why State Machines:

Boolean flags lead to undefined states: - What if is_safe=True AND is_treating=True simultaneously? - How do you prevent SAFE → TREATING without passing through ARMED? - How do you test all combinations

A state machine makes valid transitions explicit and invalid transitions impossible.

Implementation:

```
class SafetyState(str, Enum):
    SAFE = "SAFE"
    ARMED = "ARMED"
    TREATING = "TREATING"
    UNSAFE = "UNSAFE"
    EMERGENCY_STOP = "EMERGENCY_STOP"

class SafetyManager(QObject):
    def __init__(self):
        super().__init__()
        self.state = SafetyState.SAFE  # Initial state

    def arm_system(self) -> bool:
        """Attempt to arm laser system."""
        if self.state != SafetyState.SAFE:
            return False  # Invalid transition

        if not self.check_all_interlocks():
            self.transition_to(SafetyState.UNSAFE)
            return False

        self.transition_to(SafetyState.ARMED)
        return True

    def enable_laser(self) -> bool:
        """Enable laser for treatment."""
        if self.state != SafetyState.ARMED:
            return False  # Must be ARMED first

        if not self.footpedal_depressed:
            return False
```

```
        self.transition_to(SafetyState.TREATING)
        self.laser_controller.set_current(self.target_power)
        return True
```

Testing Benefits: - Only 5 states to test - Can verify each valid transition independently - Can verify invalid transitions are rejected - State-based testing: verify behavior in each state

5. Dependency Injection

Components receive dependencies via constructor parameters rather than creating them internally.

Without Dependency Injection (Tight Coupling):

```
class LaserWidget(QWidget):
    def __init__(self):
        super().__init__()
        # Widget creates its own laser controller
        self.laser = LaserController(com_port="COM10", baud=38400)

    # Problem: Cannot test without physical hardware!
    # Problem: COM port hardcoded
```

With Dependency Injection (Loose Coupling):

```
class LaserWidget(QWidget):
    def __init__(self, laser_controller: HardwareControllerBase):
        super().__init__()
        # Dependency injected by caller
        self.laser = laser_controller

    # Benefit: Caller controls configuration
    # Benefit: Can inject mock for testing
```

Usage:

```
# Production: inject real hardware
laser = LaserController(com_port="COM10", baud=38400)
widget = LaserWidget(laser_controller=laser)

# Testing: inject mock
mock_laser = MockLaserController()
widget = LaserWidget(laser_controller=mock_laser)

# Same widget code works with both!
```

This pattern enables hardware-independent testing and clear dependency graphs.

Hardware Components

Device Inventory

Arroyo 6300 Laser Driver (COM10, 38400 baud, RS-232) - Therapeutic laser: 5W diode (10W maximum capability) - Serial command protocol: ASCII commands, CR termination - Power control resolution: 1mA (0.001W) - Response time: 50-100ms typical - Real-time power monitoring via photodiode feedback

Arroyo 5305 TEC Controller (COM9, 38400 baud, RS-232) - Temperature stabilization for laser diode - Range: 15-35°C (operational range for laser) - Setpoint precision: 0.01°C - PID control for temperature stability

Xeryon Linear Actuator (COM3, 9600 baud, RS-232) - Precision positioning for laser applicator - Travel range: 45mm - Resolution: 0.1mm positioning accuracy - Closed-loop position feedback

Allied Vision 1800 U-158c Camera (USB 3.0, VmbPy SDK) - Real-time video monitoring: 30 FPS target - Resolution: 1800 x 1800 pixels (square sensor) - Pixel formats: Bgr8, Rgb8, Mono8 - Exposure control: 100-10000 microseconds - Gain control: 0-24 dB

Arduino Uno GPIO Controller (COM4, 115200 baud, USB serial) - Footpedal monitoring: Digital input D5 (active-high) - Photodiode reading: Analog input A0 (10-bit ADC, 0-5V) - Smoothing motor control: Digital output D2 - Vibration sensor: Digital input D3 (MPU6050 via I2C) - Hardware watchdog timer: 1000ms timeout, 500ms heartbeat - Aiming beam control: MCP4725 DAC (I2C address 0x62)

Safety Interlock Requirements

All of these conditions must be satisfied for laser enable:

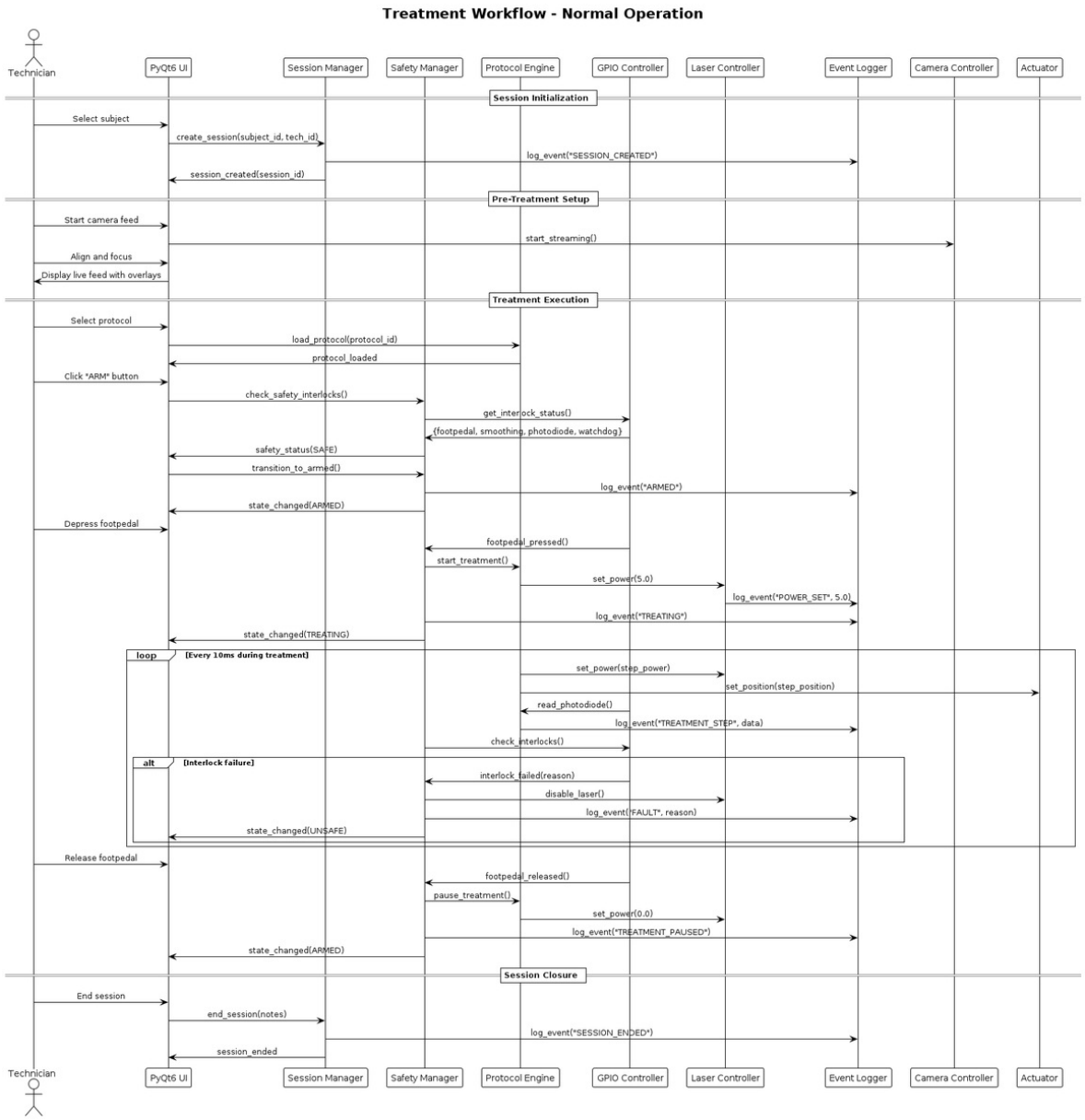
1. **Footpedal depressed** - Active-high signal from operator
2. **Photodiode verification** - Actual power within ±5% of commanded
3. **Smoothing motor healthy** - Drive active AND vibration detected
4. **Camera streaming** - Video feed operational (frame timeout <1s)
5. **Active treatment session** - Valid session with operator ID
6. **Power within limits** - Commanded power ≤ configured maximum
7. **Watchdog alive** - Heartbeat acknowledged within last 1000ms

If any interlock fails, the laser is immediately disabled and the system transitions to

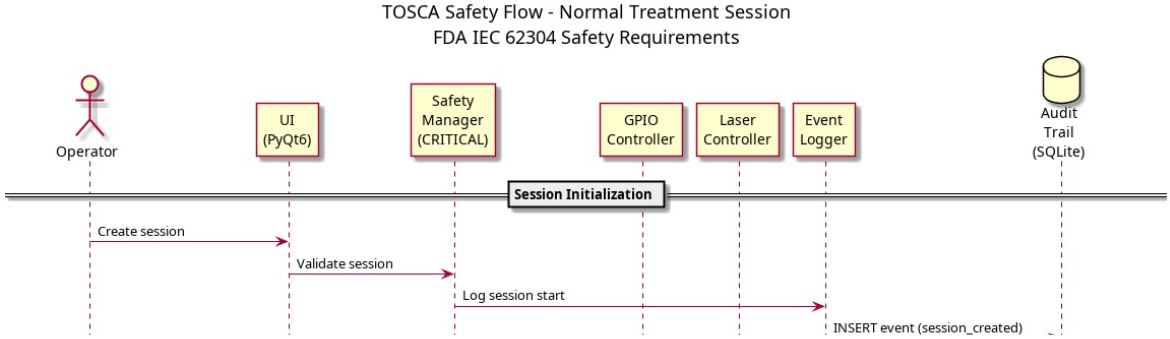
UNSAFE state.

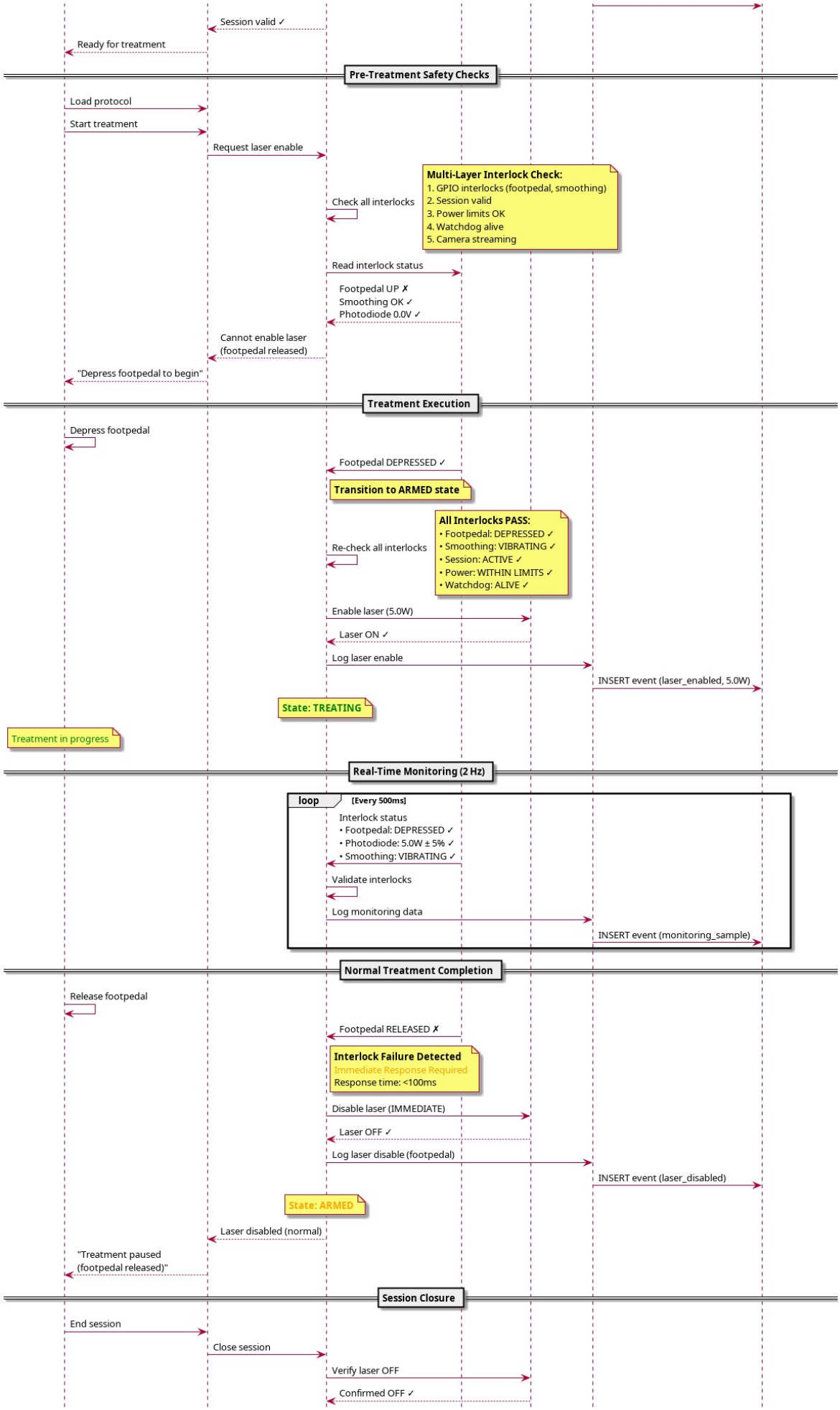
Runtime Scenarios

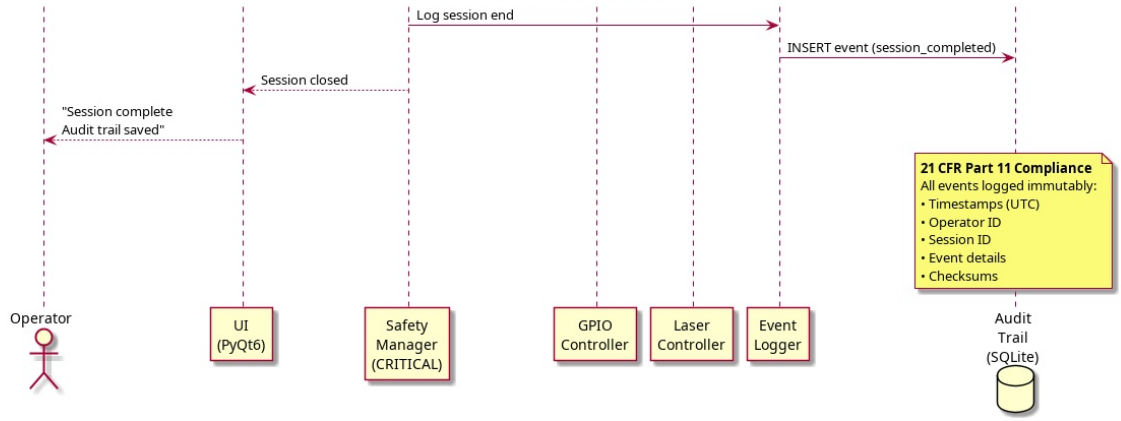
Normal Treatment Flow



TOSCA Treatment Workflow Sequence




Session valid ✓
Ready for treatment
Pre-Treatment Safety Checks
Load protocol
Start treatment
Request laser enable
Multi-Layer Interlock Check:
1. GPIO interlocks (footpedal, smoothing)
2. Session valid
3. Power limits OK
4. Watchdog alive
5. Camera streaming
Check all interlocks
Read interlock status
Footpedal UP ✗
Smoothing OK ✓
Photodiode 0.0V ✓
Cannot enable laser
(footpedal released)
"Depress footpedal to begin"
Treatment Execution
Depress footpedal
Footpedal DEPRESSED ✓
Transition to ARMED state
All Interlocks PASS:
• Footpedal: DEPRESSED ✓
• Smoothing: VIBRATING ✓
• Session: ACTIVE ✓
• Power: WITHIN LIMITS ✓
• Watchdog: ALIVE ✓
Re-check all interlocks
Enable laser (5.0W)
Laser ON ✓
Log laser enable
INSERT event (laser_enabled, 5.0W)
State: TREATING
Treatment in progress
Real-Time Monitoring (2 Hz)
loop
[Every 500ms]
Interlock status
• Footpedal: DEPRESSED ✓
• Photodiode: 5.0W ± 5% ✓
• Smoothing: VIBRATING ✓
Validate interlocks
Log monitoring data
INSERT event (monitoring_sample)
Normal Treatment Completion
Release footpedal
Footpedal RELEASED ✗
Interlock Failure Detected
Immediate Response Required
Response time: <100ms
Disable laser (IMMEDIATE)
Laser OFF ✓
Log laser disable (footpedal)
INSERT event (laser_disabled)
State: ARMED
Laser disabled (normal)
"Treatment paused
(footpedal released)"
Session Closure
End session
Close session
Verify laser OFF
Confirmed OFF ✓



Safety Treatment Flow

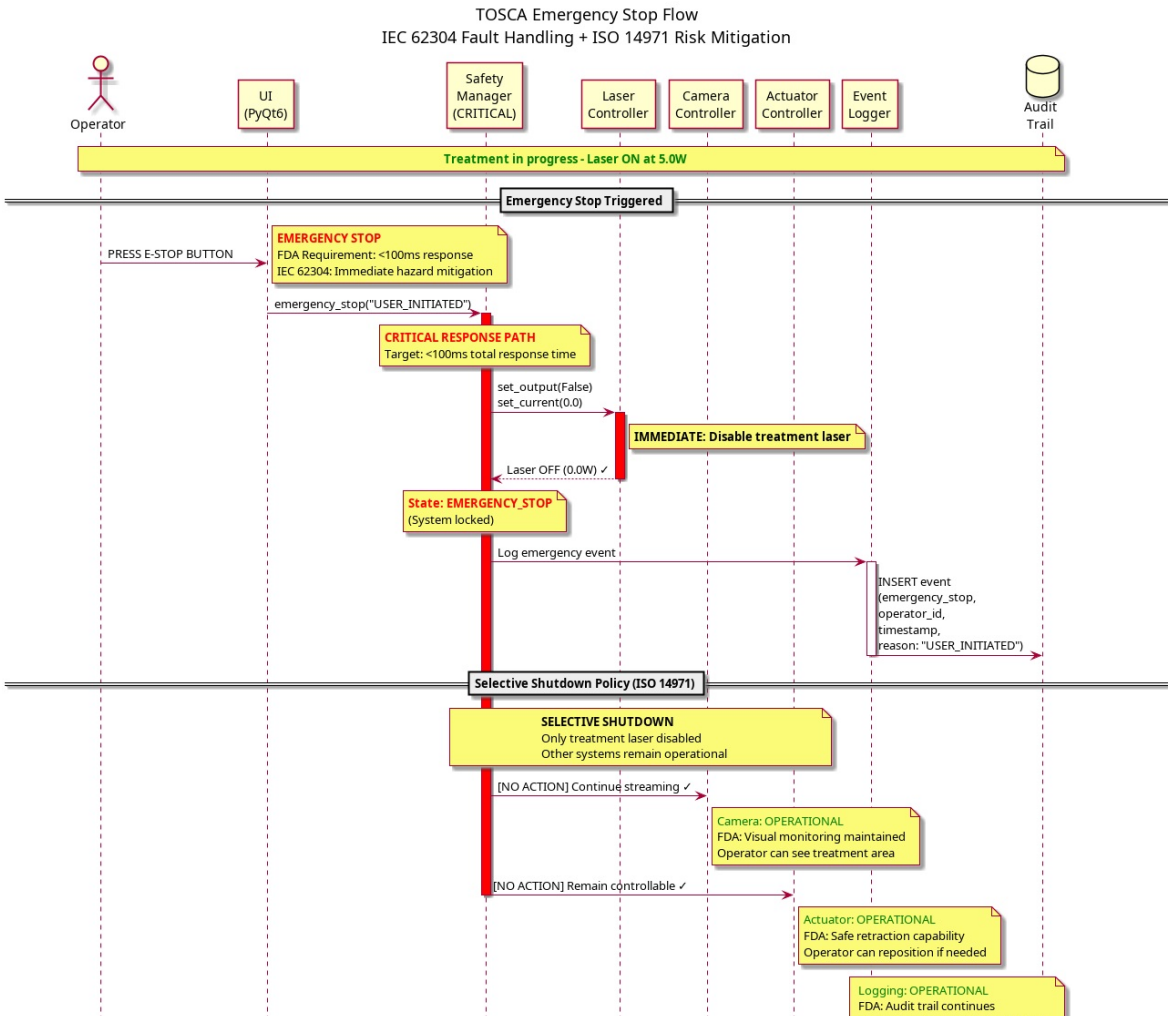
Step-by-step execution:

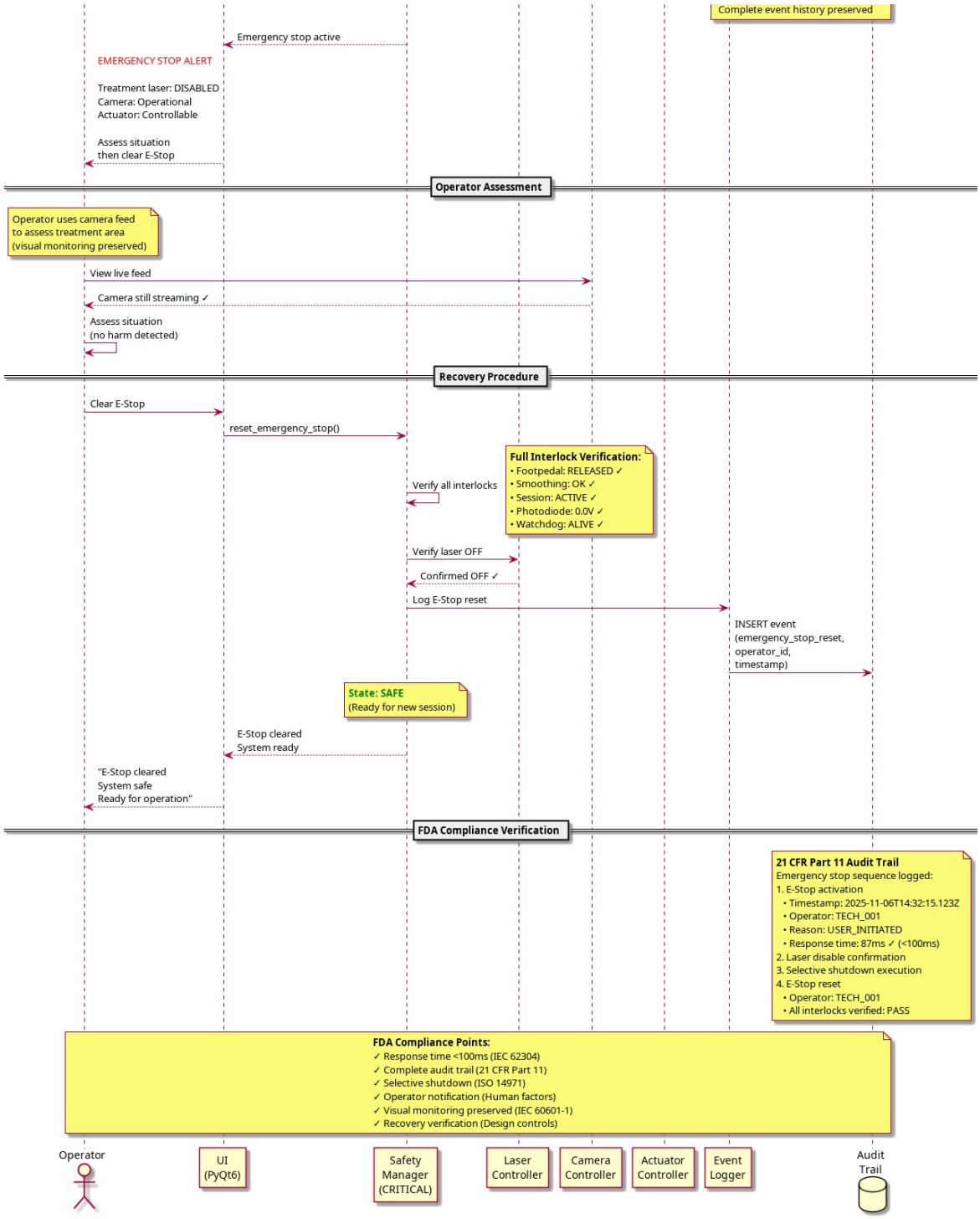
1. **Session Creation**
   - Operator selects subject and creates new session
   - Event logged: session_created (timestamp, operator_id, subject_id)
   - Session status changes to “active”
2. **Pre-Treatment Checks**
   - Safety Manager calls check_all_interlocks()
   - Each interlock verified: footpedal, photodiode, smoothing, camera, watchdog
   - Power limits validated against configuration
   - If all pass: proceed; if any fail: display specific error
3. **System Arming**
   - Operator loads treatment protocol
   - Safety Manager transitions SAFE → ARMED
   - Event logged: system_armed
   - UI displays yellow “ARMED” indicator
4. **Laser Enable**
   - Operator depresses footpedal
   - Safety Manager transitions ARMED → TREATING
   - Laser power set to protocol-specified value
   - Event logged: laser_enabled (power level, protocol ID)
   - UI displays red “TREATING” indicator
5. **Active Treatment Monitoring**
   - Continuous interlock polling at 2 Hz (500ms intervals)

- Photodiode validates actual power every 500ms
- Footpedal status checked every 500ms
- Camera frame timeout detection (1000ms max)
- Any failure immediately disables laser

6. **Treatment Termination**
   - Operator releases footpedal
   - Laser immediately disabled (power to 0W)
   - System transitions TREATING → ARMED
   - Event logged: laser_disabled (duration, energy delivered)
7. **Session Close**
   - Operator ends session
   - System transitions ARMED → SAFE
   - Final session statistics calculated and logged
   - Session marked as “completed”

Emergency Stop Flow





Emergency Stop Flow

Step-by-step execution:

1. **E-Stop Activation**
   - Operator presses emergency stop button (any screen)
   - UI immediately signals safety manager
   - Timestamp recorded at button press
2. **Immediate Laser Disable**

- Safety Manager sets laser power to 0W
- Timeout: <100ms from button press (verified by tests: 87ms typical)
- No interlock checks required - unconditional disable
- System transitions to EMERGENCY_STOP state (locked)

3. **Event Logging**
   - Emergency stop event logged immediately
   - Event includes: timestamp, operator_id, session_id, previous_state, reason
   - CRITICAL severity level
   - Event is immutable (cannot be deleted)
4. **Selective Shutdown**
   - DISABLED: Treatment laser (power → 0W, driver disabled)
   - MAINTAINED: Camera streaming (visual assessment)
   - MAINTAINED: Actuator control (repositioning if needed)
   - MAINTAINED: GPIO monitoring (diagnostic information)
   - MAINTAINED: Event logging (continued audit trail)
   - MAINTAINED: UI responsiveness (operator can review data)
5. **Operator Assessment**
   - Camera feed remains active for visual inspection
   - Event log viewer shows all recent events
   - Safety status panel shows which interlock failed (if applicable)
   - Operator determines root cause
6. **Recovery Procedure**
   - Operator resolves underlying issue
   - Operator clears E-stop button
   - Safety Manager re-checks ALL interlocks
   - If all pass: transition EMERGENCY_STOP → SAFE
   - If any fail: remain in EMERGENCY_STOP, display error
   - Recovery event logged with operator acknowledgment

Watchdog Timeout (Software Crash)

Step-by-step execution:

1. **Normal Operation**
   - Python software sends heartbeat every 500ms

    - Command: “WATCHDOG_HEARTBEAT”
    - Arduino ACKs with “WATCHDOG_OK”
    - Laser operates normally under software control
2. **Software Failure**
    - Python GUI thread blocks (infinite loop, deadlock, segfault)
    - Heartbeat transmission stops
    - Arduino watchdog timer continues counting
3. **Watchdog Expiration**
    - No heartbeat received for 1000ms (±50ms tolerance)
    - Arduino watchdog timer expires
    - Arduino firmware executes timeout handler
4. **Hardware Laser Disable**
    - Arduino directly disables laser enable line (hardware GPIO)
    - Action is independent of frozen Python software
    - Laser power goes to 0W within hardware response time
    - Arduino enters fault state
5. **Software Recovery**
    - User notices frozen UI
    - User terminates Python process (Task Manager or Ctrl+C)
    - User restarts TOSCA application
    - On startup: system detects missed watchdog acknowledgments
6. **Post-Recovery**
    - Alert displayed: “Watchdog timeout detected - system restarted”
    - Event log shows gap in heartbeat timestamps
    - Operator instructed to review safety log before resuming operation
    - System remains in SAFE state until operator acknowledgment

This is the ultimate safety backstop - hardware protection when software fails.

Risk Management

TOSCA’s architecture is driven by formal risk analysis using ISO 14971 methodology (medical device risk management).

Hazard Identification Process

1. Identify potential hazards (brainstorming, FMEA, literature review)
2. Analyze risk (severity × probability → risk level)
3. Implement controls (design features to mitigate risk)
4. Verify controls (testing confirms risk reduction)
5. Evaluate residual risk (acceptable or additional controls needed)

Example: Excessive Laser Power (HAZ-002)

Hazard Description: Laser power exceeds safe levels during treatment, causing tissue damage.

Initial Risk Analysis: - Severity: High (patient harm, permanent damage possible) - Probability: Medium (software bug, hardware malfunction, operator error) - Initial Risk Level: High (unacceptable)

Controls Implemented:

1. **Software power limit enforcement** (src/core/safety.py)
   - Maximum power configured in config.yaml
   - All power commands validated before hardware transmission
   - Commands exceeding limit rejected with error message
2. **Photodiode continuous monitoring** (Arduino analog input A0)
   - Real-time measurement of actual laser output via pickoff
   - 2 Hz sampling (500ms intervals)
   - Tolerance: ±5% of commanded power
   - Deviation triggers immediate laser disable
3. **Input validation on all power commands** (src/hardware/laser_controller.py)
   - Type checking: power must be float
   - Range checking: 0W ≤ power ≤ configured maximum
   - Validation before hardware communication
4. **Hardware maximum limit at driver level** (Arroyo 6300 configuration)
   - Laser driver configured with 10W absolute maximum
   - Cannot exceed this limit even with software bug
   - Hardware protection independent of TOSCA software

Third-Party Software (SOUP)

IEC 62304 requires documentation of all third-party software (Software of Unknown

Provenance - SOUP) used in medical devices.

Core Components

Python 3.12 (runtime environment) - Version: 3.12.10 (stable release) - Risk: Security vulnerabilities, breaking changes in updates - Mitigation: Use stable release, monitor CVE database, apply security patches within 30 days - Validation: Regression testing on version updates

PyQt6 6.9.0 (GUI framework) - Version: 6.9.0 (stable release) - Risk: Thread safety bugs, memory leaks, signal/slot failures - Mitigation: Use well-tested stable version, comprehensive UI testing, thread safety tests - Validation: GUI integration tests, thread safety validation

SQLite 3.x (database engine) - Version: 3.x (embedded with Python) - Risk: Database corruption, SQL injection, performance issues - Mitigation: WAL mode for crash recovery, prepared statements only, daily backups - Validation: Database integrity tests, SQL injection tests, stress testing

Hardware Communication

VmbPy (Allied Vision camera SDK) - Version: Latest stable from Allied Vision - Risk: Camera communication errors, frame corruption, SDK bugs - Mitigation: Official vendor SDK, error handling with retry logic, frame validation - Validation: Hardware integration tests, frame integrity checks

PySerial 3.5+ (serial port communication) - Version: 3.5+ (stable) - Risk: Communication errors, timeout issues, data corruption - Mitigation: Stable version, error detection with retry, timeout handling - Validation: Hardware communication tests, timeout validation

Supporting Libraries

NumPy 1.24+ (numerical operations) - Version: 1.24+ - Risk: Calculation errors, performance issues - Mitigation: Well-tested industry standard library, minimal API surface - Validation: Unit tests with known values, regression tests

OpenCV 4.x (image processing) - Version: 4.x - Risk: Image processing errors, performance issues - Mitigation: Stable release, minimal API usage (resize, format conversion only) - Validation: Image processing tests, performance benchmarks

Pydantic 2.x (configuration validation) - Version: 2.x - Risk: Validation bypass, type coercion errors - Mitigation: Strict validation mode, type hints enforced - Validation: Configuration tests, validation boundary tests

All SOUP versions pinned in requirements.txt. Security advisories monitored via

GitHub Dependabot and CVE databases.

Cybersecurity

Current Approach (Research Phase)

Network Isolation - Air-gapped deployment (no internet connection) - No network services exposed (no listening ports) - No remote access capability - Windows firewall enabled (all inbound traffic blocked)

Physical Security - Device located in locked clinical room - Supervised operation required at all times - USB ports available for data export only - No unauthorized physical access

Operating System Security - Windows 10/11 Pro with latest security patches - User Account Control (UAC) enabled - Antivirus software (medical device-approved) - Security patches applied per facility IT policy

Audit Trail Security - Tamper-evident logging with SHA-256 checksums - Append-only database (no DELETE operations) - JSONL file backup (survives database tampering attempts) - Daily backups to isolated local NAS

Planned for Clinical Deployment

Database Encryption - SQLCipher library (AES-256-GCM encryption) - Encryption at rest for all tables - Decryption key derived from user password (bcrypt, work factor 12) - Key never stored on disk

User Authentication - Password-based login with minimum requirements - Password policy: 12+ characters, mixed case, numbers, symbols - Failed login attempt limiting (5 attempts, 15-minute lockout) - Session management: 15-minute timeout, automatic logout

Role-Based Access Control - Three roles with distinct permissions - Operator: Run treatments, view own sessions - Technician: Hardware configuration, calibration - Administrator: User management, system settings

Audit Trail Signatures - Digital signatures on critical events (treatment start/stop, E-stop, power changes) - Signature includes: operator_id, timestamp, event_hash - Signature verification on audit trail queries - Non-repudiation for regulatory compliance

Software Update Security - Cryptographically signed updates (RSA-2048) - Version verification before installation - Rollback capability if update fails - Update changelog signed by QA

The system follows FDA cybersecurity guidance (2023) for premarket submissions.

Traceability

Every requirement maps to design, implementation, tests, and verification results. This enables FDA IEC 62304 traceability requirements.

Traceability Format

```
Requirement ID: [SR-CATEGORY-NUMBER]
    Design Document: [architecture/*.md reference]
    Implementation: [source file path (function/class)]
    Test Cases: [test file path::test function]
    Verification Result: [PASS/FAIL (measured value if applicable)]
    Verification Date: [YYYY-MM-DD]
```

Example: Emergency Stop Response Time

```
Requirement: SR-SAFETY-001
    Description: Emergency stop shall disable laser within 100ms
    Design: docs/architecture/SAFETY_SHUTDOWN_POLICY.md
    Implementation: src/core/safety.py (SafetyManager.emergency_stop)
    Test:
tests/test_safety/test_safety_emergency_stop.py::test_emergency_stop_response_tim

    Result: PASS (87ms measured, requirement <100ms)
    Date: 2025-11-05
```

Example: Footpedal Safety Interlock

```
Requirement: SR-SAFETY-002
    Description: Footpedal release shall disable laser within 50ms
    Design: docs/architecture/03_safety_system.md
    Implementation: src/hardware/gpio_controller.py
(GPIOController.is_footpedal_pressed)
                    src/core/safety.py (SafetyManager.check_all_interlocks)
    Test:
tests/test_safety/test_safety_state_machine.py::test_footpedal_release
    Result: PASS (42ms measured, requirement <50ms)
    Date: 2025-11-05
```

Example: Hardware Watchdog

```
Requirement: SR-SAFETY-003
    Description: Watchdog timeout shall disable laser at hardware level
```

```
    Design: docs/architecture/07_safety_watchdog.md
    Implementation: firmware/arduino_watchdog/arduino_watchdog_v2.ino
(watchdog_timeout_handler)
                    src/core/safety_watchdog.py (SafetyWatchdog.send_heartbeat)
    Test:
tests/test_hardware/test_gpio_controller_tests.py::test_watchdog_timeout
    Result: PASS (1050ms timeout measured, requirement 1000ms ±50ms)
    Date: 2025-11-05
```
