Technical Specifications

Qcraft

1. INTRODUCTION

1.1 EXECUTIVE SUMMARY

1.1.1 Brief Overview of the Project

QCraft represents a paradigm shift in quantum computing infrastructure, delivering the first desktop-based, adaptive quantum compiler and error correction platform specifically designed for fault-tolerant quantum computing. The system addresses the critical transition point where quantum computing companies have shifted from targeting physical qubits to logical qubits, with every roadmap now including quantum error correction. QCraft combines cutting-edge reinforcement learning algorithms with graph neural networks to optimize quantum error correction code placement and circuit compilation, while maintaining strict privacy controls that ensure logical circuits never leave the user's desktop environment.

1.1.2 Core Business Problem Being Solved

The quantum computing industry faces a fundamental challenge: quantum error correction only provides exponential suppression of logical error rates if the physical error rate is below a critical threshold. Current solutions suffer from three critical limitations:

- **Resource Overhead Crisis**: Traditional surface codes require nearly 3,000 qubits to protect 12 logical qubits for roughly a million cycles, while advanced qLDPC codes can achieve the same protection with only 288 qubits
- **Privacy and Security Concerns**: Existing cloud-based quantum compilation services expose sensitive logical circuit designs to external systems

- **Static Optimization Approaches**: Current compilers use fixed heuristics that cannot adapt to specific hardware characteristics or learn from execution feedback

1.1.3 Key Stakeholders and Users

Stakeholder Category	Primary Needs	QCraft Value Proposition
Quantum Researchers	Hardware-aware compilation, fault-tolerant circuits	Adaptive RL-based optimization, multi-QEC family support
Enterprise Users	Privacy-preserving workflows, scalable solutions	Local processing, encrypted circuit export
Hardware Providers	Benchmarking tools, co-optimization capabilities	Device abstraction layer, empirical noise profiling

1.1.4 Expected Business Impact and Value Proposition

QCraft delivers measurable improvements across critical quantum computing metrics:

- **Fidelity Enhancement**: Target improvement from ~60-65% to ~80-85% on medium-depth circuits through adaptive optimization
- **Resource Efficiency**: Up to 10x reduction in physical qubit requirements using qLDPC codes with 1/24 logical-to-physical qubit encoding rate
- **Privacy Assurance**: 100% local processing of logical circuits with only obfuscated, encoded circuits exported
- **Adaptive Performance**: Continuous learning and optimization through reinforcement learning feedback loops

1.2 SYSTEM OVERVIEW

1.2.1 Project Context

Business Context and Market Positioning

2024 marked a turning point where quantum computing companies shifted from targeting physical qubits to logical qubits, with companies predicting deployment of real-time QEC capabilities by 2028 at the latest. QCraft positions itself at the forefront of this transition, providing the essential infrastructure needed for practical fault-tolerant quantum computing.

The quantum error correction landscape has experienced significant breakthroughs in 2024, including Google's demonstration of below-threshold surface code memories with a distance-7 code achieving 0.143% ± 0.003% error per cycle and IBM's Nature-published work on qLDPC codes that perform as well as surface codes while requiring only one-tenth of the qubits.

Current System Limitations

Existing quantum compilation and error correction solutions exhibit critical deficiencies:

- **Cloud Dependency**: Most current solutions require uploading logical circuits to cloud services, creating security vulnerabilities
- **Static Optimization**: Fixed heuristic approaches cannot adapt to hardware-specific noise characteristics or learn from execution feedback
- **Limited QEC Support**: Most systems support only surface codes, missing opportunities for more efficient qLDPC implementations
- **Scalability Constraints**: Current methods rely on complex unscalable neural networks such as transformers for circuit optimization

Integration with Existing Enterprise Landscape

QCraft integrates seamlessly with existing quantum computing infrastructure through:

- **Hardware Agnostic Design**: Compatible with IBM, IonQ, Rigetti, and major simulator platforms
- **Standard Interface Compliance**: Supports Qiskit, Cirq, and other major quantum programming frameworks
- **Enterprise Security**: Local processing ensures compliance with corporate security policies
- **Scalable Architecture**: YAML-driven configuration enables integration with existing DevOps workflows

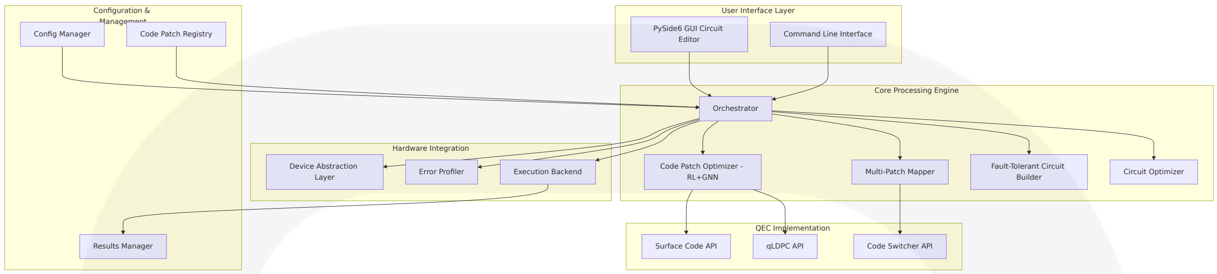
1.2.2 High-Level Description

Primary System Capabilities

QCraft delivers comprehensive quantum circuit compilation and error correction through five core capabilities:

1. **Adaptive QEC Code Selection**: Reinforcement learning algorithms balance exploration and exploitation to discover optimal quantum error correction strategies for specific hardware and circuit combinations
2. **Multi-Family QEC Support**: Native support for surface codes, qLDPC codes (including bivariate bicycle codes), and extensible architecture for future QEC families
3. **Privacy-Preserving Compilation**: Complete local processing with encrypted export of only fault-tolerant, hardware-optimized circuits
4. **Hardware-Aware Optimization**: Device-specific noise modeling and connectivity-aware circuit placement
5. **Continuous Learning**: Deep reinforcement learning with graph neural networks that continuously improve optimization strategies based on execution feedback

Major System Components



Core Technical Approach

QCraft employs a novel hybrid approach combining:

- **Reinforcement Learning Optimization**: PPO-based agents using Graph Neural Networks to approximate policy and value functions for quantum circuit optimization
- **Multi-Objective Reward Functions**: Sophisticated reward shaping balancing circuit fidelity, resource utilization, and hardware constraints
- **Curriculum Learning**: Progressive training from simple structure mastery to complex noise-aware optimization
- **Graph-Based Circuit Representation**: ZX-calculus and graph-theoretic simplification rules for efficient circuit manipulation and optimization

1.2.3 Success Criteria

Measurable Objectives

Metric Category	Target Performance	Measurement Method
Compilation Speed	< 5 seconds for d=3 patches	Automated benchmarking suite
Fidelity Improvement	80-85% on medium-depth circuits	Statistical analysis across test circuits

Metric Category	Target Performance	Measurement Method
Resource Efficiency	10x reduction in physical qubits	Comparative analysis with surface codes

Critical Success Factors

1. **RL Convergence Performance**: Training convergence within 10^5 steps across diverse circuit types
2. **Hardware Adaptability**: Successful deployment across IBM, IonQ, and Rigetti platforms
3. **Privacy Compliance**: Zero logical circuit exposure beyond local machine
4. **Scalability Demonstration**: Support for up to 20 logical qubits with multi-patch configurations

Key Performance Indicators (KPIs)

- **Logical Error Rate (LER) Reduction**: Primary metric for quantum error correction effectiveness
- **Physical Qubit Utilization Efficiency**: Ratio of logical to physical qubits required
- **Compilation Time Performance**: End-to-end processing time for various circuit complexities
- **Hardware Adaptability Score**: Success rate across different quantum hardware platforms
- **User Adoption Metrics**: Desktop application usage patterns and feature utilization

1.3 SCOPE

1.3.1 In-Scope

Core Features and Functionalities

Quantum Error Correction Implementation

- Surface code implementation with distances 3, 5, and 7
- qLDPC code support including bivariate bicycle (BB) codes and hypergraph product codes
- Automated QEC family selection based on circuit and hardware characteristics
- Multi-patch mapping for multiple logical qubits

Reinforcement Learning Optimization

- Maskable Proximal Policy Optimization agents achieving geometric mean improvements of 2.2% over best available approaches
- Graph Neural Network-based policy and value function approximation
- Curriculum learning with progressive complexity stages
- Multi-objective reward function optimization

Desktop Application Features

- PySide6-based GUI providing access to complete Qt 6.0+ framework capabilities
- Drag-and-drop circuit design interface
- Real-time fault-tolerant circuit visualization
- YAML/JSON configuration management

Hardware Integration

- Device abstraction layer supporting IBM, IonQ, Rigetti platforms
- Empirical noise model construction from execution feedback
- Connectivity-aware circuit placement and routing
- Real-time syndrome decoding capabilities

Primary User Workflows

1. **Circuit Design and Compilation**: Logical circuit creation → QEC family selection → Multi-patch mapping → Fault-tolerant encoding → Hardware optimization
2. **Privacy-Preserving Execution**: Local compilation → Encrypted circuit export → Remote execution → Local decoding and analysis
3. **Adaptive Learning**: Execution feedback collection → RL model updates → Performance improvement validation
4. **Multi-Hardware Deployment**: Hardware profile selection → Device-specific optimization → Cross-platform validation

Essential Integrations

- **Quantum Programming Frameworks**: Qiskit, Cirq, PennyLane compatibility
- **Simulation Platforms**: Integration with major quantum simulators
- **Hardware Providers**: Native API support for IBM Quantum, IonQ, Rigetti
- **Development Tools**: Git integration, CI/CD pipeline support

Key Technical Requirements

- **Performance**: Sub-5-second compilation for standard circuits, <10^5 step RL convergence
- **Scalability**: Support for up to 20 logical qubits, linear scaling with qubit count
- **Reliability**: 99.9% uptime for local processing, robust error handling
- **Security**: End-to-end encryption, local-only logical circuit processing

1.3.2 Implementation Boundaries

System Boundaries

Processing Boundaries

- All logical circuit processing occurs locally on user's desktop

- Only fault-tolerant, encoded circuits are exported to external systems
- RL training and model updates performed locally with optional cloud synchronization

Hardware Boundaries

- Support limited to gate-based quantum computers
- Focus on superconducting and trapped-ion platforms
- Exclusion of photonic and topological quantum computing platforms

User Group Coverage

- Primary: Quantum researchers and algorithm developers
- Secondary: Enterprise quantum application developers
- Tertiary: Academic institutions and quantum hardware providers

Geographic/Market Coverage

- Global deployment with localization support for major markets
- Compliance with international quantum technology export regulations
- Multi-language support for GUI and documentation

Data Domains Included

- Quantum circuit representations (logical and physical)
- Hardware device specifications and noise models
- QEC code definitions and patch configurations
- RL training data and model checkpoints
- Execution results and performance metrics

1.3.3 Out-of-Scope

Explicitly Excluded Features/Capabilities

Quantum Hardware Development

- Physical qubit fabrication or control system development

- Low-level pulse sequence generation and optimization
- Quantum hardware calibration and characterization tools

Cloud-Based Processing

- Server-side logical circuit compilation or optimization
- Centralized RL model training or sharing
- Cloud-based quantum circuit simulation services

Alternative Computing Paradigms

- Analog quantum computing support
- Quantum annealing optimization
- Measurement-based quantum computing

Advanced Research Features

- Blind quantum computing protocols (future phase consideration)
- Non-Markovian noise modeling
- Distributed quantum computing architectures

Future Phase Considerations

Phase 2 Enhancements (12-18 months)

- FPGA/ASIC-based local decoder implementation
- Advanced noise models including correlated errors
- Distributed RL training across multiple instances

Phase 3 Extensions (18-24 months)

- Plugin system for custom QEC code families
- Integration with quantum networking protocols
- Advanced blind quantum computing capabilities

Integration Points Not Covered

- Direct integration with quantum cloud services (AWS Braket, Azure Quantum)
- Real-time collaboration features for multi-user circuit design
- Integration with classical HPC clusters for large-scale simulation

Unsupported Use Cases

- Production quantum application deployment (development tool only)
- Real-time control of quantum hardware during execution
- Quantum error correction for non-gate-based quantum computing models
- Commercial quantum algorithm intellectual property protection beyond basic privacy measures

2. PRODUCT REQUIREMENTS

2.1 FEATURE CATALOG

2.1.1 Core Quantum Circuit Processing Features

Feature ID	Feature Name	Category	Priority	Status
F-001	Logical Circuit Editor	User Interface	Critical	Proposed
F-002	Quantum Error Correction Engine	Core Processing	Critical	Proposed
F-003	Multi-Patch Mapping System	Core Processing	Critical	Proposed
F-004	Reinforcement Learning Optimizer	Core Processing	Critical	Proposed

F-001: Logical Circuit Editor

Description

- **Overview**: PySide6-based desktop application providing access to the complete Qt 6.0+ framework for quantum circuit design and visualization
- **Business Value**: Enables intuitive circuit design with real-time fault-tolerant visualization, reducing development time and improving user productivity
- **User Benefits**: Drag-and-drop interface, immediate visual feedback, integrated QEC family selection
- **Technical Context**: Built on PySide6 with custom quantum circuit widgets and MVC architecture

Dependencies

- **Prerequisite Features**: None (foundational feature)
- **System Dependencies**: PySide6, Qt 6.0+, Python 3.9+
- **External Dependencies**: Quantum programming frameworks (Qiskit, Cirq)
- **Integration Requirements**: Config Manager, Results Manager

F-002: Quantum Error Correction Engine

Description

- **Overview**: Multi-family QEC implementation supporting surface codes requiring nearly 3,000 qubits versus qLDPC codes using only 288 qubits for protecting 12 logical qubits
- **Business Value**: Up to 10x reduction in physical qubit requirements using qLDPC codes with 1/24 logical-to-physical qubit encoding rate
- **User Benefits**: Automatic QEC family selection, resource optimization, hardware-aware encoding
- **Technical Context**: Implements surface codes (d=3,5,7) and qLDPC codes including bivariate bicycle codes

Dependencies

- **Prerequisite Features**: F-001 (Logical Circuit Editor)
- **System Dependencies**: Stim, PyMatching, custom QEC libraries
- **External Dependencies**: Hardware device specifications
- **Integration Requirements**: Multi-Patch Mapper, RL Optimizer

F-003: Multi-Patch Mapping System

Description

- **Overview**: Intelligent placement of multiple QEC patches onto quantum hardware topologies with connectivity optimization
- **Business Value**: Enables scaling to 20+ logical qubits with optimal resource utilization
- **User Benefits**: Automated patch placement, hardware constraint satisfaction, connectivity optimization
- **Technical Context**: Graph-based mapping algorithms with hardware topology awareness

Dependencies

- **Prerequisite Features**: F-002 (QEC Engine)
- **System Dependencies**: NetworkX, hardware abstraction layer
- **External Dependencies**: Device connectivity specifications
- **Integration Requirements**: RL Optimizer, Hardware Integration Layer

F-004: Reinforcement Learning Optimizer

Description

- **Overview**: PPO-based agents using Graph Neural Networks to approximate policy and value functions for quantum circuit optimization

- **Business Value**: Geometric mean improvements of 2.2% over best available approaches with reduced additional CNOT gates
- **User Benefits**: Adaptive learning, continuous improvement, hardware-specific optimization
- **Technical Context**: Scales from 5-qubit training circuits to 80-qubit production circuits with up to 10% gate-count reductions

Dependencies

- **Prerequisite Features**: F-002 (QEC Engine), F-003 (Multi-Patch Mapper)
- **System Dependencies**: RLlib/Stable-Baselines3, PyTorch/TensorFlow
- **External Dependencies**: Training data, execution feedback
- **Integration Requirements**: All core processing components

2.1.2 Hardware Integration Features

Feature ID	Feature Name	Category	Priority	Status
F-005	Device Abstraction Layer	Hardware Integration	High	Proposed
F-006	Error Profiler	Hardware Integration	High	Proposed
F-007	Execution Backend	Hardware Integration	High	Proposed
F-008	Syndrome Decoder	Hardware Integration	Medium	Proposed

F-005: Device Abstraction Layer

Description

- **Overview**: Unified interface supporting IBM, IonQ, Rigetti platforms with standardized device specifications

- **Business Value**: Hardware-agnostic deployment reducing vendor lock-in and enabling multi-platform optimization
- **User Benefits**: Seamless hardware switching, consistent interface, automatic device discovery
- **Technical Context**: Plugin-based architecture with standardized device capability APIs

Dependencies

- **Prerequisite Features**: None (foundational feature)
- **System Dependencies**: Hardware provider SDKs
- **External Dependencies**: IBM Quantum, IonQ, Rigetti APIs
- **Integration Requirements**: All hardware-dependent features

F-006: Error Profiler

Description

- **Overview**: Empirical noise model construction from execution feedback to augment provider specifications
- **Business Value**: Improved fidelity through accurate noise characterization and adaptive optimization
- **User Benefits**: Real-time noise tracking, improved error correction, hardware-specific tuning
- **Technical Context**: Statistical analysis of execution results with noise model parameter estimation

Dependencies

- **Prerequisite Features**: F-005 (Device Abstraction Layer), F-007 (Execution Backend)
- **System Dependencies**: Statistical analysis libraries, data persistence
- **External Dependencies**: Execution result data
- **Integration Requirements**: RL Optimizer, QEC Engine

2.1.3 Configuration and Management Features

Feature ID	Feature Name	Category	Priority	Status
F-009	Configuration Manager	System Management	High	Proposed
F-010	Code Patch Registry	System Management	Medium	Proposed
F-011	Results Manager	System Management	Medium	Proposed
F-012	Workflow Orchestrator	System Management	High	Proposed

F-009: Configuration Manager

Description

- **Overview**: YAML/JSON-driven configuration system with schema validation and dynamic parameter management
- **Business Value**: Reproducible experiments, easy parameter tuning, configuration version control
- **User Benefits**: No hardcoded values, easy experimentation, configuration sharing
- **Technical Context**: Schema-based validation with hierarchical configuration inheritance

Dependencies

- **Prerequisite Features**: None (foundational feature)
- **System Dependencies**: YAML/JSON parsers, schema validation libraries
- **External Dependencies**: Configuration files
- **Integration Requirements**: All system components

2.2 FUNCTIONAL REQUIREMENTS TABLE

2.2.1 F-001: Logical Circuit Editor Requirements

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-001-RQ-001	Circuit Design Interface	Drag-and-drop gate placement with real-time validation	Must-Have	Medium
F-001-RQ-002	QEC Family Selection	Toggle between Surface/qLDPC codes with visual feedback	Must-Have	Low
F-001-RQ-003	Circuit Visualization	Real-time fault-tolerant circuit preview	Should-Have	High
F-001-RQ-004	Import/Export Support	Qiskit/Cirq circuit format compatibility	Must-Have	Medium

Technical Specifications

- **Input Parameters**: Gate types, qubit indices, circuit depth, QEC family selection
- **Output/Response**: Visual circuit representation, validation messages, export formats
- **Performance Criteria**: <5ms response time for gate placement, <1s for circuit validation
- **Data Requirements**: Circuit topology, gate parameters, QEC code specifications

Validation Rules

- **Business Rules**: Valid quantum circuits only, supported gate sets, hardware constraints
- **Data Validation**: Qubit index bounds, gate parameter ranges, circuit depth limits
- **Security Requirements**: Local-only circuit processing, no external data transmission
- **Compliance Requirements**: Quantum programming framework compatibility

2.2.2 F-002: Quantum Error Correction Engine Requirements

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-002-RQ-001	Surface Code Implementation	Support distances 3, 5, 7 with <0.5% logical error rate	Must-Have	High
F-002-RQ-002	qLDPC Code Support	Bivariate Bicycle codes with 288 qubits protecting 12 logical qubits	Must-Have	High
F-002-RQ-003	Automatic Family Selection	RL-based selection achieving >95% optimal choice accuracy	Should-Have	High
F-002-RQ-004	Fault-Tolerant Encoding	Gate-level encoding with error propagation analysis	Must-Have	High

Technical Specifications

- **Input Parameters**: Logical circuit, hardware specifications, error rate targets
- **Output/Response**: Encoded fault-tolerant circuit, resource requirements, error estimates

- **Performance Criteria**: <5s encoding time for d=3 patches, <10^-3 logical error rate
- **Data Requirements**: QEC code definitions, stabilizer generators, logical operators

Validation Rules

- **Business Rules**: Below-threshold operation with exponential error suppression
- **Data Validation**: Valid stabilizer codes, distance constraints, hardware compatibility
- **Security Requirements**: Local encoding processing, encrypted circuit export only
- **Compliance Requirements**: Quantum error correction standards

2.2.3 F-004: Reinforcement Learning Optimizer Requirements

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-004-RQ-001	PPO Agent Training	Convergence within 10^5 steps on 20-qubit architectures	Must-Have	High
F-004-RQ-002	Graph Neural Networks	GNN-based policy and value function approximation	Must-Have	High
F-004-RQ-003	Multi-Objective Rewards	Configurable reward function with 10+ optimization objectives	Must-Have	Medium
F-004-RQ-004	Curriculum Learning	Progressive training from structure mastery to noise-aware optimization	Should-Have	High

Technical Specifications

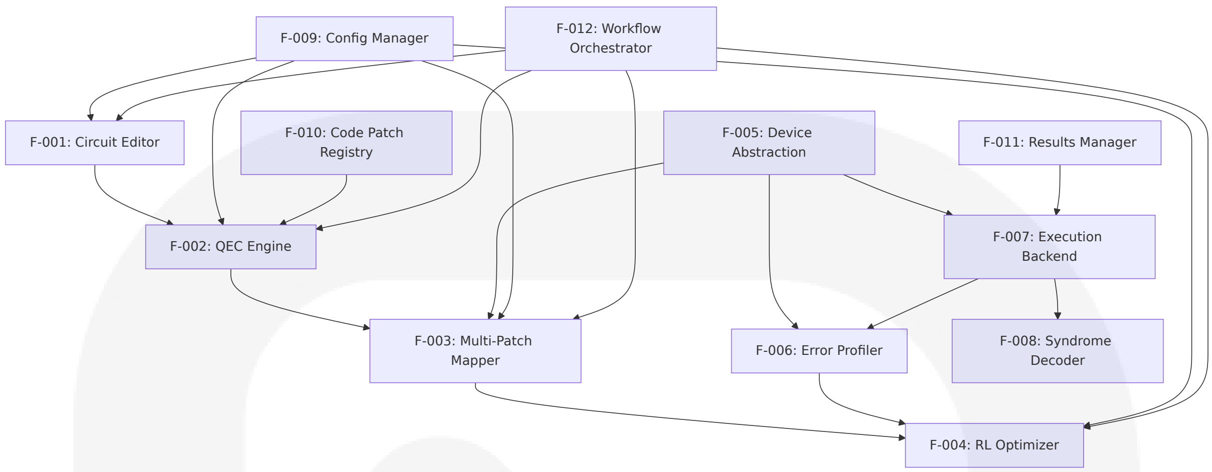
- **Input Parameters**: Circuit graphs, hardware topology, reward weights, training parameters
- **Output/Response**: Optimized mappings, policy updates, performance metrics
- **Performance Criteria**: 2.2% geometric mean improvement over baselines
- **Data Requirements**: Training circuits, hardware profiles, execution feedback

Validation Rules

- **Business Rules**: Monotonic improvement over training, stable convergence
- **Data Validation**: Valid graph structures, reward function bounds, action space constraints
- **Security Requirements**: Local model training, optional encrypted model sharing
- **Compliance Requirements**: Reproducible training with configuration versioning

2.3 FEATURE RELATIONSHIPS

2.3.1 Feature Dependencies Map



2.3.2 Integration Points

Integration Point	Features Involved	Shared Components	Data Exchange
Circuit Processing Pipeline	F-001, F-002, F-003, F-004	Workflow Orchestrator	Circuit representations, optimization parameters
Hardware Interface	F-005, F-006, F-007, F-008	Device Abstraction Layer	Device specifications, execution results
Configuration System	F-009, F-010, F-011, F-012	Config Manager	YAML/JSON configurations, validation schemas
RL Training Loop	F-004, F-006, F-007, F-011	Results Manager	Training data, performance metrics, model updates

2.3.3 Common Services

Service	Description	Dependent Features	Implementation
Graph Processing	Circuit and hardware topology manipulation	F-002, F-003, F-004	NetworkX-based graph operations

Service	Description	Dependent Features	Implementation
Data Persistence	Configuration and results storage	F-009, F-010, F-011	SQLite/JSON file storage
Event System	Inter-component communication	F-001, F-012	Qt signals/slots mechanism
Validation Engine	Schema and constraint validation	F-001, F-002, F-009	JSON Schema validation

2.4 IMPLEMENTATION CONSIDERATIONS

2.4.1 Technical Constraints

Feature	Constraints	Mitigation Strategy
F-001	PySide6 Qt 6.0+ framework dependency	Use official PySide6 packages, maintain compatibility matrix
F-002	Quantum error correction computational complexity	Implement efficient stabilizer algorithms, use sparse matrix representations
F-004	Scalable neural networks avoiding complex transformers	Use Graph Neural Networks with linear scaling properties
F-007	Hardware API rate limits and availability	Implement request queuing, fallback to simulators

2.4.2 Performance Requirements

Feature	Performance Target	Measurement Method	Scaling Considerations
F-001	<5ms gate placement response	UI responsiveness testing	Linear with circuit size

Feature	Performance Target	Measurement Method	Scaling Considerations
F-002	<5s compilation for d=3 patches	Automated benchmarking	Quadratic with distance
F-003	<10s mapping for 20 logical qubits	Algorithm complexity analysis	Polynomial with qubit count
F-004	10^5 step convergence	Training curve analysis	Linear with problem complexity

2.4.3 Scalability Considerations

Aspect	Current Target	Future Scaling	Implementation Strategy
Logical Qubits	20 qubits	100+ qubits	Hierarchical patch organization
Circuit Depth	1000 gates	10,000+ gates	Streaming circuit processing
Hardware Platforms	3 platforms	10+ platforms	Plugin-based architecture
RL Training	Single agent	Multi-agent systems	Distributed training framework

2.4.4 Security Implications

Feature	Security Requirement	Implementation	Validation Method
F-001	Local circuit processing only	No network transmission of logical circuits	Code audit, network monitoring
F-002	Encrypted fault-tolerant export	AES-256 encryption for circuit export	Cryptographic testing
F-004	Model privacy protection	Local training with optional encrypted sharing	Privacy impact assessment

Feature	Security Requirement	Implementation	Validation Method
F-009	Configuration integrity	Digital signatures for configuration files	Signature verification testing

2.4.5 Maintenance Requirements

Feature	Maintenance Aspect	Frequency	Automation Level
F-002	QEC code library updates	Quarterly	Semi-automated with validation
F-004	RL model retraining	Monthly	Fully automated with performance monitoring
F-005	Hardware API compatibility	As needed	Automated testing with CI/CD
F-009	Configuration schema updates	Per release	Version-controlled with migration scripts

3. TECHNOLOGY STACK

3.1 PROGRAMMING LANGUAGES

3.1.1 Primary Language Selection

Component	Language	Version	Justification
Desktop Application	Python	3.9+	PySide6 requires Python 3.9+ and provides access to the complete Qt 6.0+ framework
Core Processing Engine	Python	3.9+	Native integration with quantum computing libraries and RL frameworks

Component	Language	Version	Justification
GUI Frontend	Python	3.9+	PySide6 is the official Python module from the Qt for Python project
Configuration Management	Python	3.9+	YAML/JSON processing and schema validation capabilities

3.1.2 Language Selection Criteria

Python 3.9+ Selection Rationale

- **Quantum Computing Ecosystem**: Qiskit is an open-source SDK for working with quantum computers with 550,000 users and 3 trillion quantum circuits
- **Desktop GUI Framework**: PySide6 provides access to the complete Qt 6.0+ framework with native desktop application capabilities
- **Scientific Computing**: Extensive ecosystem for numerical computing, machine learning, and graph processing
- **Performance Optimization**: Stim's hot loops are heavily vectorized using 256 bit wide AVX instructions, making them very fast with the ability to multiply Pauli strings with 100 billion terms in one second

3.1.3 Platform-Specific Considerations

Platform	Python Distribution	Additional Requirements
Windows	CPython 3.9+	Visual C++ Redistributable for compiled extensions
macOS	CPython 3.9+	Xcode Command Line Tools for native compilation
Linux	CPython 3.9+	Build-essential package for compilation dependencies

3.2 FRAMEWORKS & LIBRARIES

3.2.1 Desktop Application Framework

Framew ork	Version	Purpose	Justification
PySide6	6.9.2	GUI Frame work	Official Python module from the Qt for Python project providing access to complete Qt 6.0+ fra mework
Qt	6.0+	Native UI C omponents	Cross-platform native desktop a pplication development

PySide6 Framework Selection

- **Official Qt Support**: PySide6 is the official Python module from the Qt for Python project developed in the open with all facilities expected from modern OSS projects
- **Comprehensive UI Capabilities**: Full access to Qt's widget system, graphics framework, and native platform integration
- **Cross-Platform Compatibility**: Native look and feel across Windows, macOS, and Linux platforms
- **Professional Desktop Applications**: Enterprise-grade UI framework suitable for complex scientific applications

3.2.2 Quantum Computing Libraries

Library	Version	Purpose	Integration Requirements
Qiskit	2.2.0	Quantum Cir cuit Framew ork	Stable, high-performance relea se with 55% decrease in memo ry usage and 16x faster bindin g and transpiling
Stim	1.14.0	Quantum Err or Correctio n	Fast stabilizer circuit library for high performance simulation a

Library	Version	Purpose	Integration Requirements
			nd analysis of quantum error correction circuits
PyMatching	2.2.1	QEC Decoding	New implementation of blossom algorithm 100-1000x faster than previous versions

Quantum Library Integration Strategy

- **Qiskit Integration**: Updated base primitives v2 interface and native support for OpenQASM 3 for quantum circuit representation and hardware abstraction
- **Stim Performance**: Vectorized code using 256 bit wide AVX instructions enabling multiplication of Pauli strings with 100 billion terms in one second
- **PyMatching Efficiency**: Over 100x faster than previous versions and can decode surface code circuits up to distance 17 in under 1 microsecond per round

3.2.3 Reinforcement Learning Framework

Framework	Version	Purpose	Selection Criteria
Stable-Baselines3	2.7.1a3	RL Algorithms	Ideal for beginners with excellent documentation and strong institutional backing
RLlib	2.9.2+	Distributed RL	Strongest choice for production deployment with superior performance for large-scale applications

RL Framework Selection Rationale

- **Stable-Baselines3**: Set of reliable implementations of reinforcement learning algorithms in PyTorch as the next major version of Stable Baselines

- **RLlib Alternative**: Scalable, distributed reinforcement learning library supporting wide array of algorithms and multi-agent settings
- **Production Readiness**: Strong institutional backing and active maintenance making them safe choices for long-term projects

3.2.4 Graph Processing Libraries

Library	Version	Purpose	Performance Characteristics
NetworkX	3.5	Graph Operations	Python package for creation, manipulation, and study of complex networks
PyTorch	2.0+	Neural Networks	GPU acceleration for Graph Neural Networks
NumPy	1.24+	Numerical Computing	Efficient array operations and linear algebra

Graph Processing Integration

- **NetworkX Core**: Python package for creation, manipulation, and study of structure, dynamics, and functions of complex networks with ability to load, store, generate, and analyze networks
- **Scalability Considerations**: Suitable for operation on large real-world graphs in excess of 10 million nodes and 100 million edges with reasonably efficient, scalable, and portable framework
- **Backend Acceleration**: Support for accelerated backends allowing NetworkX to be both easy to use and fast, incorporating workflows with similar accelerators

3.3 OPEN SOURCE DEPENDENCIES

3.3.1 Core Scientific Computing Stack

Package	Version	Registry	Purpose
numpy	≥1.24.0	PyPI	Numerical computing and array operations
scipy	≥1.10.0	PyPI	Scientific computing and optimization
matplotlib	≥3.6.0	PyPI	Plotting and visualization
pandas	≥2.0.0	PyPI	Data manipulation and analysis

3.3.2 Quantum Computing Dependencies

Package	Version	Registry	Purpose
qiskit	2.2.0	PyPI	Open-source SDK for working with quantum computers at the level of extended quantum circuits, operators, and primitives
stim	1.14.0	PyPI	High performance simulation and analysis of quantum stabilizer circuits, especially quantum error correction circuits
pymatching	2.2.1	PyPI	Package for decoding quantum error correcting codes using minimum-weight perfect matching
cirq	≥1.0.0	PyPI	Google's quantum computing framework for circuit compatibility

3.3.3 Machine Learning Dependencies

Package	Version	Registry	Purpose
stable-baselines3	2.7.1a3	PyPI	Reliable implementations of reinforcement learning algorithms in PyTorch

Package	Version	Registry	Purpose
torch	≥2.0.0	PyPI	Deep learning framework for neural networks
ray[rllib]	≥2.9.2	PyPI	Distributed reinforcement learning library
gymnasium	≥0.28.0	PyPI	Reinforcement learning environment interface

3.3.4 GUI and System Dependencies

Package	Version	Registry	Purpose
PySide6	6.9.2	PyPI	Official Python module from Qt for Python project
networkx	3.5	PyPI	Python package for creation, manipulation, and study of complex networks
pyyaml	≥6.0	PyPI	YAML configuration file processing
jsonschema	≥4.0.0	PyPI	JSON schema validation
cryptography	≥40.0.0	PyPI	Encryption and security functions

3.3.5 Development and Testing Dependencies

Package	Version	Registry	Purpose
pytest	≥7.0.0	PyPI	Testing framework
pytest-qt	≥4.0.0	PyPI	Qt application testing
black	≥23.0.0	PyPI	Code formatting
mypy	≥1.0.0	PyPI	Static type checking

Package	Version	Registry	Purpose
sphinx	≥6.0.0	PyPI	Documentation generation

3.4 THIRD-PARTY SERVICES

3.4.1 Quantum Hardware Integration

Service	Purpose	Authentication	Integration Method
IBM Quantum	Hardware execution	API Token	IBM Quantum Platform credentials
IonQ	Trapped-ion systems	API Key	REST API integration
Rigetti	Superconducting systems	API Key	Forest SDK integration
AWS Braket	Multi-vendor access	AWS Credentials	Boto3 SDK integration

3.4.2 External APIs and Services

Service Category	Service	Purpose	Usage Pattern
Hardware Providers	IBM Quantum Network	Real quantum hardware access	On-demand execution
Cloud Simulators	AWS Braket Simulators	High-performance simulation	Batch processing
Version Control	GitHub API	Configuration synchronization	Optional integration
Telemetry	Local Analytics	Usage metrics (privacy-preserving)	Local-only processing

3.4.3 Security and Privacy Considerations

Data Privacy Requirements

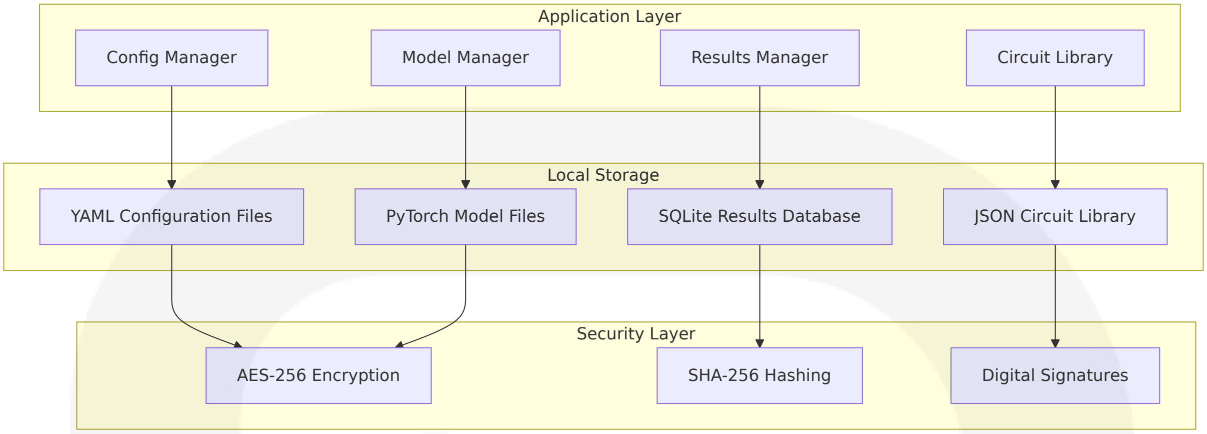
- **Local Processing**: All logical circuits remain on local desktop environment
- **Encrypted Export**: Only fault-tolerant, encoded circuits exported with AES-256 encryption
- **No Cloud Dependencies**: Core functionality operates without external service dependencies
- **Optional Integrations**: Hardware access and cloud services are optional, user-controlled features

3.5 DATABASES & STORAGE

3.5.1 Local Data Persistence

Storage Type	Technology	Purpose	Implementation
Configuration	YAML/JSON Files	User settings and parameters	File-based with schema validation
Results Cache	SQLite	Execution results and metrics	Embedded database
Model Storage	Pickle/PyTorch	RL model checkpoints	Binary serialization
Circuit Library	JSON	Saved quantum circuits	Structured file format

3.5.2 Data Storage Architecture



3.5.3 Storage Requirements and Specifications

Data Type	Storage Format	Encryption	Retention Policy
User Configurations	YAML with JSON Schema	Optional AES-256	User-controlled
RL Training Data	HDF5/Parquet	Not required	Configurable cleanup
Circuit Definitions	JSON with validation	Digital signatures	Permanent
Execution Results	SQLite with indexing	SHA-256 hashing	Configurable retention
Model Checkpoints	PyTorch native format	AES-256 for export	Version-controlled

3.5.4 Caching Strategy

Multi-Level Caching Architecture

- **Memory Cache**: Frequently accessed circuits and configurations
- **Disk Cache**: Compiled quantum circuits and optimization results
- **Model Cache**: Pre-trained RL models and optimization strategies
- **Results Cache**: Hardware execution results and performance metrics

Cache Management

- **LRU Eviction**: Least recently used items removed when cache limits reached
- **Configurable Limits**: User-defined cache sizes and retention periods
- **Cache Validation**: Automatic invalidation when source data changes
- **Performance Monitoring**: Cache hit rates and performance metrics tracking

3.6 DEVELOPMENT & DEPLOYMENT

3.6.1 Development Environment

Tool Category	Tool	Version	Purpose
Package Manager	pip	Latest	Python package installation
Environment Manager	conda/venv	Latest	Isolated development environments
Code Editor	VS Code/PyCharm	Latest	IDE with Python and Qt support
Version Control	Git	2.40+	Source code management

3.6.2 Build System and Packaging

Component	Technology	Configuration	Output
Package Builder	setuptools	setup.py/pyproject.toml	Python wheel distribution
Dependency Manager	pip-tools	requirements.in files	Locked dependency versions
Application Bundler	PyInstaller	spec files	Standalone executables

Component	Technology	Configuration	Output
Installer Creator	NSIS/DMG/DEB	Platform-specific scripts	Native installers

3.6.3 Containerization Strategy

Development Containers

```
FROM python:3.9-slim
RUN apt-get update && apt-get install -y \
    build-essential \
    qt6-base-dev \
    libgl1-mesa-glx \
    && rm -rf /var/lib/apt/lists/*
COPY requirements.txt .
RUN pip install -r requirements.txt
```

Container Usage Patterns

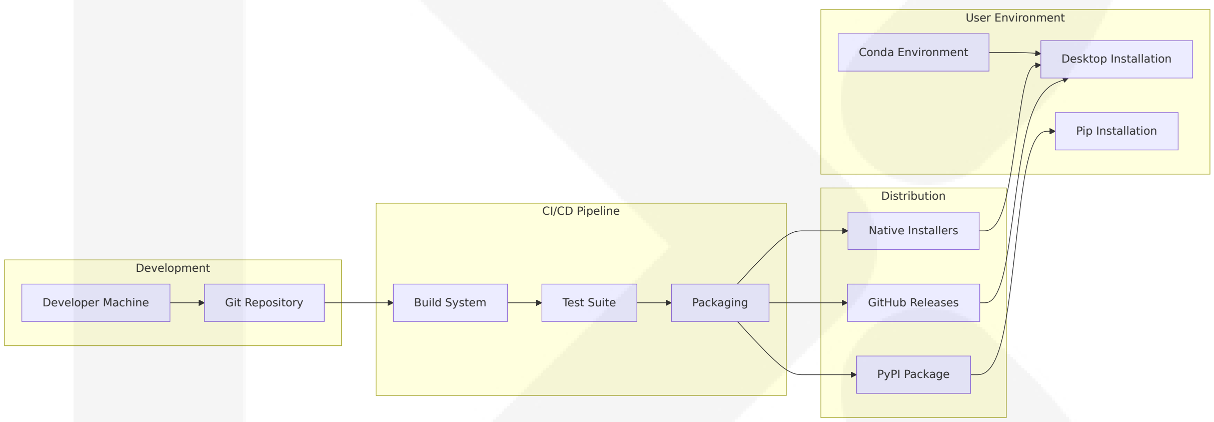
- **Development Environment**: Consistent development setup across team members
- **Testing Environment**: Isolated testing with specific dependency versions
- **CI/CD Pipeline**: Automated testing and building in controlled environments
- **Distribution**: Optional containerized deployment for enterprise environments

3.6.4 Continuous Integration and Deployment

Stage	Tools	Purpose	Triggers
Code Quality	Black, MyPy, Flake8	Code formatting and type checking	Every commit

Stage	Tools	Purpose	Triggers
Testing	pytest, pytest-qt	Unit and integration testing	Pull requests
Security	Bandit, Safety	Security vulnerability scanning	Nightly builds
Documentation	Sphinx	API documentation generation	Release branches
Packaging	PyInstaller	Executable creation	Tagged releases

3.6.5 Deployment Architecture



3.6.6 Platform-Specific Deployment

Platform	Distribution Method	Package Format	Installation Method
Windows	GitHub Releases	.exe installer	NSIS-based installer
macOS	GitHub Releases	.dmg package	Native macOS installer
Linux	PyPI + GitHub	.deb/.rpm packages	Package manager integration
Cross-Platform	PyPI	Python wheel	pip install qcraft

3.6.7 Version Management and Release Strategy

Semantic Versioning

- **Major.Minor.Patch** format (e.g., 1.0.0)
- **Major**: Breaking changes to API or core functionality
- **Minor**: New features with backward compatibility
- **Patch**: Bug fixes and minor improvements

Release Channels

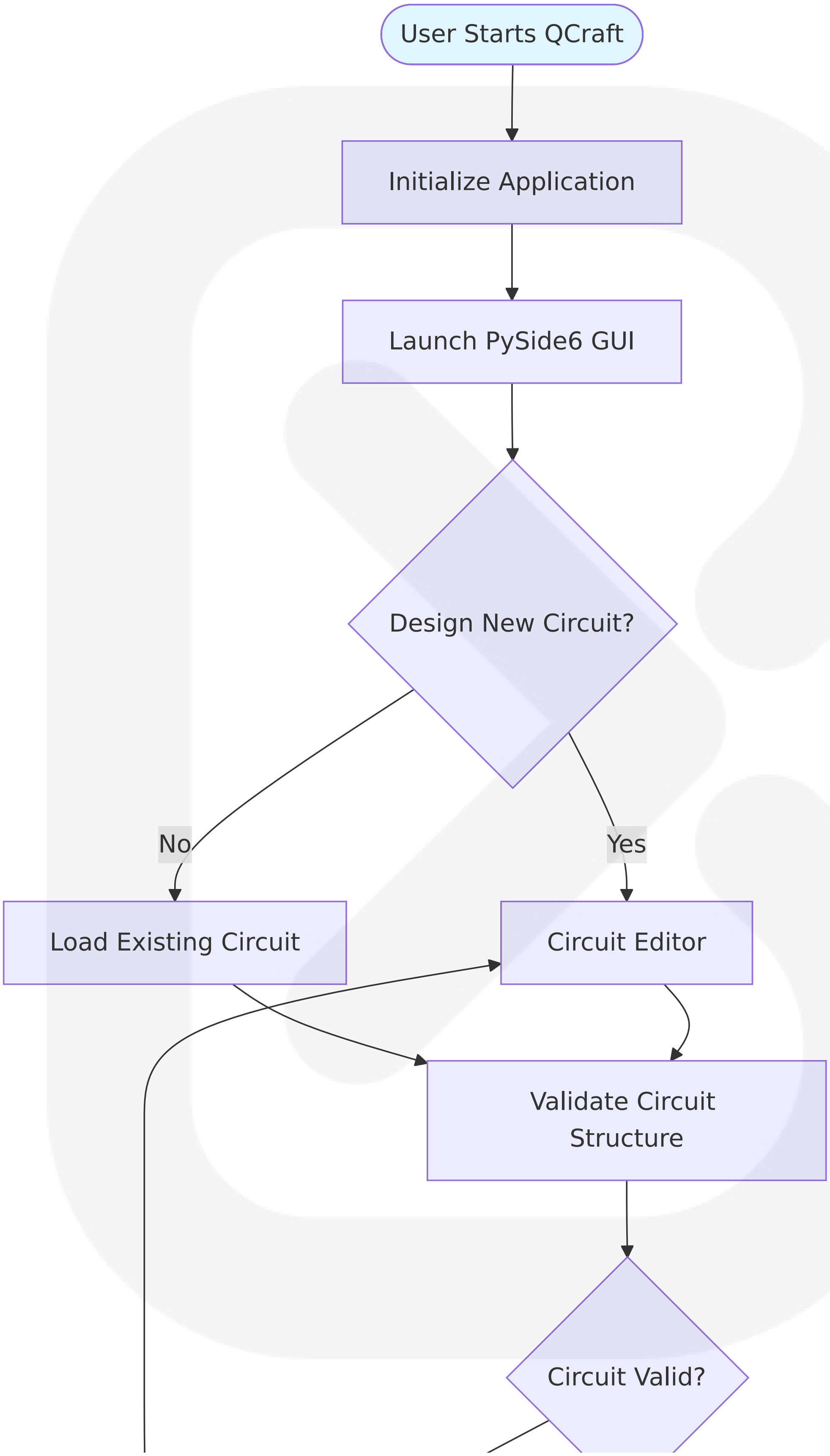
- **Stable**: Thoroughly tested releases for production use
- **Beta**: Feature-complete releases for testing and feedback
- **Alpha**: Development releases for early adopters and contributors
- **Nightly**: Automated builds from main development branch

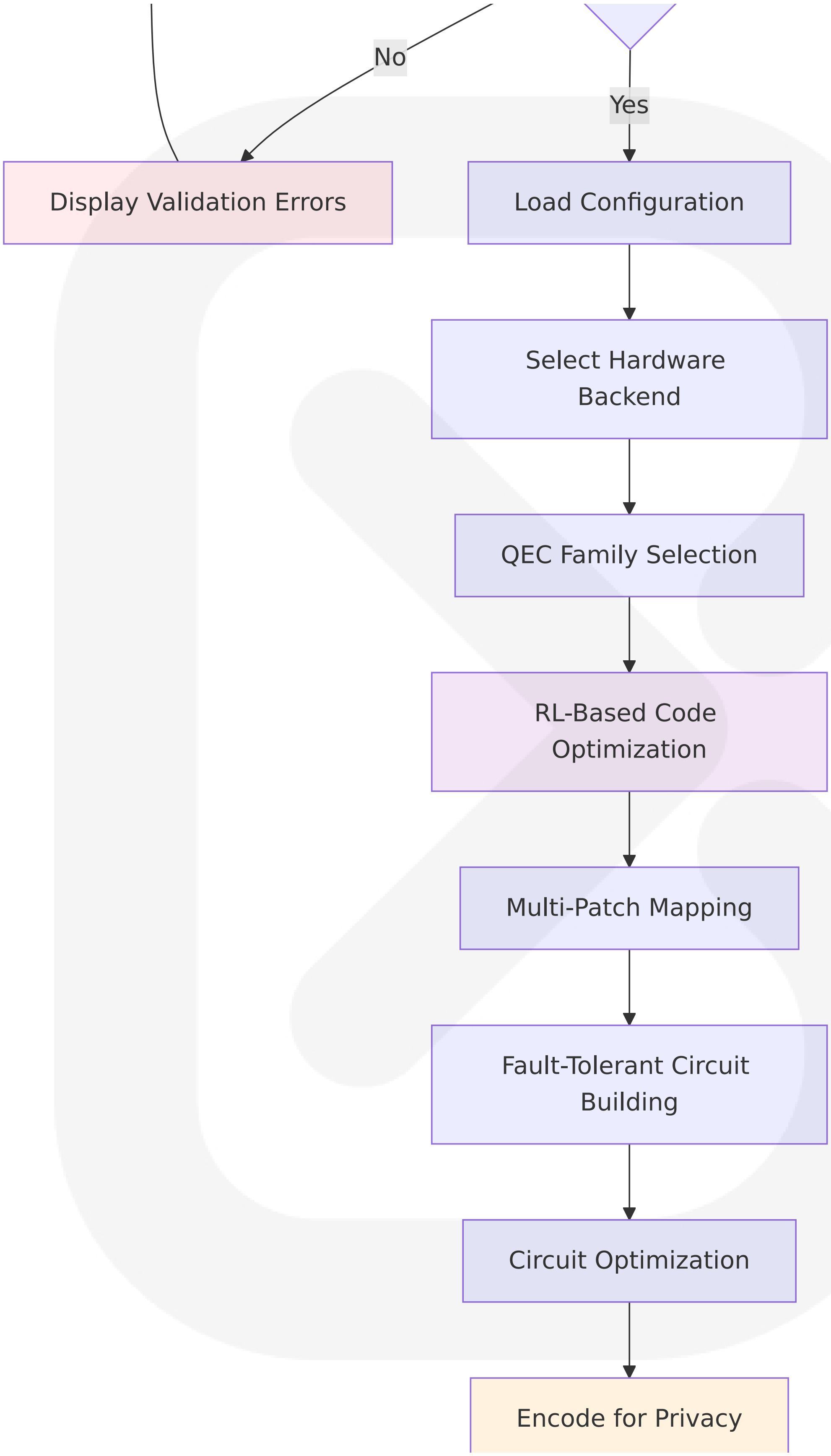
4. PROCESS FLOWCHART

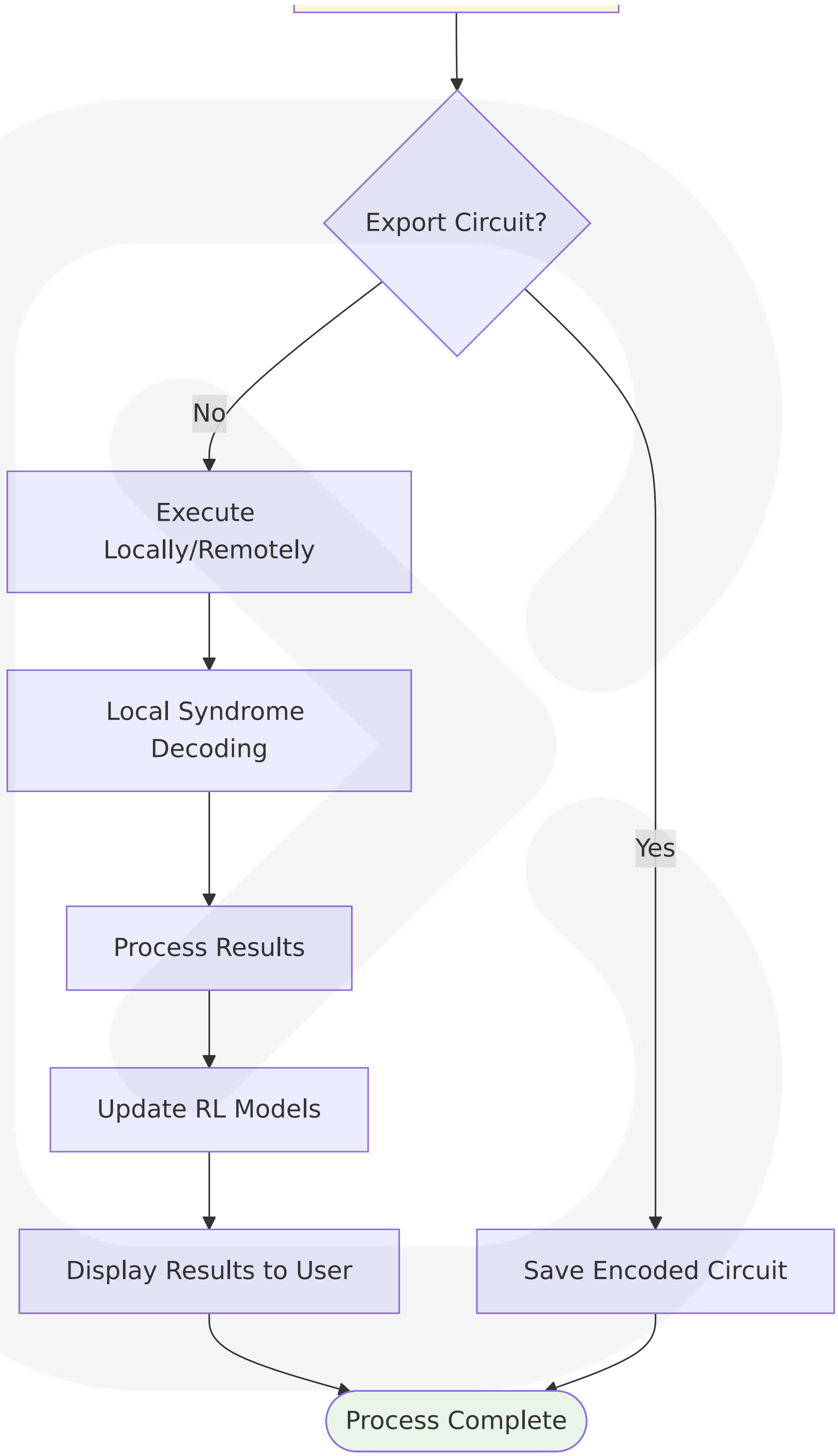
4.1 SYSTEM WORKFLOWS

4.1.1 Core Business Processes

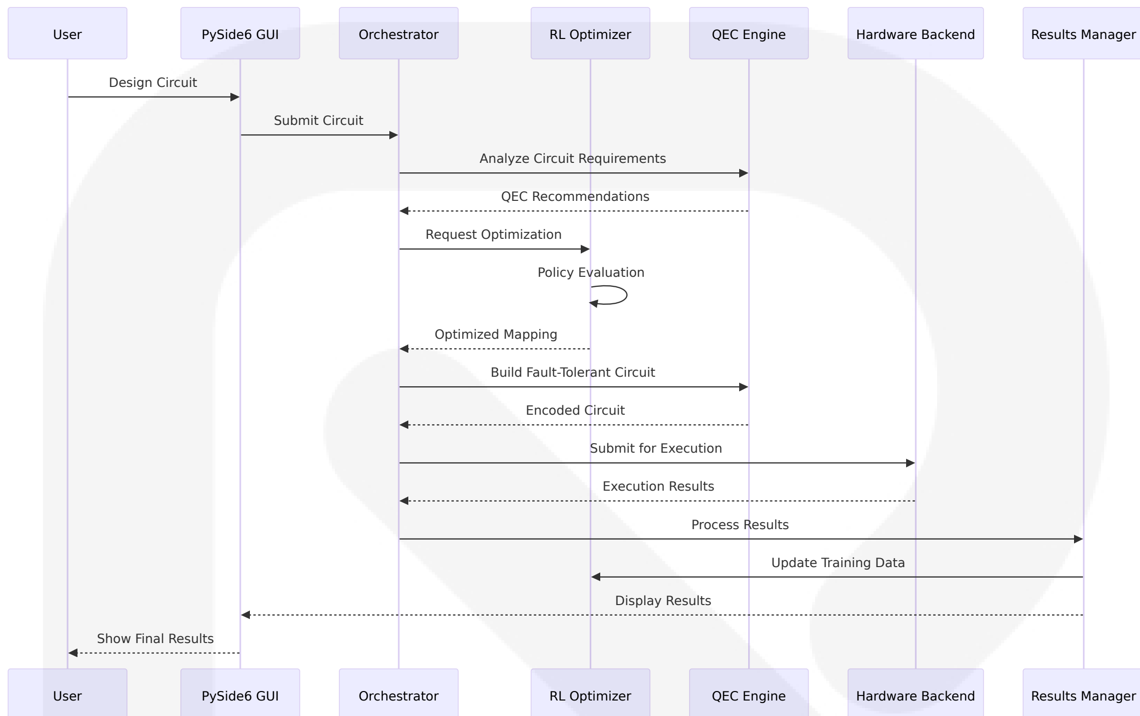
End-to-End User Journey: Quantum Circuit Compilation and Execution


User Starts QCraft
Initialize Application
Launch PySide6 GUI
Design New Circuit?
No
Yes
Load Existing Circuit
Circuit Editor
Validate Circuit Structure
Circuit Valid?


No
Yes
Display Validation Errors
Load Configuration
Select Hardware Backend
QEC Family Selection
RL-Based Code Optimization
Multi-Patch Mapping
Fault-Tolerant Circuit Building
Circuit Optimization
Encode for Privacy

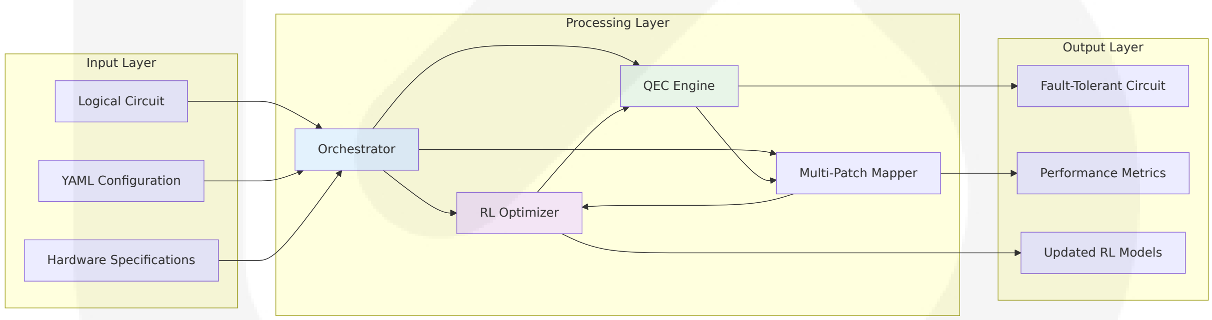

Export Circuit?
No
Yes
Execute Locally/Remotely
Local Syndrome Decoding
Process Results
Update RL Models
Display Results to User
Save Encoded Circuit
Process Complete

System Interaction Workflow

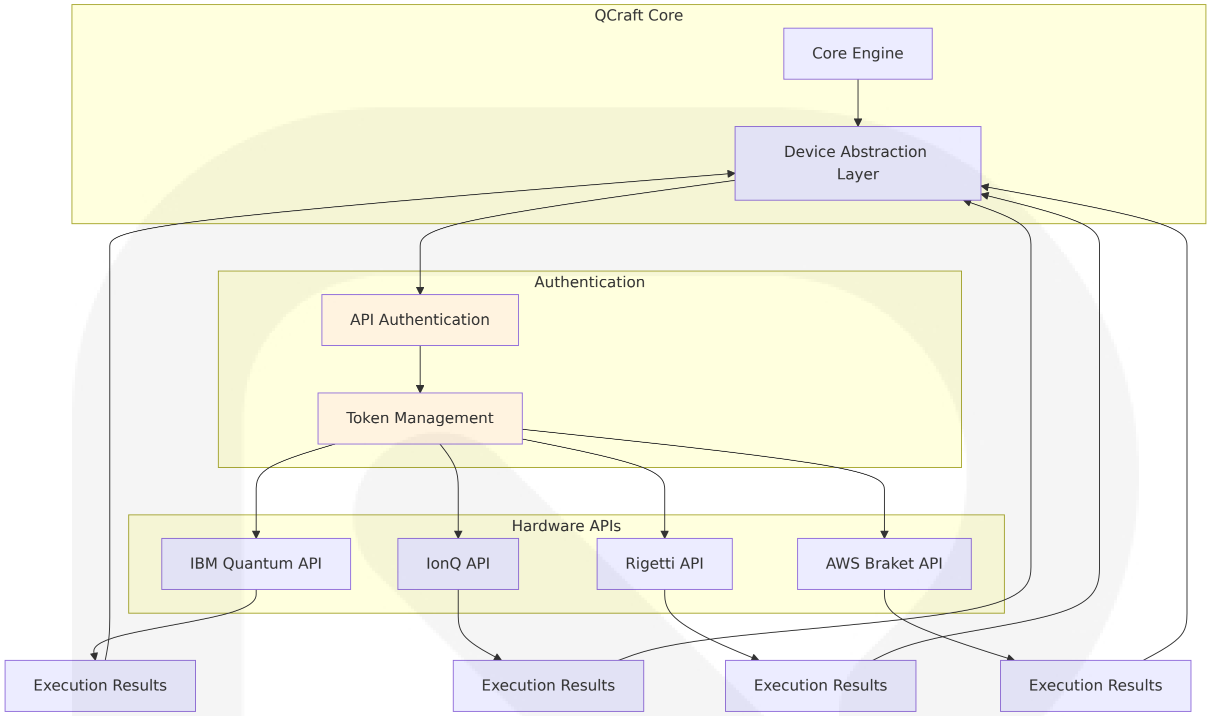


4.1.2 Integration Workflows

Data Flow Between Core Components

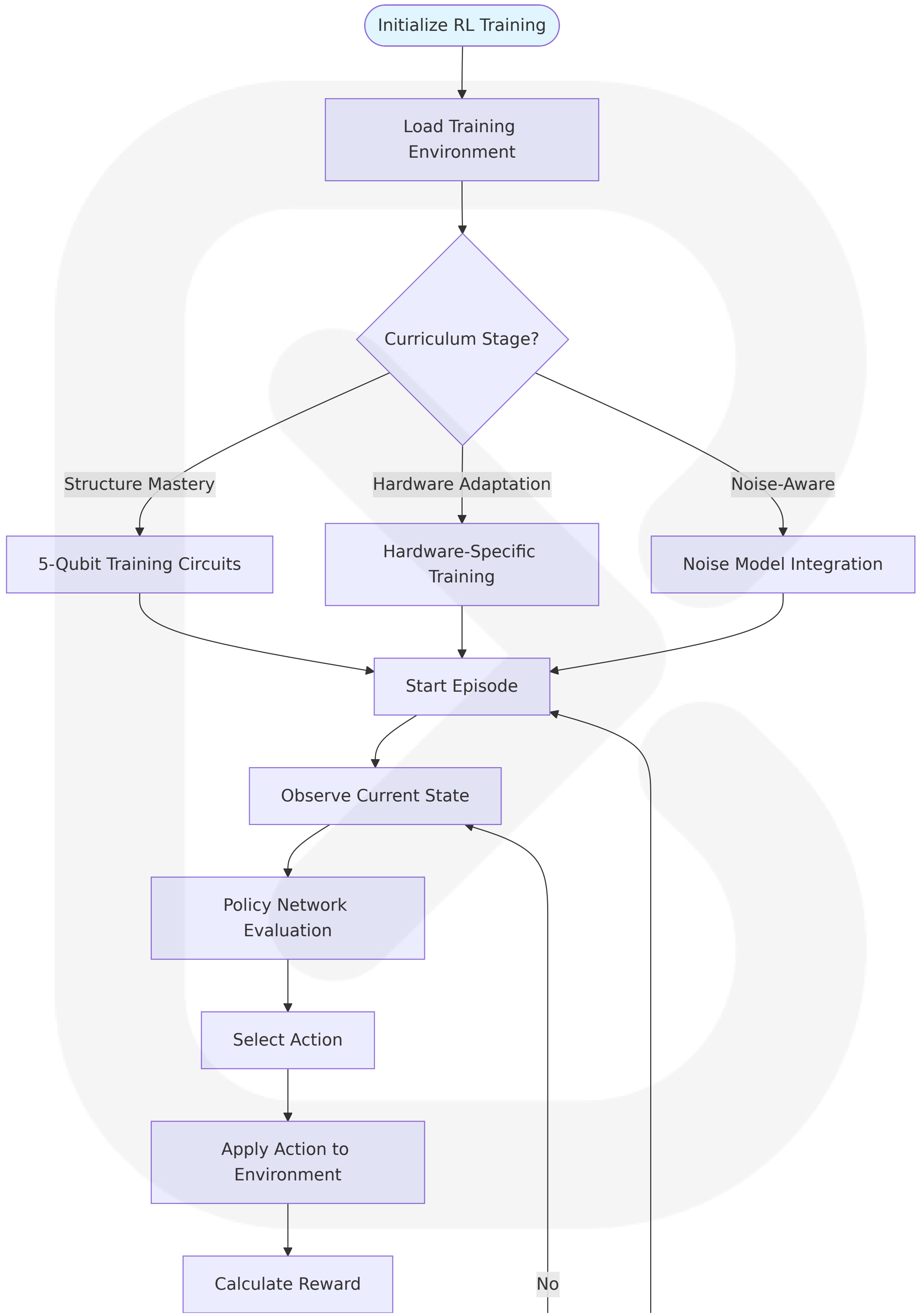


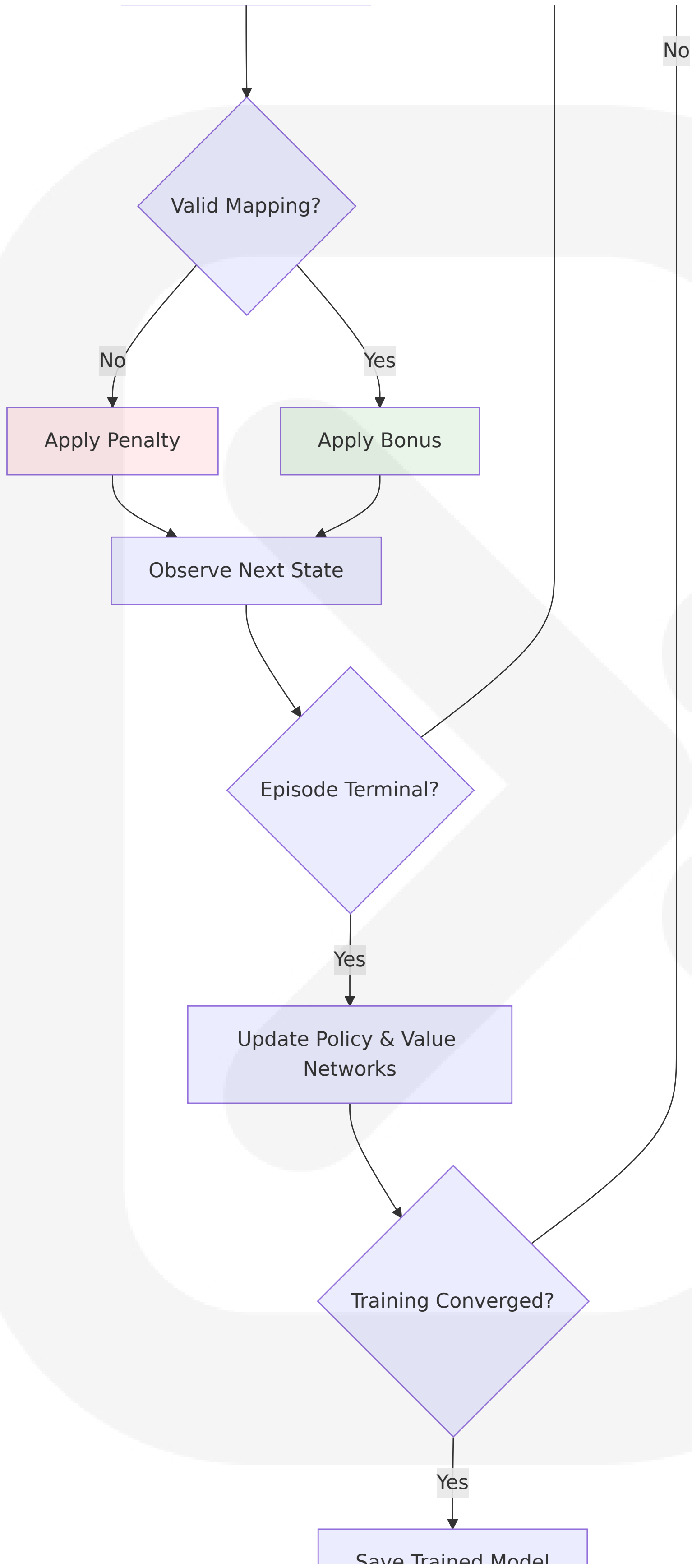
API Integration Flow

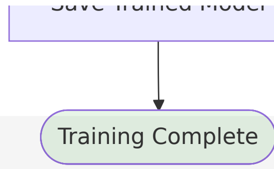


4.2 FLOWCHART REQUIREMENTS

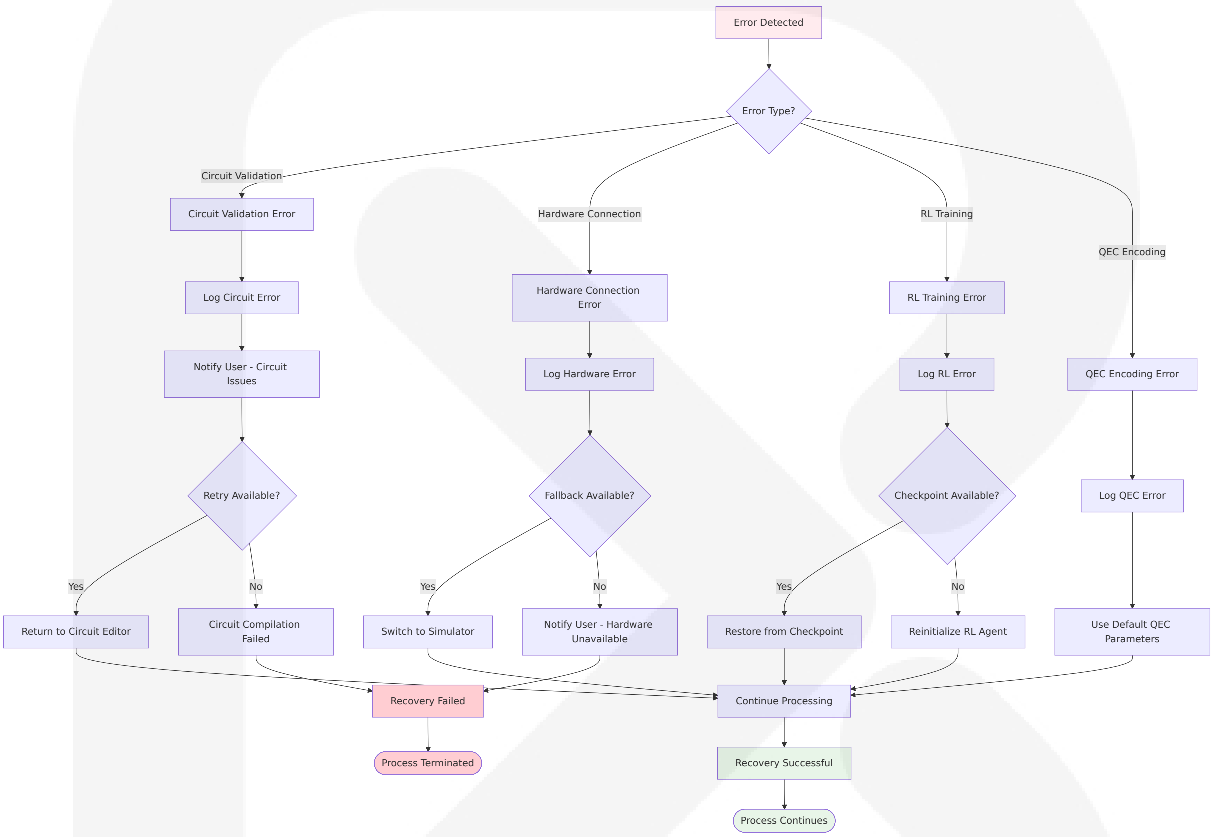
4.2.1 Reinforcement Learning Training Workflow


Initialize RL Training
Load Training Environment
Curriculum Stage?
Structure Mastery
Hardware Adaptation
Noise-Aware
5-Qubit Training Circuits
Hardware-Specific Training
Noise Model Integration
Start Episode
Observe Current State
Policy Network Evaluation
Select Action
Apply Action to Environment
Calculate Reward
No

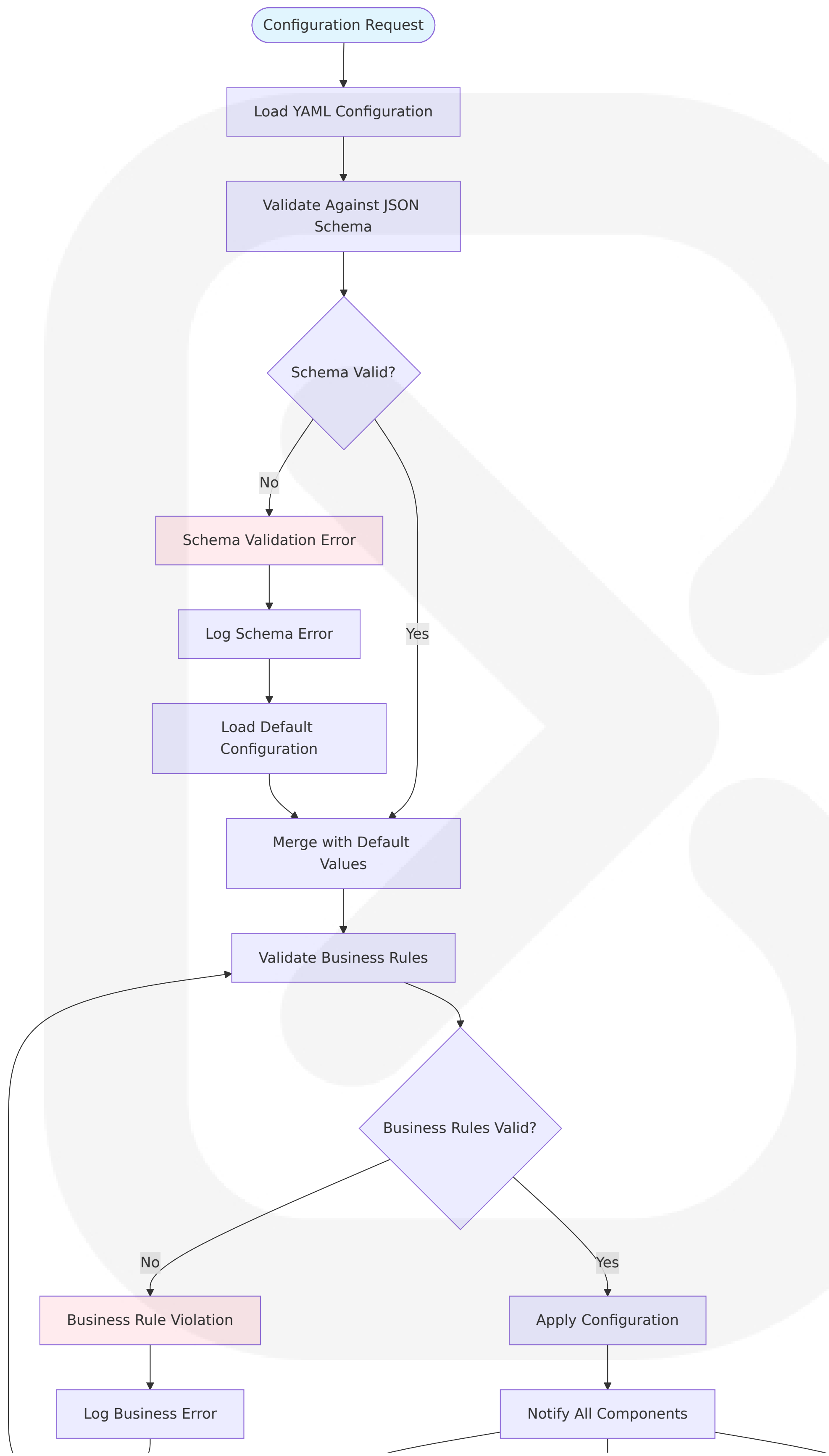

No
Valid Mapping?
No
Yes
Apply Penalty
Apply Bonus
Observe Next State
Episode Terminal?
Yes
Update Policy & Value Networks
Training Converged?
Yes
Save Trained Model

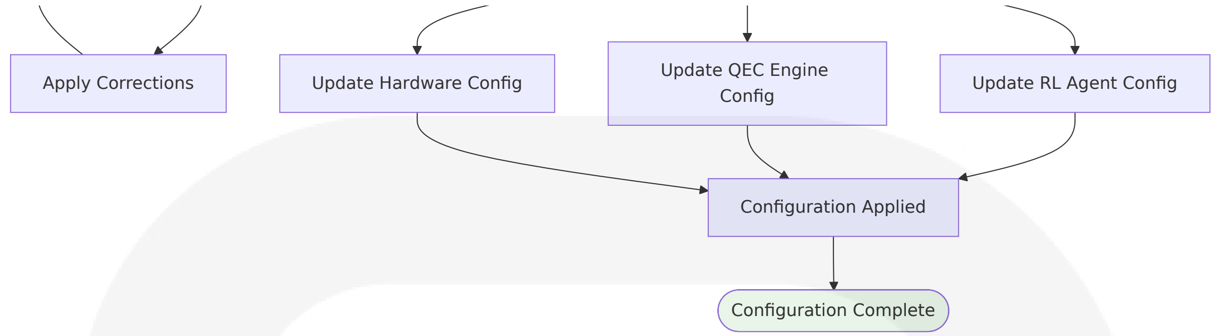


4.2.2 Error Handling and Recovery Workflow



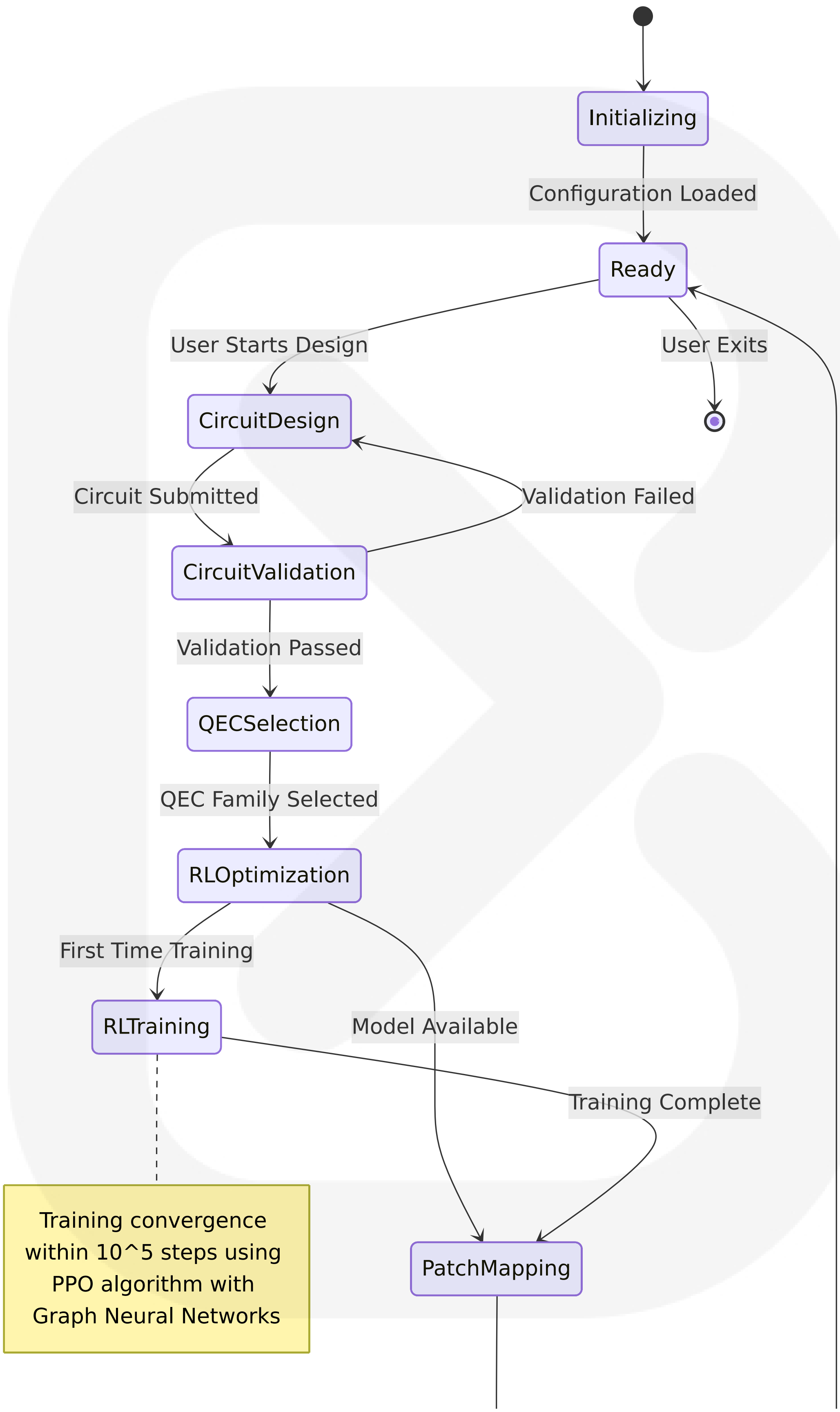
4.2.3 Configuration Management Workflow

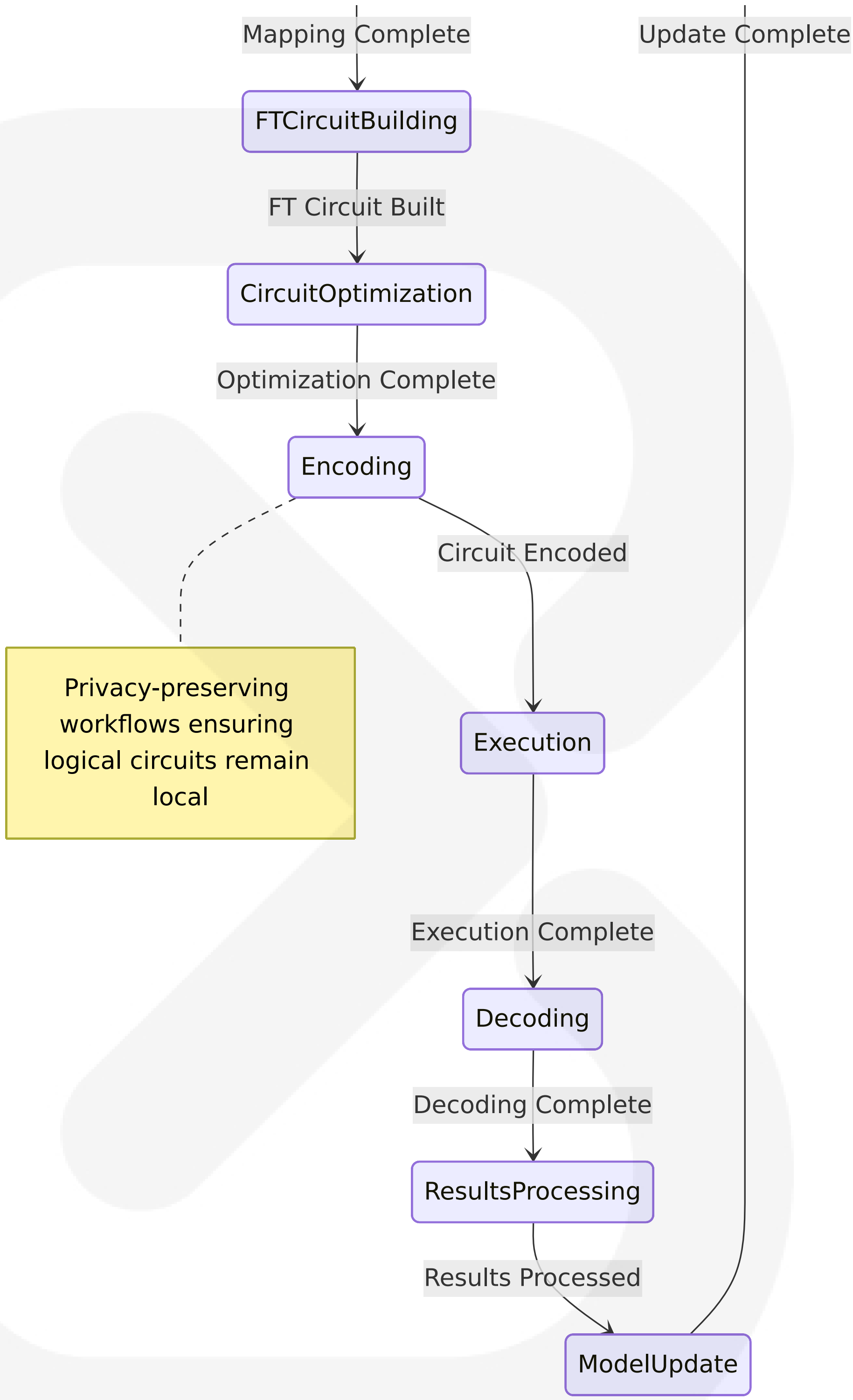

Configuration Request
Load YAML Configuration
Validate Against JSON Schema
Schema Valid?
No
Yes
Schema Validation Error
Log Schema Error
Load Default Configuration
Merge with Default Values
Validate Business Rules
Business Rules Valid?
No
Yes
Business Rule Violation
Apply Configuration
Log Business Error
Notify All Components



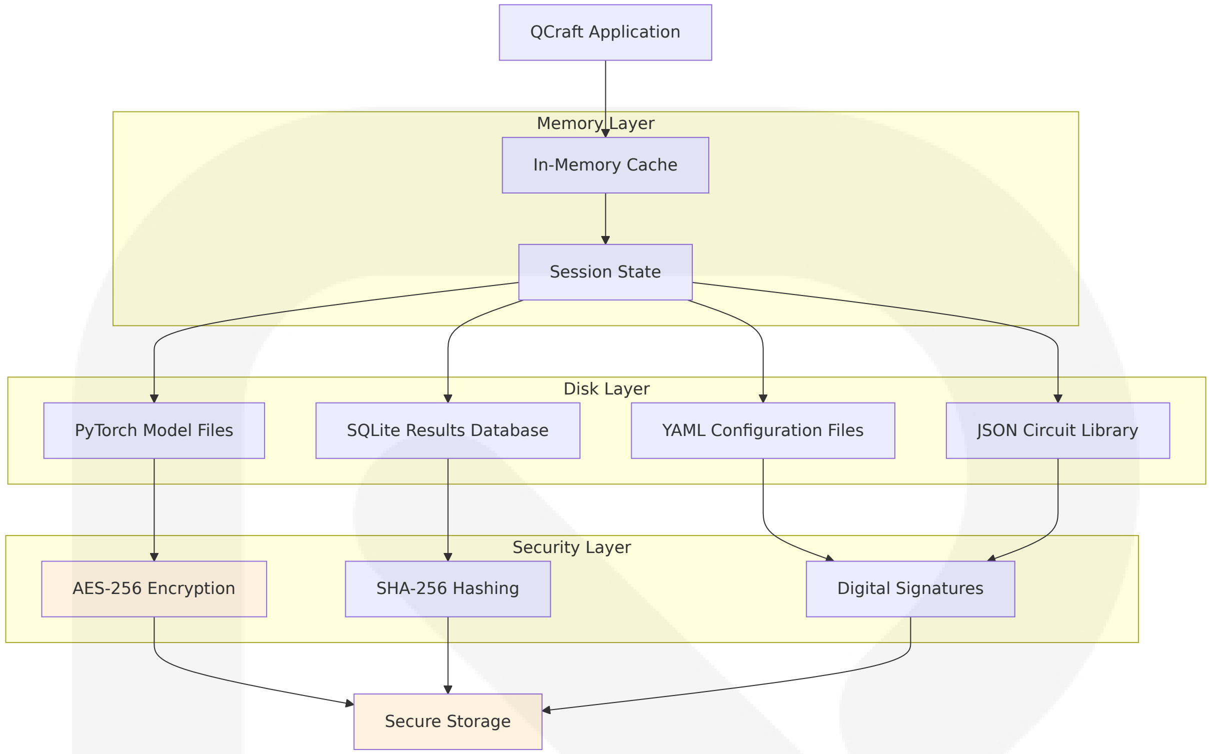
4.3 TECHNICAL IMPLEMENTATION

4.3.1 State Management Workflow

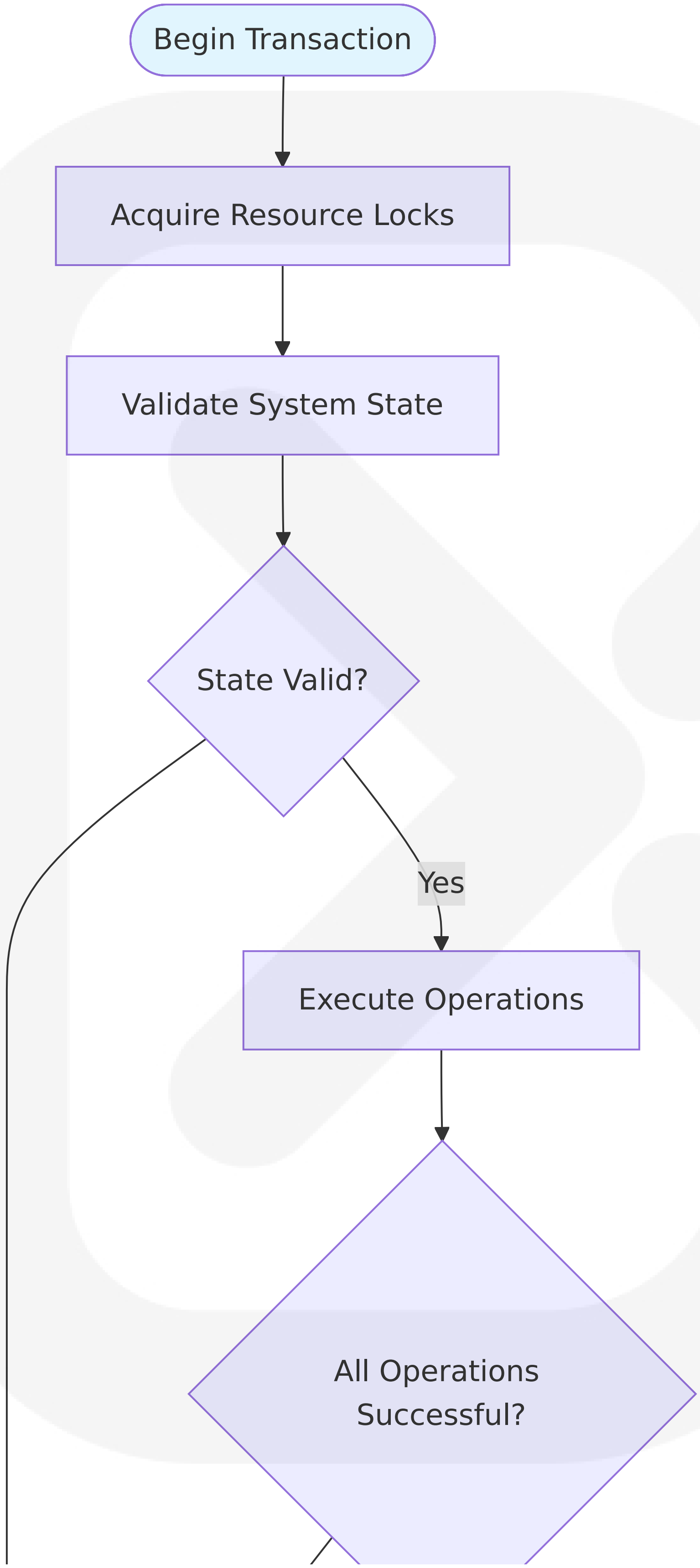

Initializing
Configuration Loaded
Ready
User Starts Design
User Exits
CircuitDesign
Circuit Submitted
Validation Failed
CircuitValidation
Validation Passed
QECSelection
QEC Family Selected
RLOptimization
First Time Training
RLTraining
Model Available
Training Complete
Training convergence within 10^5 steps using PPO algorithm with Graph Neural Networks
PatchMapping

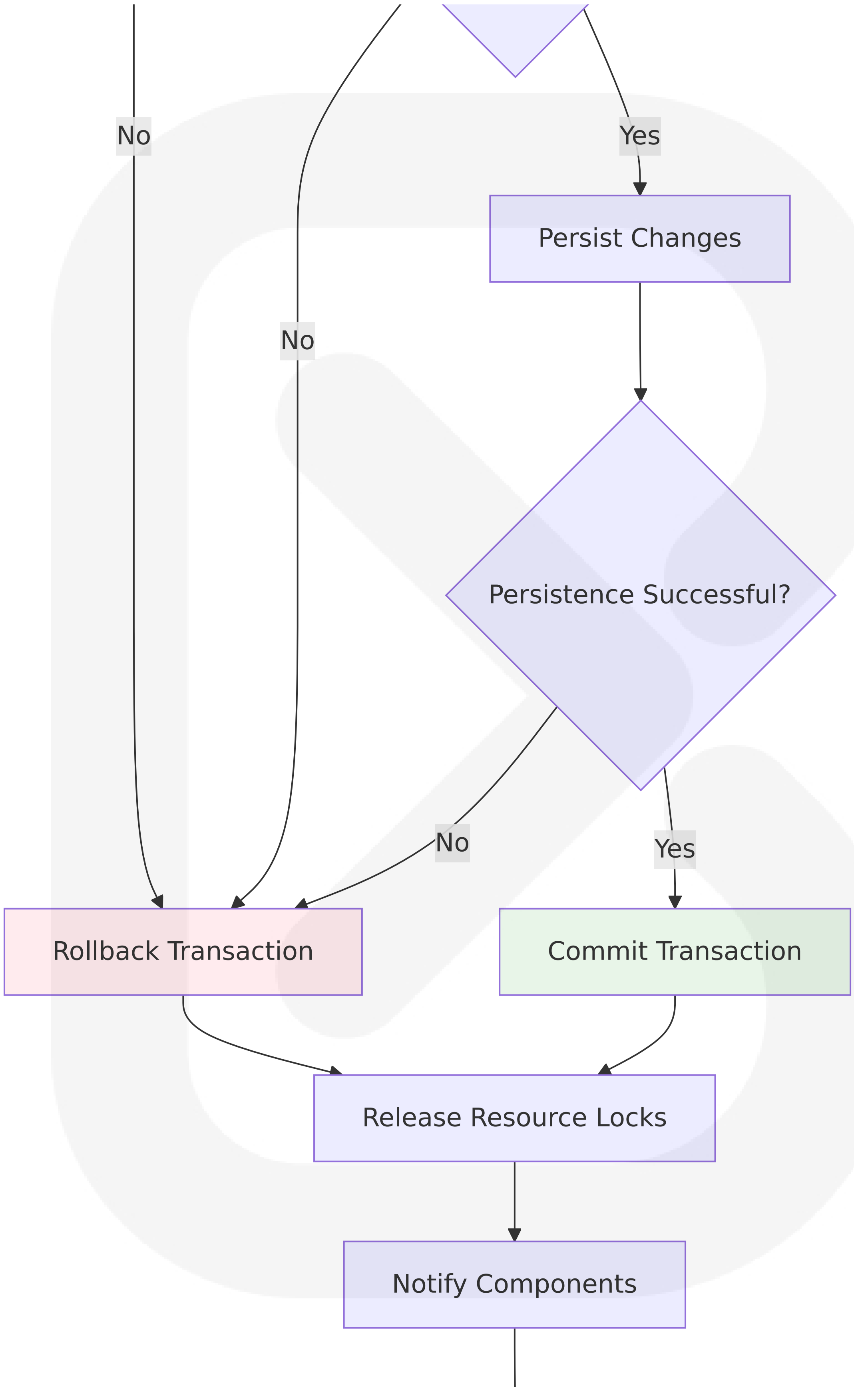


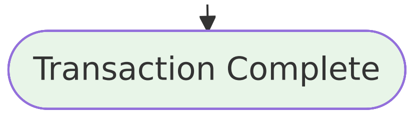
4.3.2 Data Persistence and Caching Strategy



4.3.3 Transaction Boundaries and Consistency

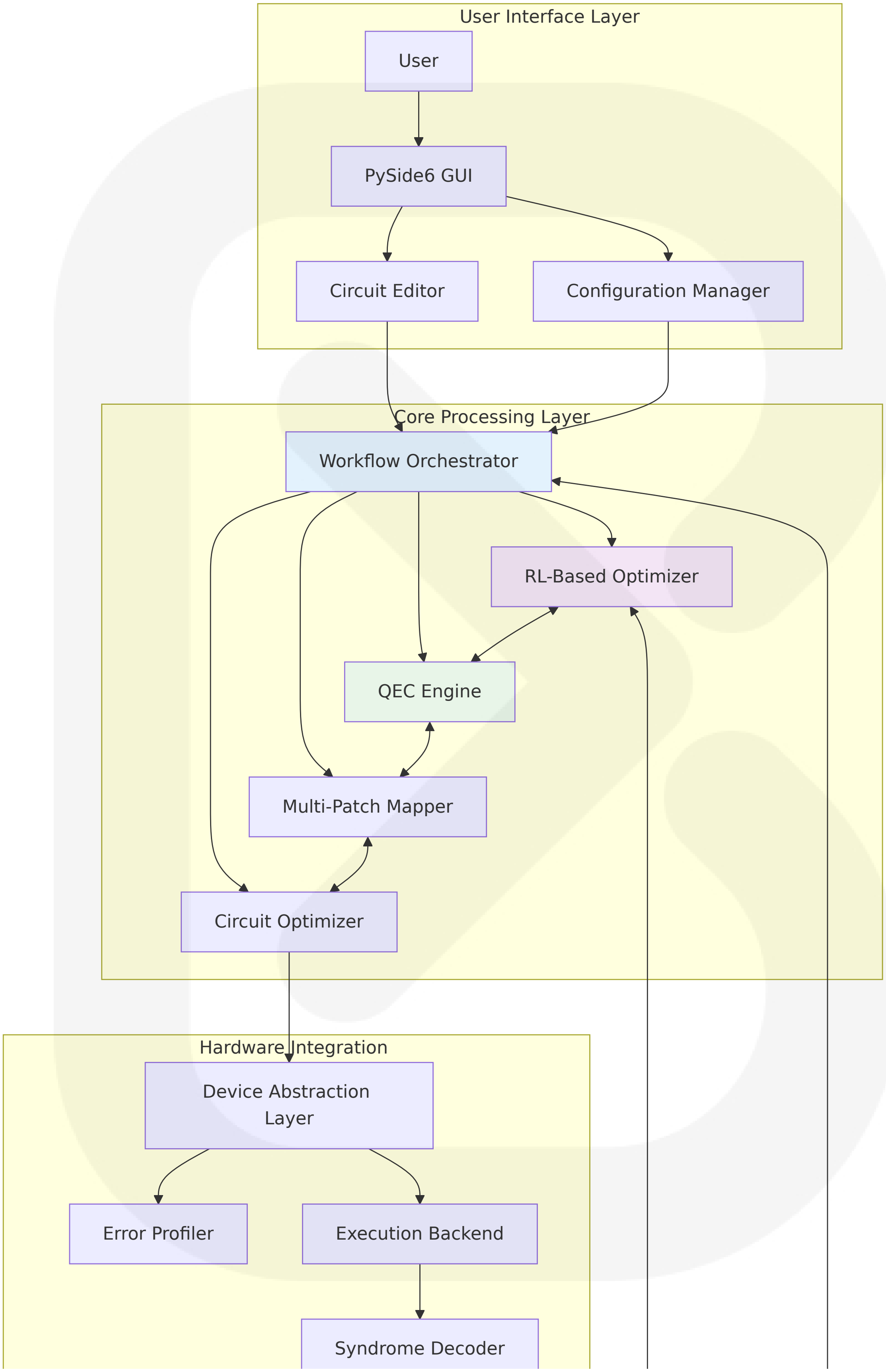

Begin Transaction
Acquire Resource Locks
Validate System State
State Valid?
Yes
Execute Operations
All Operations Successful?

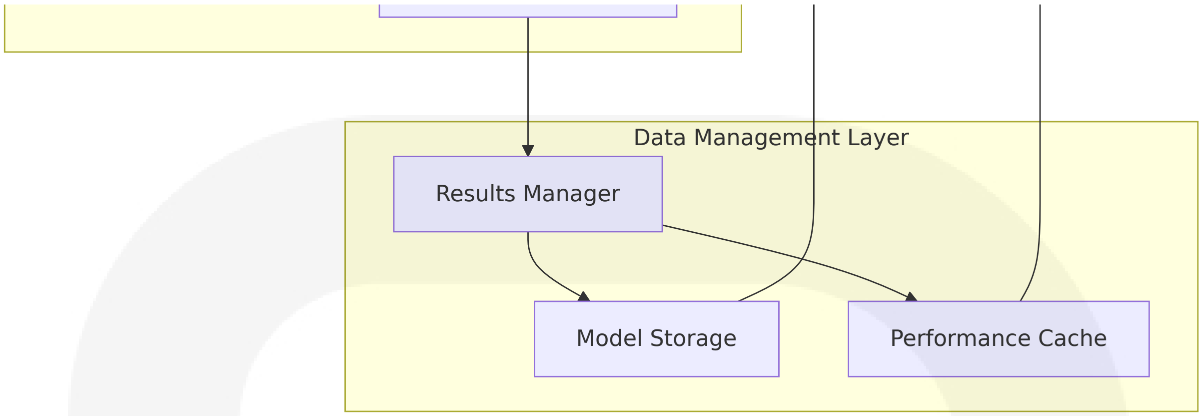

No
No
Yes
Persist Changes
Persistence Successful?
No
Yes
Rollback Transaction
Commit Transaction
Release Resource Locks
Notify Components



4.4 REQUIRED DIAGRAMS

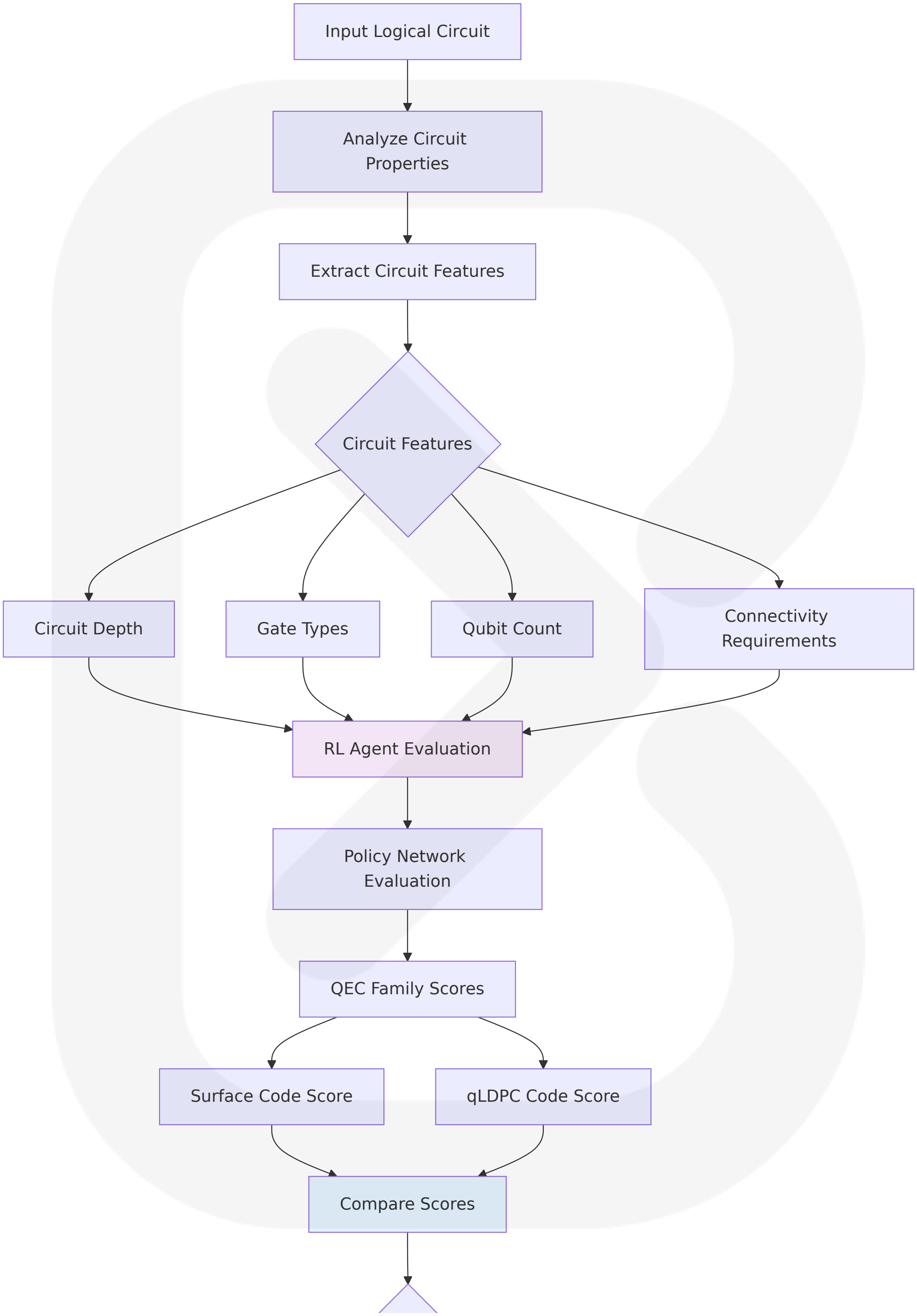
4.4.1 High-Level System Workflow

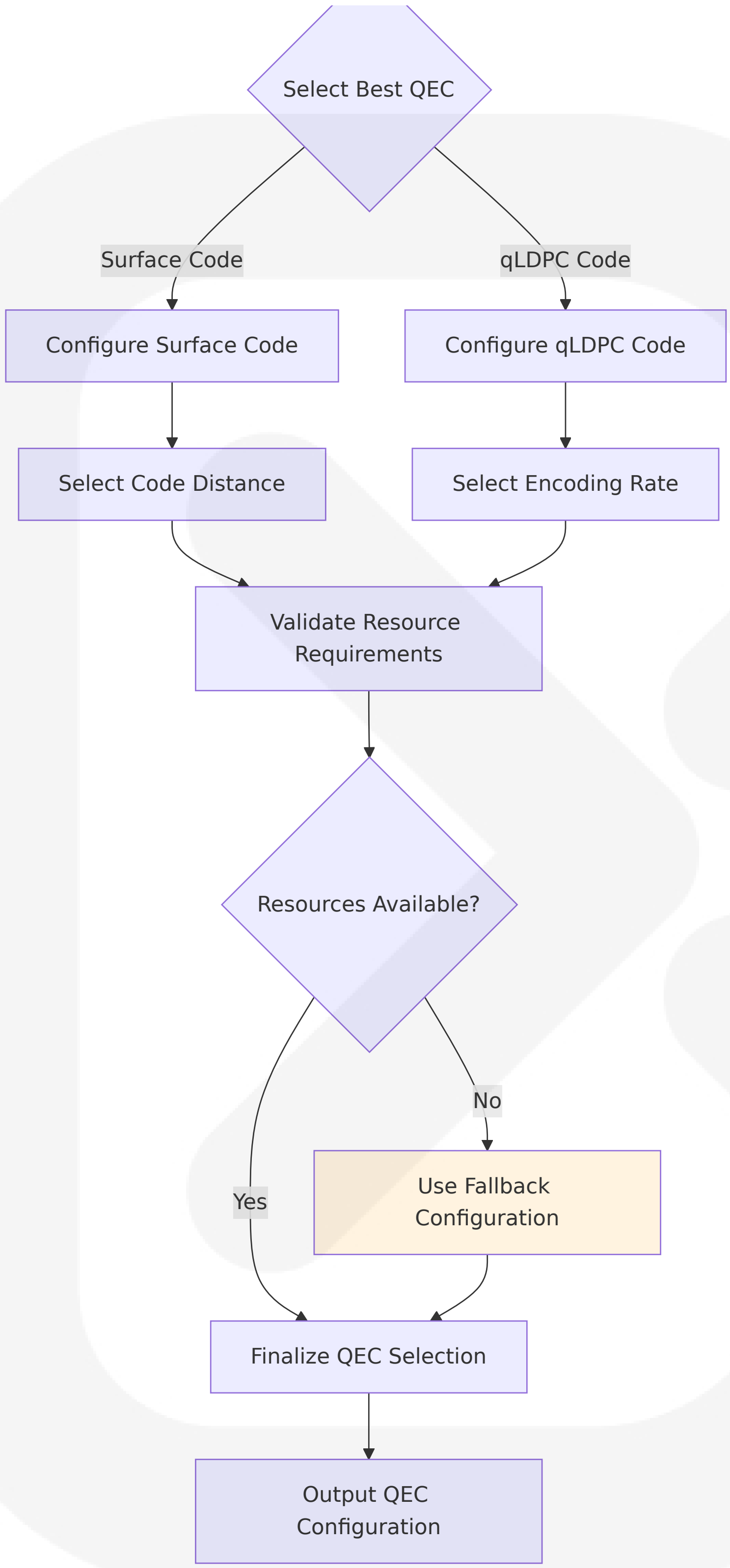

User Interface Layer
User
PySide6 GUI
Circuit Editor
Configuration Manager
Core Processing Layer
Workflow Orchestrator
RL-Based Optimizer
QEC Engine
Multi-Patch Mapper
Circuit Optimizer
Hardware Integration
Device Abstraction Layer
Error Profiler
Execution Backend
Syndrome Decoder



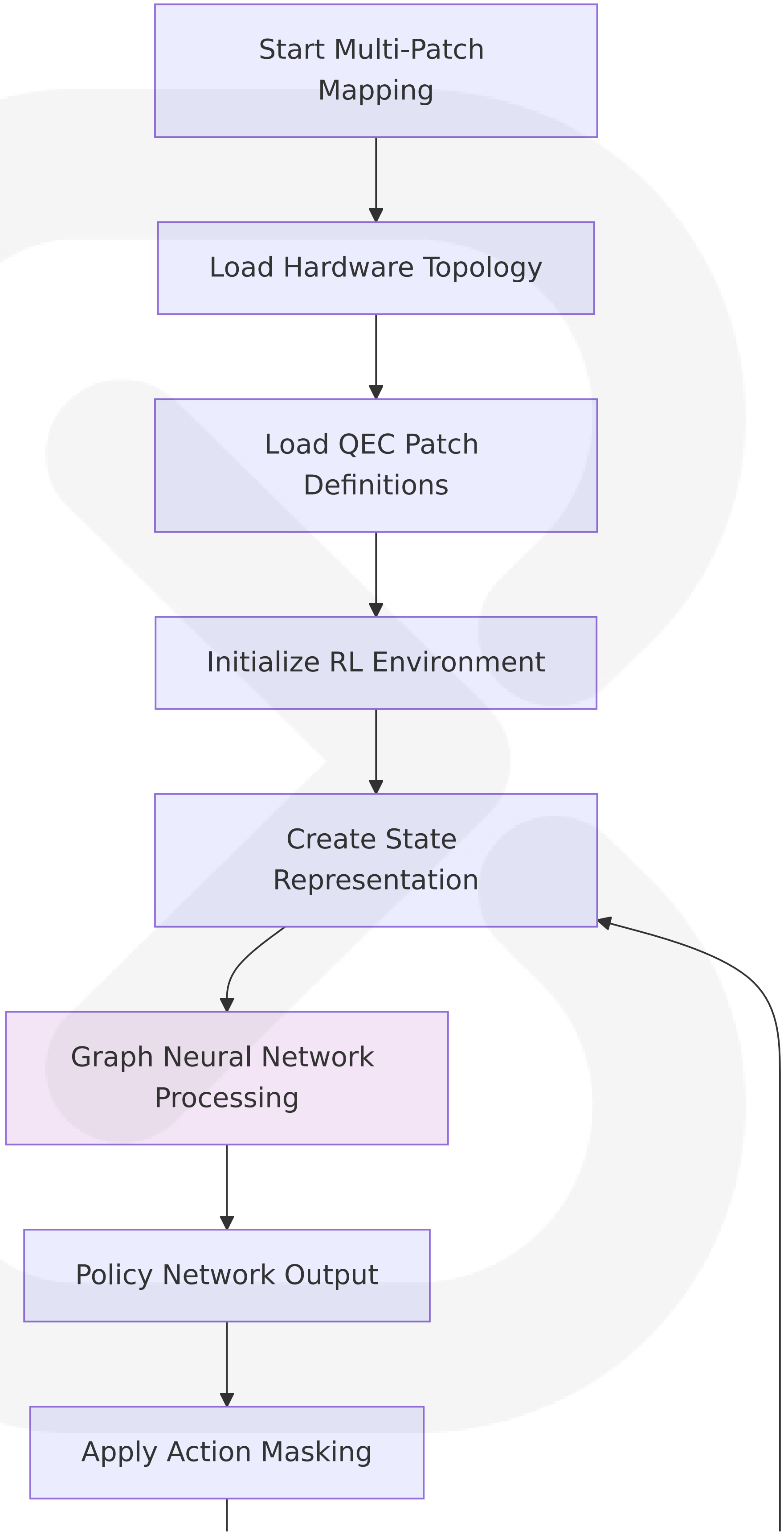
4.4.2 Detailed Process Flow for Core Features

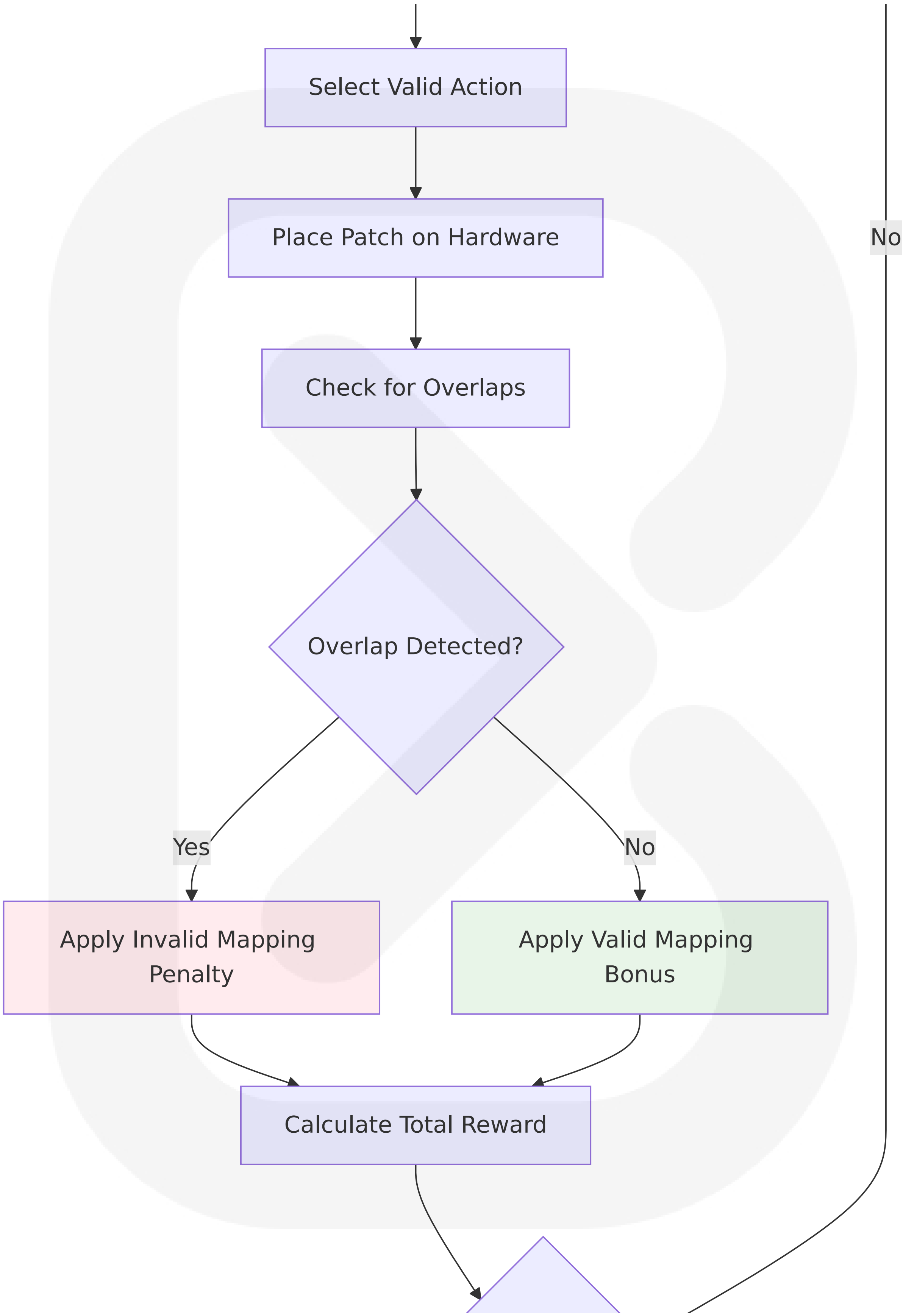
Quantum Error Correction Code Selection Process

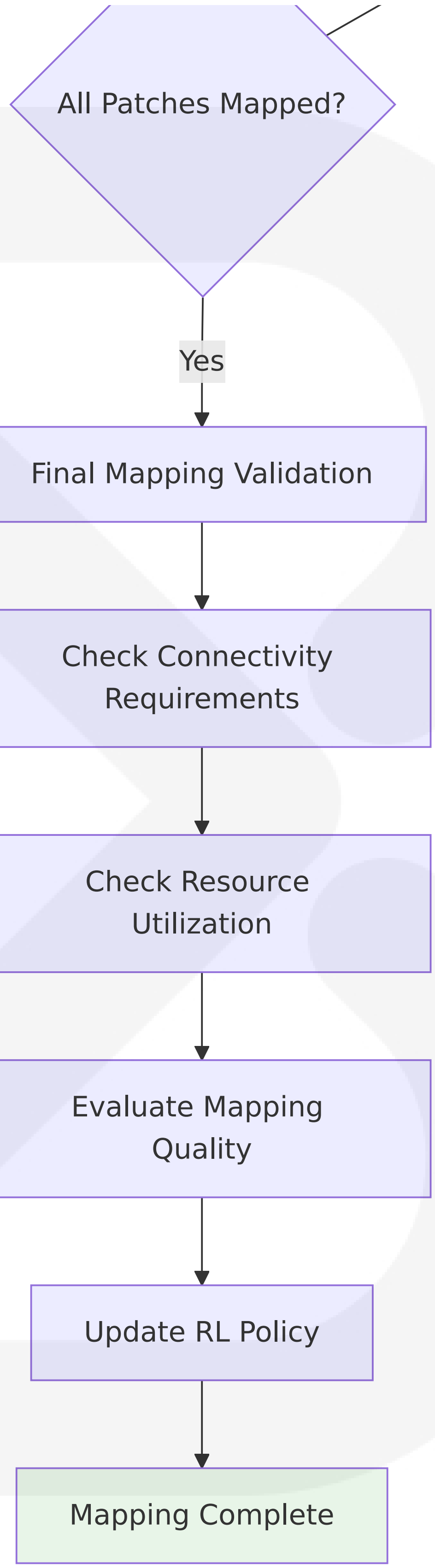

Input Logical Circuit
Analyze Circuit Properties
Extract Circuit Features
Circuit Features
Circuit Depth
Gate Types
Qubit Count
Connectivity Requirements
RL Agent Evaluation
Policy Network Evaluation
QEC Family Scores
Surface Code Score
qLDPC Code Score
Compare Scores



Multi-Patch Mapping Process

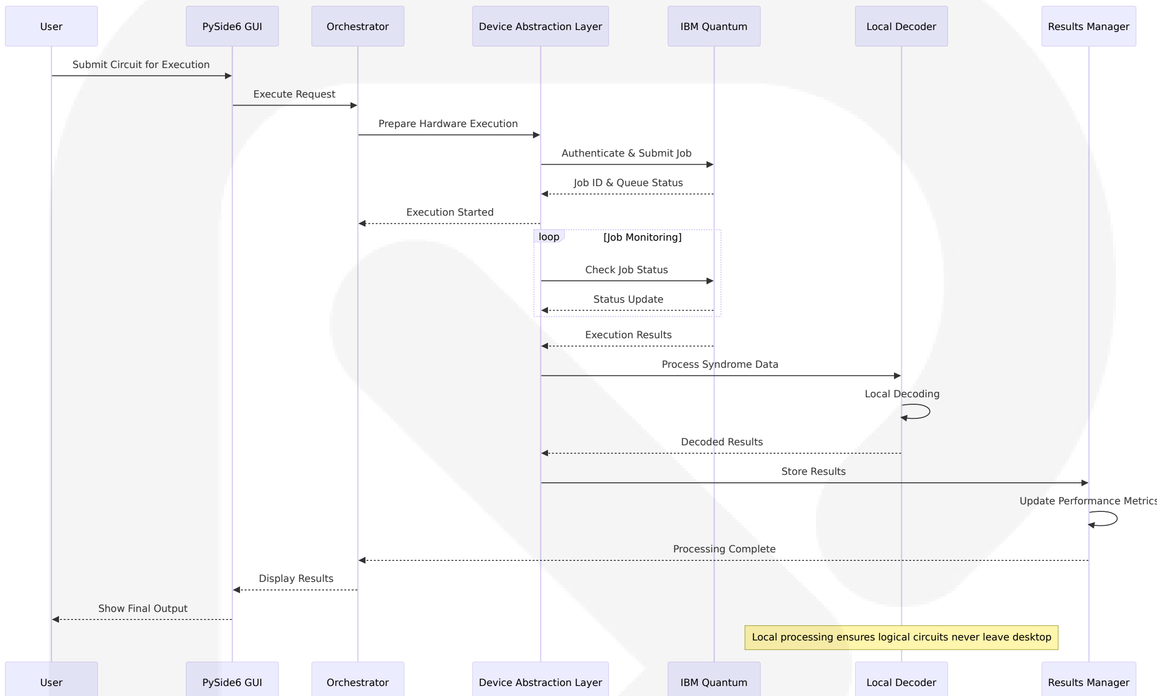

Start Multi-Patch Mapping
Load Hardware Topology
Load QEC Patch Definitions
Initialize RL Environment
Create State Representation
Graph Neural Network Processing
Policy Network Output
Apply Action Masking


Select Valid Action
Place Patch on Hardware
No
Check for Overlaps
Overlap Detected?
Yes
No
Apply Invalid Mapping Penalty
Apply Valid Mapping Bonus
Calculate Total Reward


All Patches Mapped?
Yes
Final Mapping Validation
Check Connectivity Requirements
Check Resource Utilization
Evaluate Mapping Quality
Update RL Policy
Mapping Complete

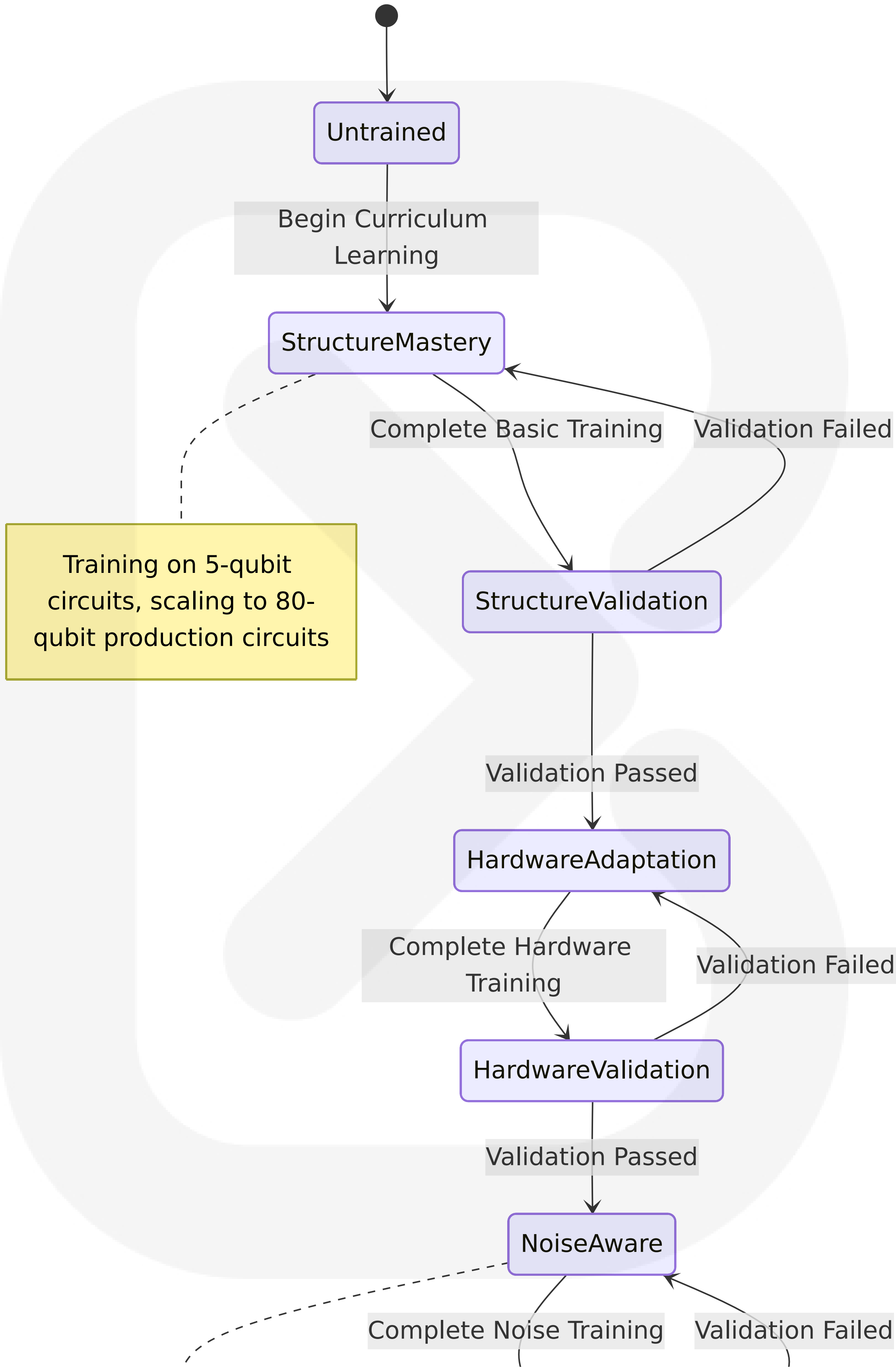
4.4.3 Integration Sequence Diagrams

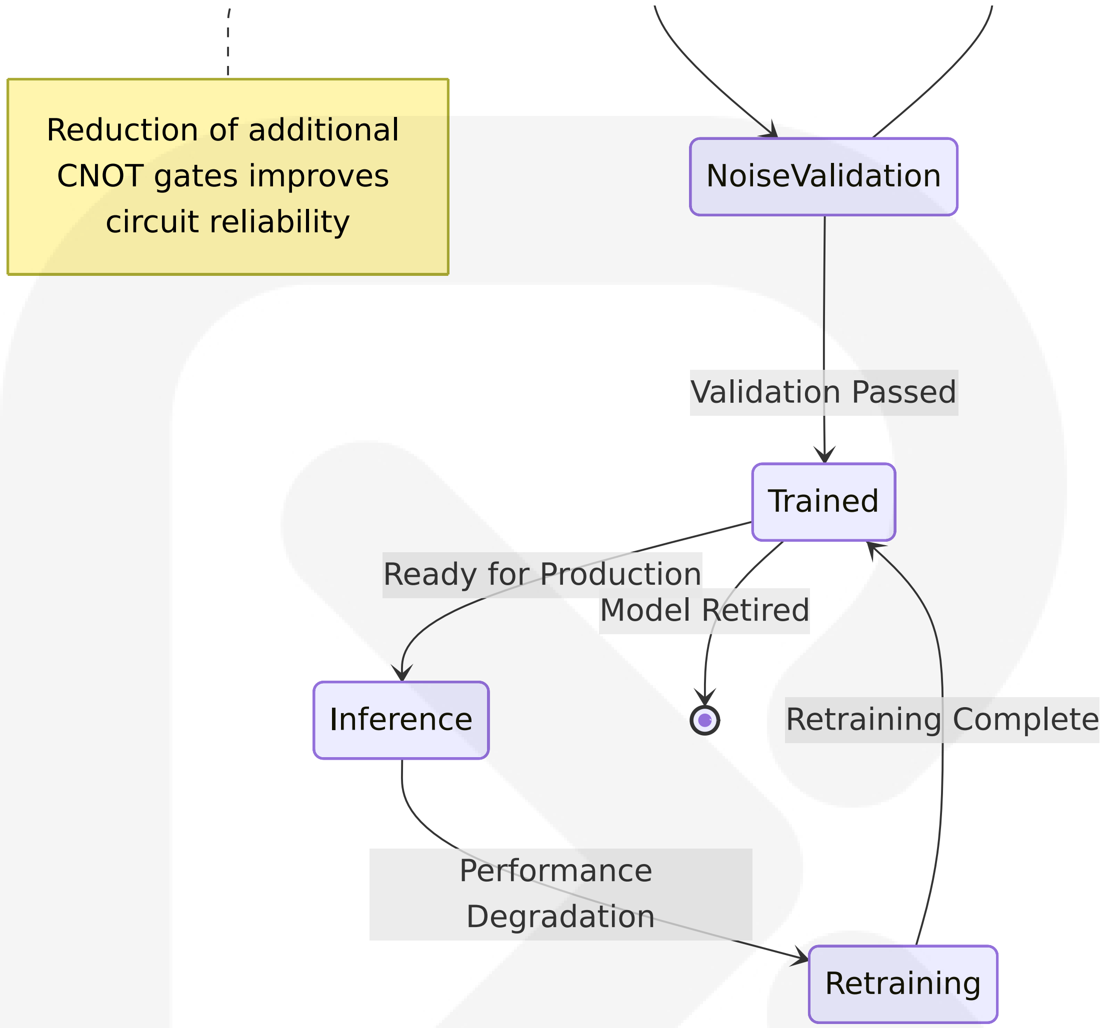
Hardware Execution Sequence



4.4.4 State Transition Diagrams

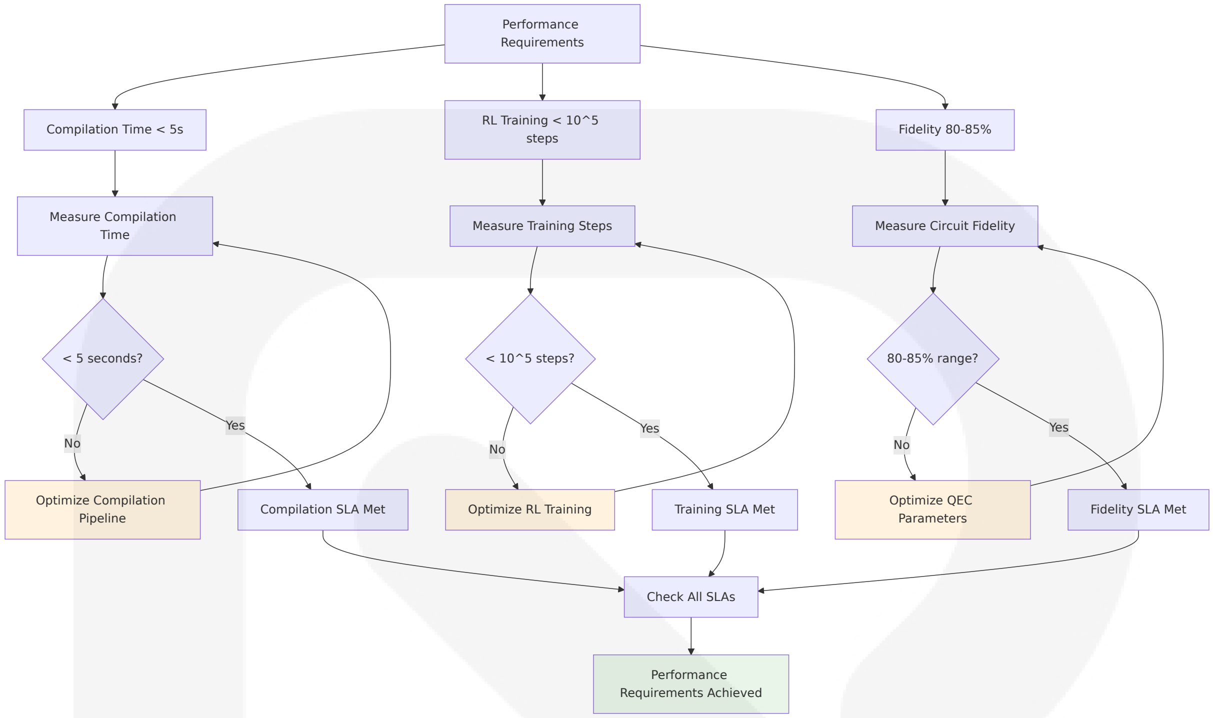
RL Agent Training State Transitions


Untrained
Begin Curriculum Learning
StructureMastery
Complete Basic Training
Validation Failed
Training on 5-qubit circuits, scaling to 80-qubit production circuits
StructureValidation
Validation Passed
HardwareAdaptation
Complete Hardware Training
Validation Failed
HardwareValidation
Validation Passed
NoiseAware
Complete Noise Training
Validation Failed



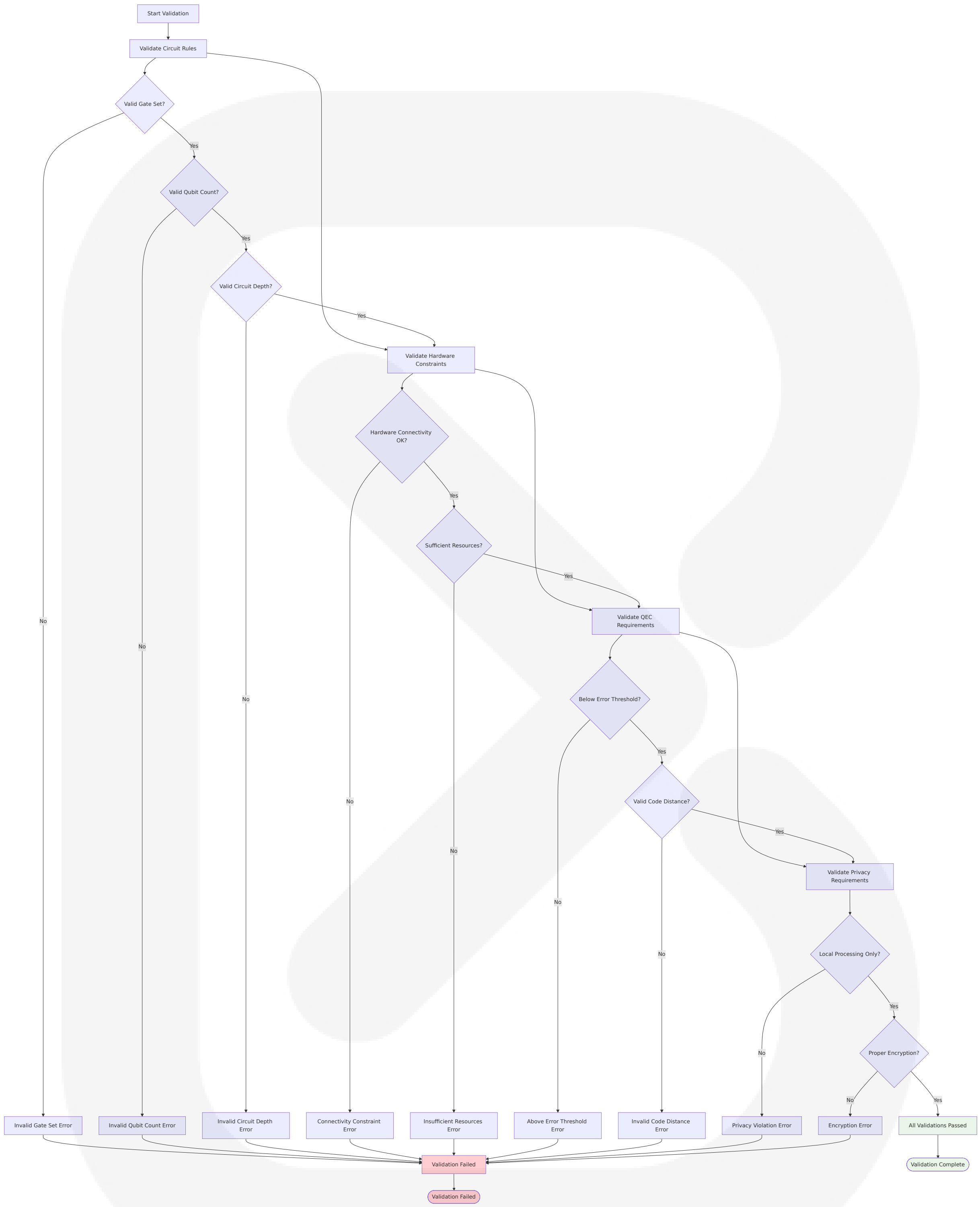
4.4.5 Timing and SLA Considerations

Performance Requirements Flowchart



4.4.6 Validation Rules and Business Logic

Business Rules Validation Flowchart



5. SYSTEM ARCHITECTURE

5.1 HIGH-LEVEL ARCHITECTURE

5.1.1 System Overview

QCraft employs a **layered desktop application architecture** with clear separation of concerns, designed specifically for privacy-preserving quantum circuit compilation and error correction. The architecture follows the **Model-View-Controller (MVC) pattern** enhanced with domain-specific layers for quantum computing operations.

The Qt ModelView architecture simplifies the linking and updating your UI with data in custom formats or from external sources, making it ideal for QCraft's complex quantum circuit representations and real-time optimization feedback. The system adopts a **plugin-based architecture** for quantum error correction families and hardware backends, ensuring extensibility while maintaining strict privacy boundaries.

The architectural style emphasizes **local-first processing** with optional cloud integration, ensuring that logical circuits never leave the user's desktop environment. This design principle drives the entire system architecture, from data flow patterns to component isolation strategies.

Key Architectural Principles:

- **Privacy by Design**: All logical circuit processing occurs locally with encrypted export only
- **Adaptive Learning**: Continuous improvement through reinforcement learning feedback loops
- **Hardware Agnostic**: Unified abstraction layer supporting multiple quantum platforms
- **Modular Extensibility**: Plugin architecture for QEC families and hardware backends
- **Configuration-Driven**: YAML/JSON-based configuration with schema validation

System Boundaries:

- **Internal Boundary**: Local desktop environment containing all logical circuit processing
- **External Boundary**: Encrypted communication with quantum hardware providers and cloud services
- **Security Boundary**: Cryptographic isolation between logical and physical circuit representations

5.1.2 Core Components Table

Component Name	Primary Responsibility	Key Dependencies	Integration Points	Critical Considerations
PySide6 GUI Frontend	User interface and circuit design	Qt 6.0+, PySide6	Workflow Orchestrator, Config Manager	PySide6 is a wrapper to Qt6, the latest version of a UI framework
Workflow Orchestrator	Central coordination and process management	All core components	GUI, Results Manager	Single point of control for all operations
RL-Based Optimizer	Quantum circuit optimization using PPO+GNN	PyTorch, Stable-Baselines3	QEC Engine, Multi-Patch Mapper	Reinforcement learning specifically suited for quantum circuit design objectives
QEC Engine	Multi-family error correction implementation	Stim, PyMatching	RL Optimizer, Circuit Builder	qLDPC codes including Bivariate Bicycle (BB) codes

5.1.3 Data Flow Description

The primary data flow follows a **pipeline architecture** with feedback loops for continuous learning. Logical circuits enter through the PySide6 GUI and undergo a series of transformations: preprocessing → QEC family selection → multi-patch mapping → fault-tolerant encoding → circuit optimization → encrypted export.

Integration Patterns:

- **Event-Driven Communication**: Qt signals/slots mechanism for real-time UI updates
- **Pipeline Processing**: Sequential transformation of quantum circuits through processing stages
- **Feedback Loops**: RL agents receive execution results to improve future optimizations
- **Configuration Injection**: YAML-driven parameters injected at each processing stage

Data Transformation Points:

- **Circuit Representation**: Conversion between logical, intermediate, and fault-tolerant representations
- **QEC Encoding**: Transformation from logical to error-corrected quantum circuits
- **Hardware Mapping**: Adaptation of circuits to specific quantum hardware topologies
- **Privacy Encoding**: Encryption and obfuscation of circuits for external execution

Key Data Stores:

- **Configuration Cache**: YAML/JSON configurations with schema validation
- **Results Database**: SQLite storage for execution results and performance metrics
- **Model Repository**: PyTorch model checkpoints for RL agents
- **Circuit Library**: JSON-based storage for quantum circuit definitions

5.1.4 External Integration Points

System Name	Integration Type	Data Exchange Pattern	Protocol/Format	SLA Requirements
IBM Quantum	REST API	Request/Response	QASM 3.0, JSON	<5s response time
IonQ Platform	REST API	Asynchronous Job Submission	JSON, Base 64	99.9% availability
Rigetti Forest	SDK Integration	Direct API Calls	Quil, JSON	Real-time execution
AWS Braket	Boto3 SDK	Batch Processing	OpenQASM, JSON	Scalable throughput

5.2 COMPONENT DETAILS

5.2.1 PySide6 GUI Frontend

Purpose and Responsibilities:
The GUI frontend provides an intuitive interface for quantum circuit design, visualization, and system configuration. PySide6 is a toolkit that lets you create software applications using Python with attractive and intuitive graphical interfaces, like a set of building blocks for software.

Technologies and Frameworks:

- **PySide6 6.9.2**: Official Python module from Qt for Python project
- **Qt 6.0+**: Native desktop application framework
- **Custom Quantum Widgets**: Specialized components for circuit visualization
- **MVC Architecture**: Separation of presentation, business logic, and data

Key Interfaces and APIs:

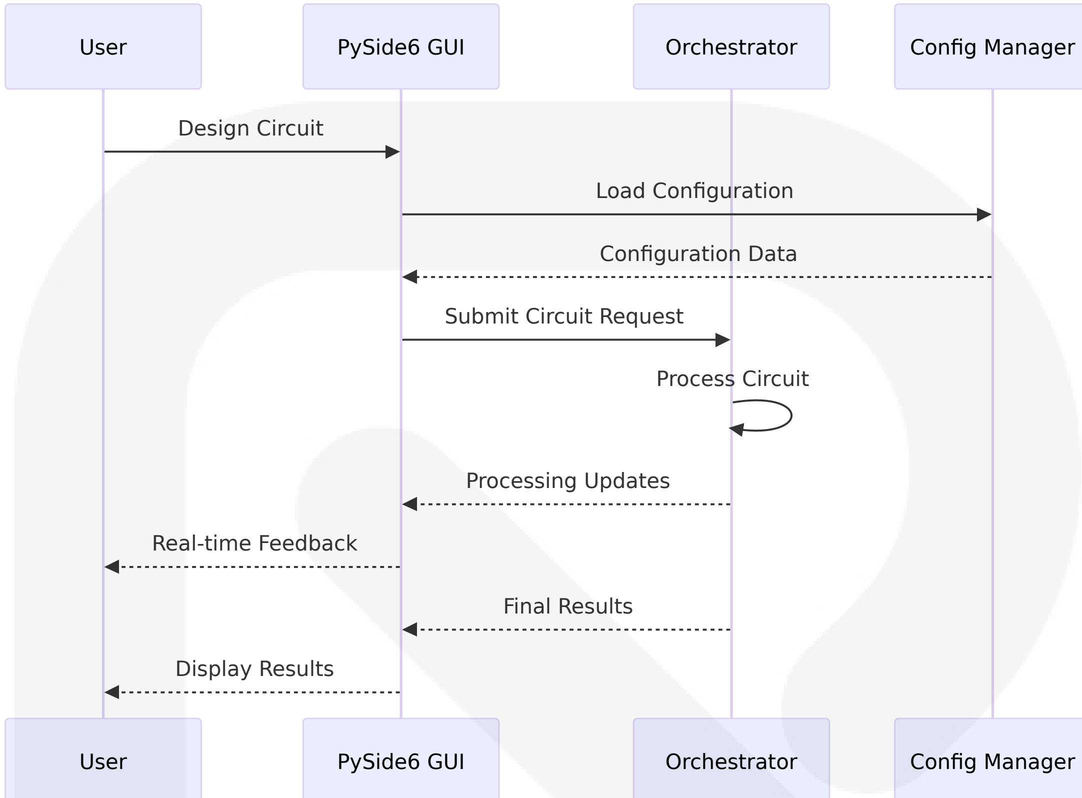
- **Circuit Editor API**: Drag-and-drop gate placement with real-time validation
- **Configuration Interface**: YAML/JSON parameter management
- **Results Visualization**: Real-time display of optimization progress and results
- **Hardware Selection**: Device abstraction layer integration

Data Persistence Requirements:

- **Session State**: In-memory storage of current circuit designs
- **User Preferences**: Local configuration file storage
- **Recent Projects**: JSON-based project history

Scaling Considerations:

- **Responsive Design**: Maintains <5ms response time for gate placement operations
- **Memory Management**: Efficient handling of large quantum circuits
- **Cross-Platform**: Native performance on Windows, macOS, and Linux



5.2.2 RL-Based Optimizer

Purpose and Responsibilities:
The RL agent develops a policy by interacting with an environment to maximize expected cumulative rewards, where observations correspond to current circuits, actions determine quantum gate placement, and rewards are based on performance.

Technologies and Frameworks:

- **Stable-Baselines3 2.7.1a3**: PPO algorithm implementation
- **PyTorch 2.0+**: Graph Neural Network implementation
- **Gymnasium 0.28.0+**: RL environment interface
- **Custom Quantum Environments**: Domain-specific reward functions and action spaces

Key Interfaces and APIs:

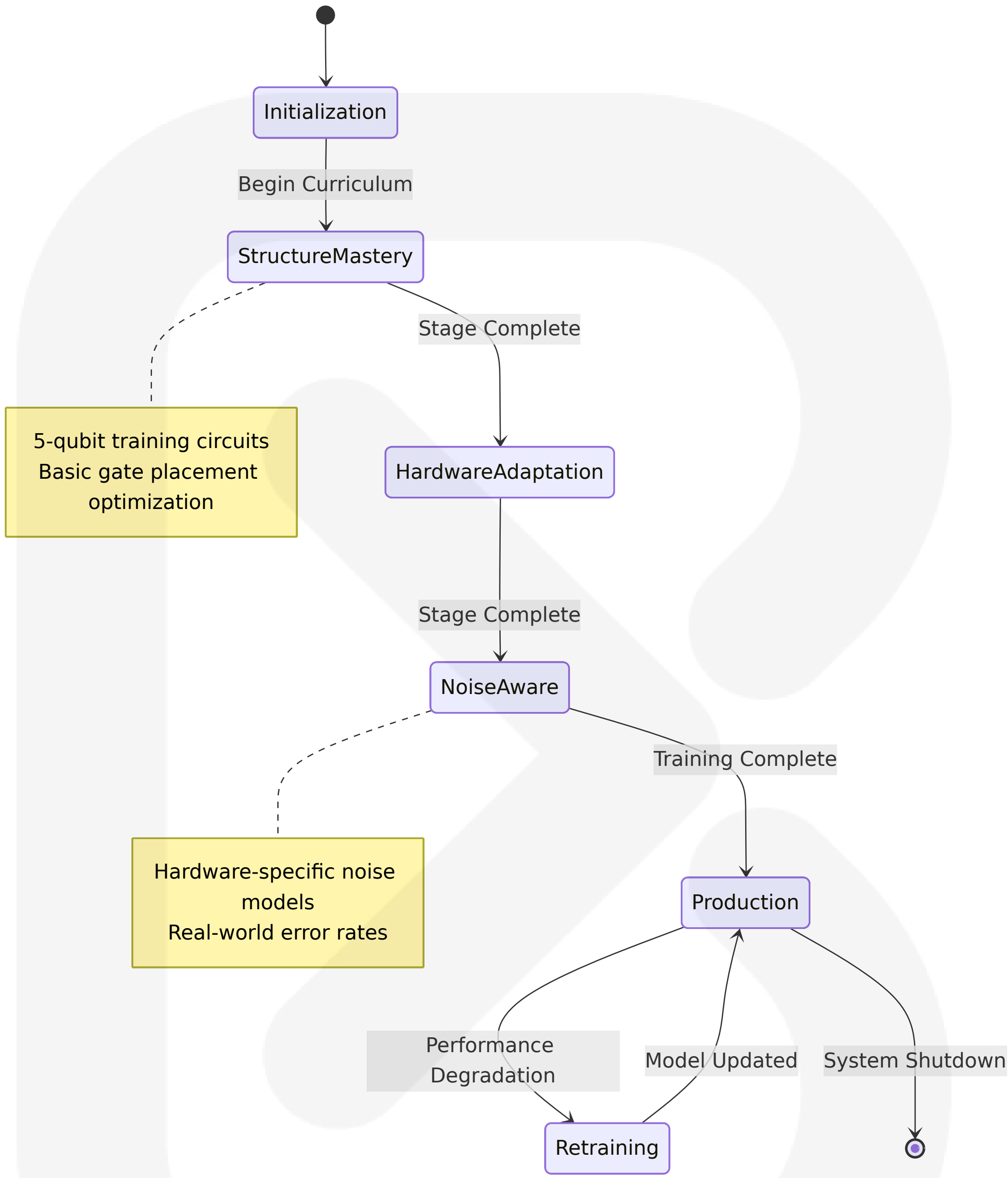
- **Policy Network API**: GNN-based policy approximation
- **Value Function API**: Circuit quality estimation
- **Training Interface**: Curriculum learning progression
- **Reward Calculation**: Multi-objective optimization metrics

Data Persistence Requirements:

- **Model Checkpoints**: PyTorch model state storage
- **Training History**: Performance metrics and convergence data
- **Experience Replay**: Circuit evaluation results for learning

Scaling Considerations:

- **Curriculum Learning**: Progressive training from 5-qubit to 80-qubit circuits
- **Distributed Training**: Multi-agent parallel optimization
- **Memory Efficiency**: Graph-based representations for scalable neural networks



5.2.3 QEC Engine

Purpose and Responsibilities:
Multi-family quantum error correction implementation supporting both surface codes and qLDPC codes including Bivariate Bicycle (BB) codes with automatic family selection based on circuit characteristics.

Technologies and Frameworks:

- **Stim 1.14.0**: High-performance stabilizer circuit simulation
- **PyMatching 2.2.1**: Minimum-weight perfect matching decoder
- **Custom QEC Libraries**: Surface code and qLDPC implementations
- **NetworkX 3.5**: Graph-based code representations

Key Interfaces and APIs:

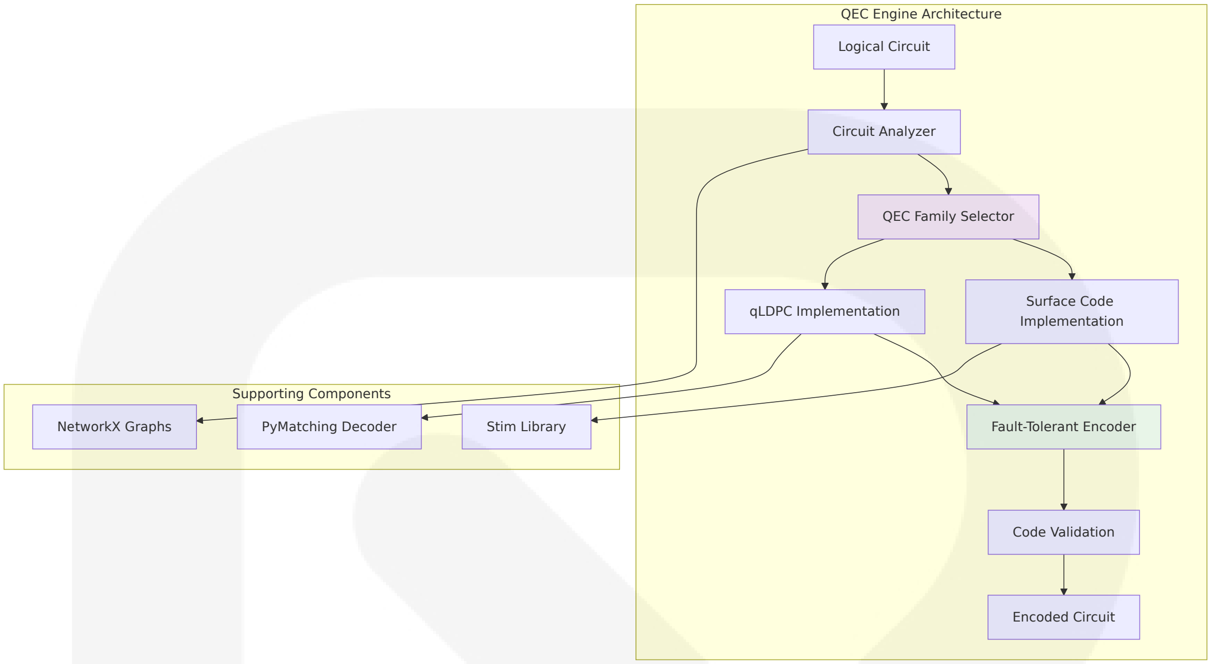
- **Surface Code API**: Distance 3, 5, 7 implementations
- **qLDPC API**: Bivariate bicycle and hypergraph product codes
- **Code Switcher API**: Automatic family selection
- **Fault-Tolerant Builder**: Logical to physical gate encoding

Data Persistence Requirements:

- **Code Definitions**: Stabilizer generators and logical operators
- **Patch Configurations**: Multi-qubit error correction layouts
- **Performance Metrics**: Error rates and resource utilization

Scaling Considerations:

- **Resource Efficiency**: qLDPC codes offer 1/24 logical-to-physical qubit encoding rate
- **Threshold Performance**: Below-threshold operation with exponential error suppression
- **Hardware Constraints**: Connectivity-aware patch placement



5.2.4 Multi-Patch Mapper

Purpose and Responsibilities:
Intelligent placement of multiple QEC patches onto quantum hardware topologies with connectivity optimization and resource constraint satisfaction.

Technologies and Frameworks:

- **NetworkX 3.5**: Hardware topology representation
- **Custom Graph Algorithms**: Patch placement optimization
- **Constraint Satisfaction**: Hardware connectivity validation
- **RL Integration**: Policy-based mapping decisions

Key Interfaces and APIs:

- **Hardware Topology API**: Device connectivity representation
- **Patch Placement API**: Multi-qubit error correction layout
- **Constraint Validation**: Hardware compatibility checking
- **Optimization Interface**: RL-based placement strategies

Data Persistence Requirements:

- **Hardware Profiles**: Device topology and connectivity data
- **Mapping Solutions**: Optimized patch placements
- **Performance History**: Mapping quality metrics

Scaling Considerations:

- **Polynomial Complexity**: Efficient algorithms for large qubit counts
- **Hardware Adaptability**: Support for diverse quantum architectures
- **Real-time Optimization**: Sub-10-second mapping for 20 logical qubits

5.3 TECHNICAL DECISIONS

5.3.1 Architecture Style Decisions

Decision	Rationale	Tradeoffs	Alternatives Considered
Layered Desktop Architecture	Privacy-first design with local processing	Limited cloud scalability	Web-based architecture
Plugin-Based QEC Support	Extensibility for new error correction families	Increased complexity	Monolithic implementation
Event-Driven Communication	Real-time UI updates and loose coupling	Debugging complexity	Direct method calls
Configuration-Driven Design	Reproducible experiments and parameter tuning	Configuration management overhead	Hardcoded parameters

Architecture Style Rationale:

The layered desktop architecture was chosen to ensure logical circuits never leave the user's desktop environment, addressing critical privacy requirements for quantum algorithm development. This approach provides complete control over sensitive quantum circuit data while enabling optional cloud integration for hardware execution.

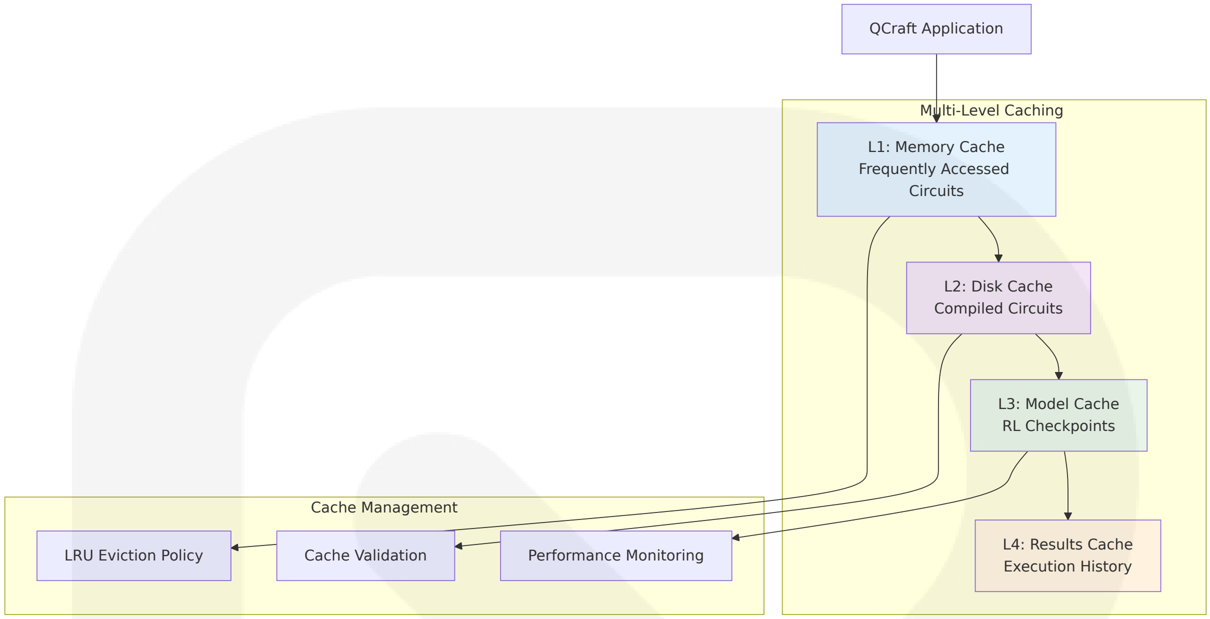
Communication Pattern Justification:
The Qt ModelView architecture simplifies linking and updating UI with data in custom formats, making event-driven communication through Qt's signals/slots mechanism the natural choice for real-time quantum circuit visualization and optimization feedback.

5.3.2 Data Storage Solution Rationale

Storage Type	Technology Choice	Justification	Performance Characteristics
Configuration	YAML/JSON Files	Human-readable, version-controllable	<1ms read/write operations
Results Cache	SQLite	Embedded, ACID compliance	1000+ queries/second
Model Storage	PyTorch Native	Framework compatibility	Optimized serialization
Circuit Library	JSON with Schema	Structured validation	Schema-enforced integrity

Local Storage Strategy:
All critical data remains local to ensure privacy compliance and reduce external dependencies. SQLite provides ACID compliance for results caching while maintaining the embedded nature required for desktop applications.

5.3.3 Caching Strategy Justification



Caching Rationale:
Multi-level caching reduces compilation times and improves user experience by storing frequently accessed quantum circuits, compiled results, and RL model states. The LRU eviction policy ensures optimal memory utilization while maintaining performance.

5.3.4 Security Mechanism Selection

Security Layer	Implementation	Purpose	Performance Impact
Local Processing	Process Isolation	Logical circuit privacy	Minimal overhead
Encrypted Export	AES-256	Secure circuit transmission	<5% performance impact
Configuration Integrity	Digital Signatures	Tamper detection	Negligible impact
Model Protection	Encrypted Storage	RL model security	<2% storage overhead

Security Architecture Decisions:
The security model prioritizes privacy through local processing while enabling secure cloud integration. AES-256 encryption ensures that only

fault-tolerant, encoded circuits are transmitted externally, maintaining the privacy of logical quantum algorithms.

5.4 CROSS-CUTTING CONCERNS

5.4.1 Monitoring and Observability Approach

Comprehensive Monitoring Strategy:

- **Performance Metrics**: Real-time tracking of compilation times, RL convergence rates, and circuit fidelity
- **System Health**: Memory usage, CPU utilization, and component responsiveness monitoring
- **User Analytics**: Privacy-preserving usage patterns and feature utilization tracking
- **Error Tracking**: Comprehensive logging of system errors and recovery actions

Observability Implementation:

- **Structured Logging**: JSON-formatted logs with correlation IDs for distributed tracing
- **Metrics Collection**: Prometheus-compatible metrics for performance monitoring
- **Health Checks**: Automated system health validation and alerting
- **Performance Profiling**: Built-in profiling tools for optimization bottleneck identification

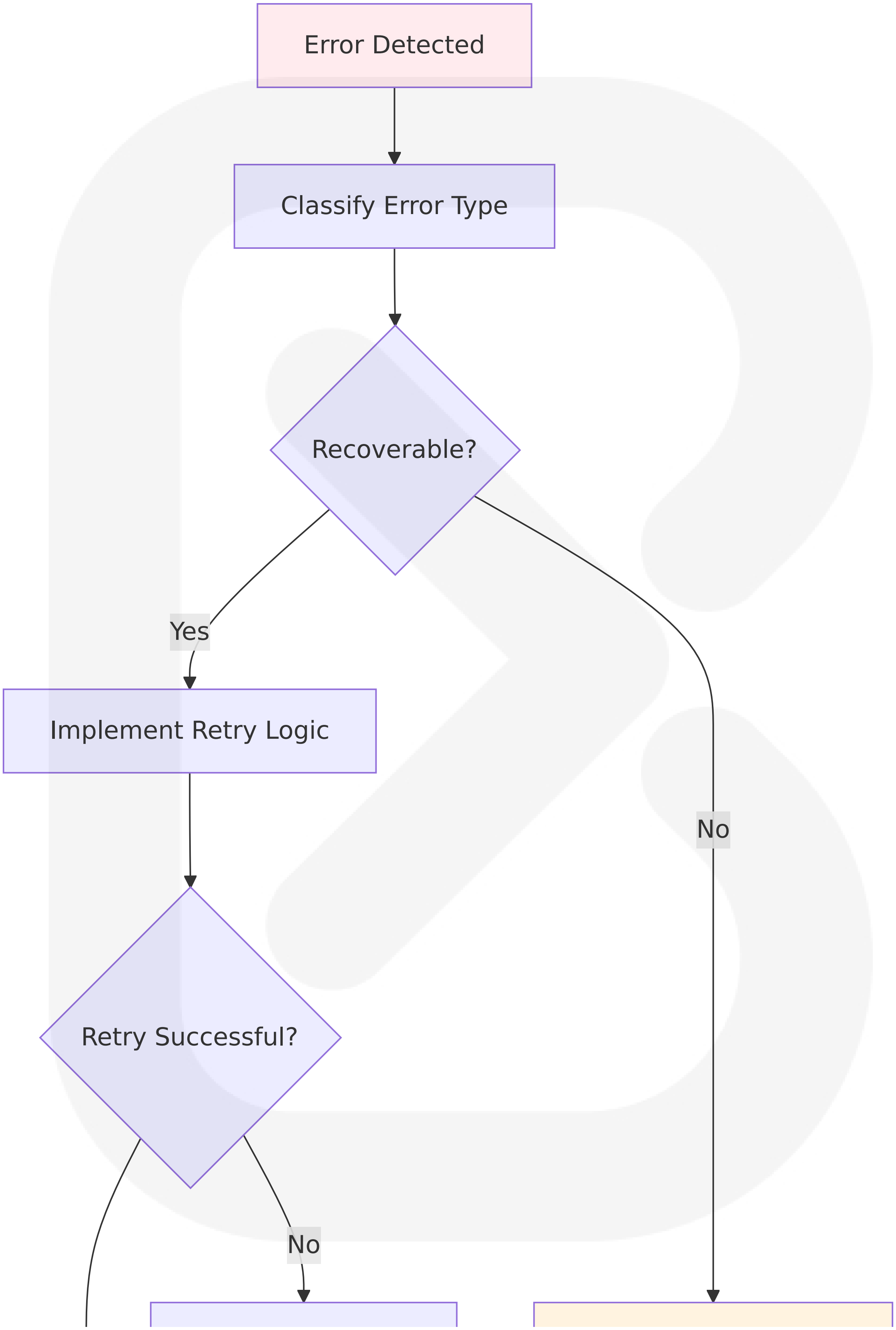
5.4.2 Logging and Tracing Strategy

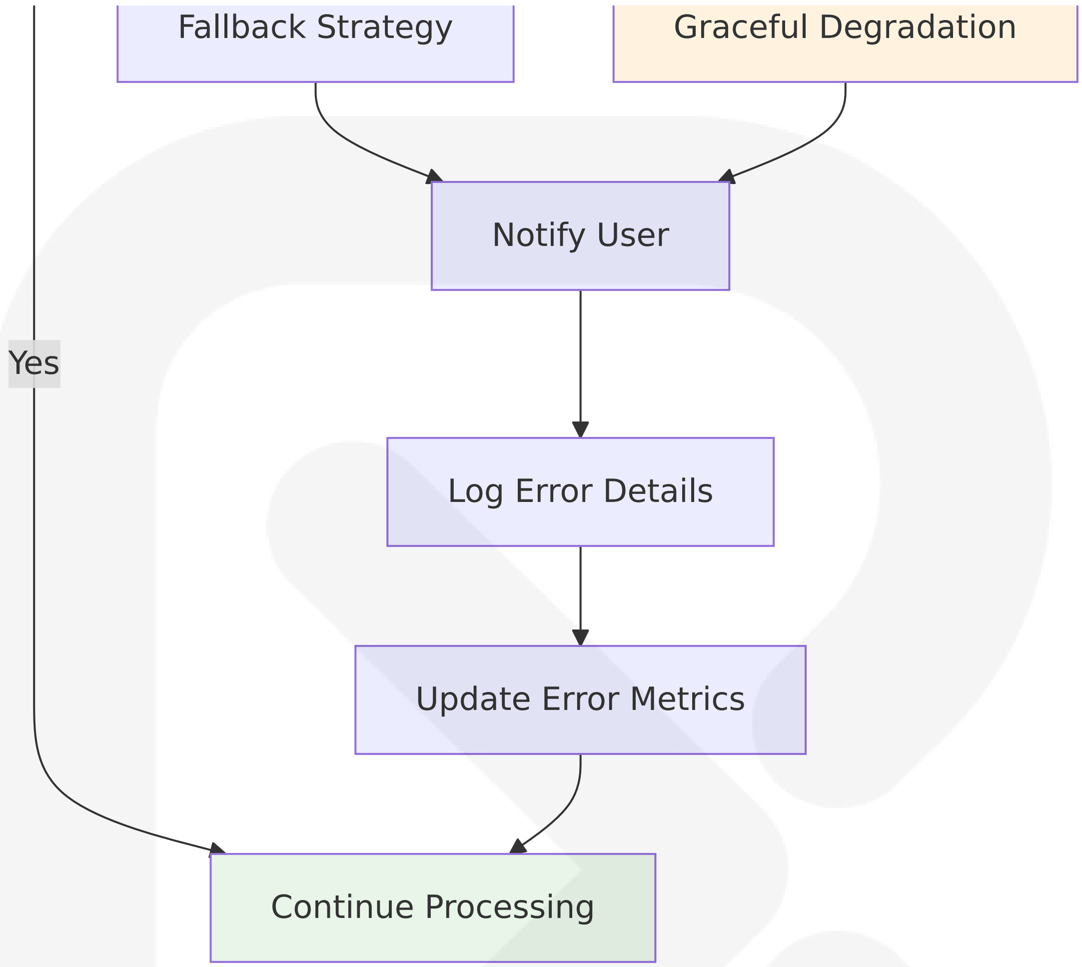
Log Level	Purpose	Retention	Privacy Considerations
DEBUG	Development troubleshooting	7 days	No circuit data logged
INFO	System operations	30 days	Anonymized performance metrics
WARN	Recoverable errors	90 days	Error context without sensitive data
ERROR	System failures	1 year	Stack traces with data sanitization

Distributed Tracing:

- **Correlation IDs**: Unique identifiers for tracking requests across components
- **Span Tracking**: Detailed timing information for performance optimization
- **Context Propagation**: Automatic context passing between system components
- **Privacy Protection**: No logical circuit data included in trace information

5.4.3 Error Handling Patterns


Error Detected
Classify Error Type
Recoverable?
Yes
No
Implement Retry Logic
Retry Successful?
No



Error Handling Principles:

- **Fail-Safe Design**: System continues operation with reduced functionality rather than complete failure
- **Automatic Recovery**: Intelligent retry mechanisms with exponential backoff
- **User Communication**: Clear error messages with actionable guidance
- **Learning from Failures**: Error patterns inform system improvements

5.4.4 Authentication and Authorization Framework

Local Security Model:

- **No External Authentication**: Desktop application operates with local user permissions
- **Configuration Protection**: Digital signatures prevent unauthorized configuration changes
- **Model Integrity**: Cryptographic validation of RL model checkpoints
- **Audit Logging**: Comprehensive logging of all system operations

External Integration Security:

- **API Key Management**: Secure storage of quantum hardware provider credentials
- **Token Rotation**: Automatic refresh of authentication tokens
- **Encrypted Communication**: TLS 1.3 for all external API communications
- **Credential Isolation**: Hardware credentials stored separately from application data

5.4.5 Performance Requirements and SLAs

Component	Performance Target	Measurement Method	SLA Commitment
Circuit Compilation	<5 seconds for d=3 patches	Automated benchmarking	95th percentile
RL Training Convergence	<10^5 steps	Training curve analysis	Average performance
GUI Responsiveness	<5ms gate placement	UI event timing	99th percentile
Hardware Integration	<10s job submission	API response timing	90th percentile

Performance Monitoring:

- **Real-time Metrics**: Continuous performance tracking with alerting
- **Benchmark Suites**: Automated performance regression testing

- **User Experience Metrics**: Response time tracking for interactive operations
- **Scalability Testing**: Performance validation across different circuit sizes

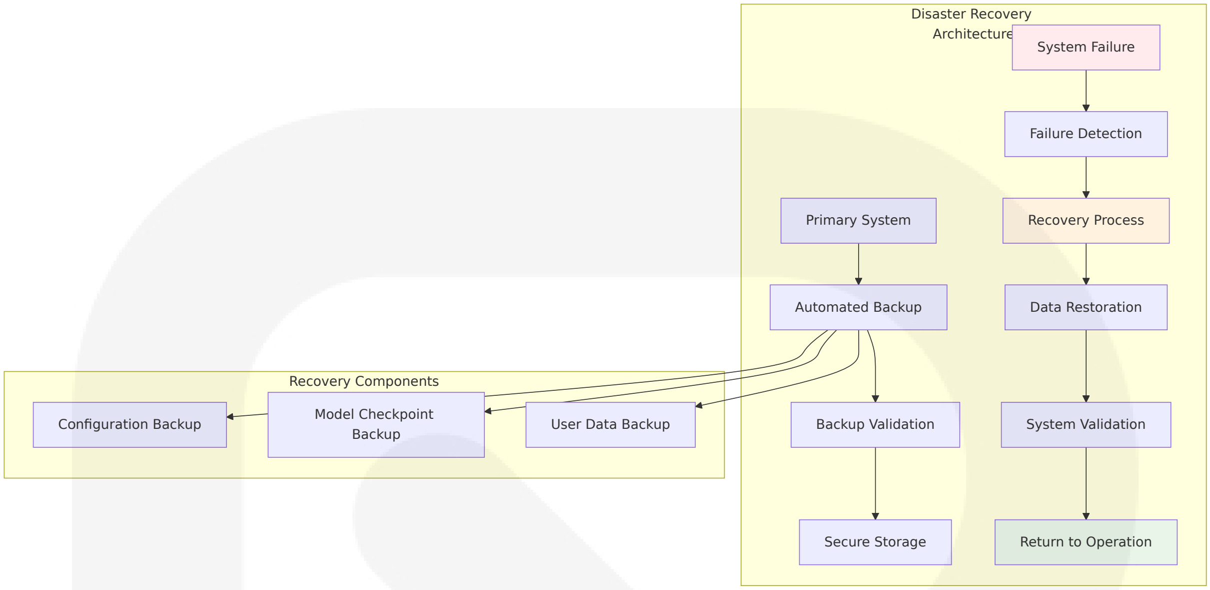
5.4.6 Disaster Recovery Procedures

Data Protection Strategy:

- **Automatic Backups**: Incremental backups of configuration and model data
- **Version Control**: Git-based versioning for configuration files
- **Recovery Testing**: Regular validation of backup and recovery procedures
- **Data Integrity**: Checksums and validation for all critical data

System Recovery Procedures:

- **Graceful Shutdown**: Proper cleanup and state preservation during shutdown
- **Crash Recovery**: Automatic recovery from unexpected system failures
- **Configuration Rollback**: Ability to revert to previous working configurations
- **Model Recovery**: Restoration of RL models from validated checkpoints



Recovery Time Objectives:

- **Configuration Recovery**: <5 minutes for complete configuration restoration
- **Model Recovery**: <15 minutes for RL model checkpoint restoration
- **Full System Recovery**: <30 minutes for complete system restoration
- **Data Integrity Validation**: <10 minutes for comprehensive data validation

6. SYSTEM COMPONENTS DESIGN

6.1 COMPONENT ARCHITECTURE

6.1.1 Core Component Overview

The QCraft system architecture employs a **modular, plugin-based design** with clear separation of concerns across five primary layers. Each component is designed for extensibility while maintaining strict privacy boundaries and performance requirements.

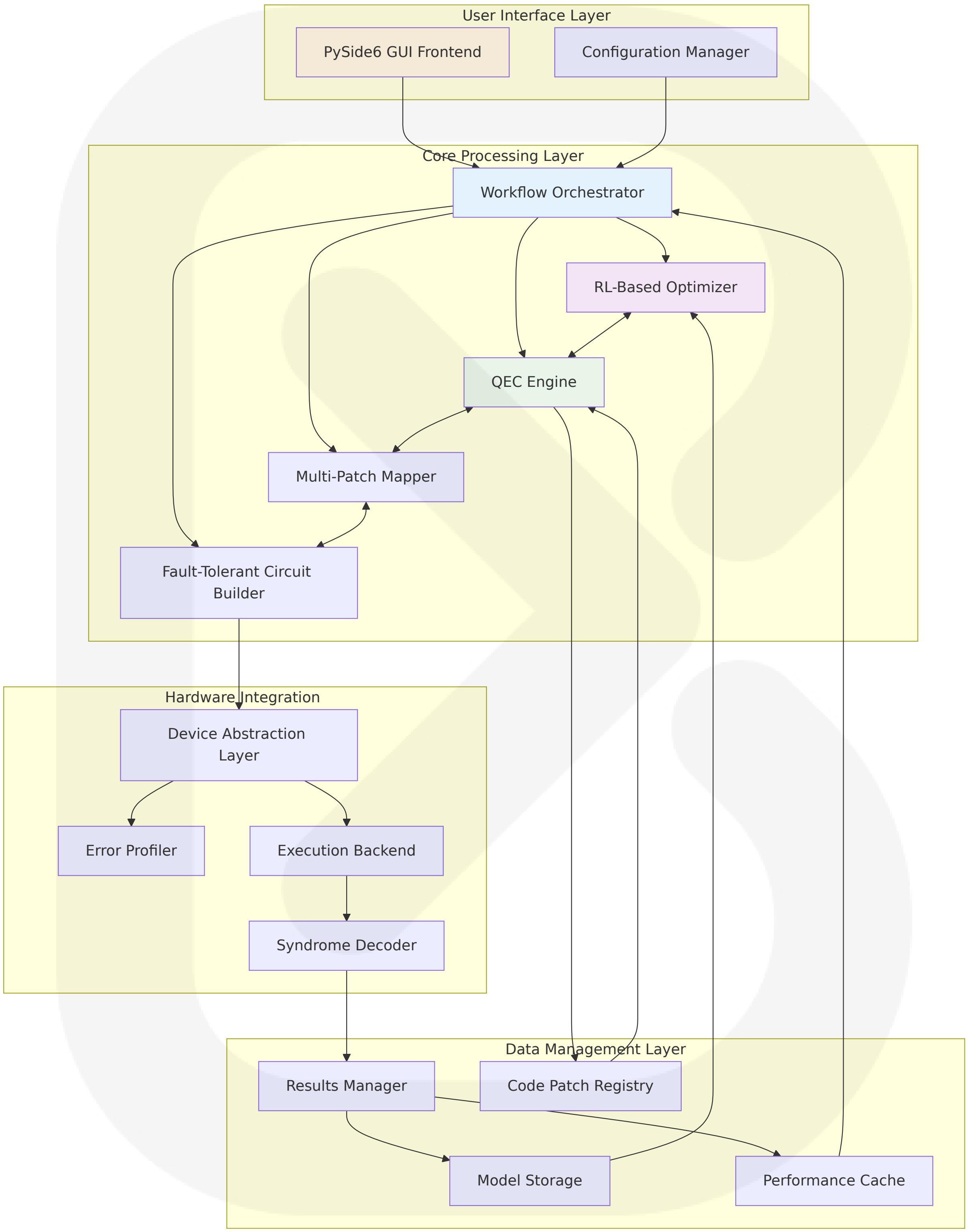
Architectural Principles:

- **Privacy by Design**: PySide6 is the official Python module from the Qt for Python project, which provides access to the complete Qt 6.0+ framework enabling local-only processing
- **Adaptive Learning**: The agent, trained using the Proximal Policy Optimization (PPO) algorithm, employs Graph Neural Networks to approximate the policy and value functions
- **Hardware Agnostic**: Unified abstraction supporting multiple quantum platforms
- **Configuration-Driven**: YAML/JSON-based parameter management with schema validation

6.1.2 Component Interaction Matrix

Component	PySide6 GUI	RL Optimizer	QEC Engine	Multi-Patch Mapper	Hardware Layer
PySide6 GUI	-	Configuration	Circuit Submission	Visualization	Status Updates
RL Optimizer	Progress Updates	-	Policy Evaluation	Action Selection	Performance Feedback
QEC Engine	Error Reporting	Code Selection	-	Patch Definitions	Resource Requirements
Multi-Patch Mapper	Mapping Display	Reward Calculation	Constraint Validation	-	Topology Queries
Hardware Layer	Device Status	Execution Results	Error Rates	Connectivity Data	-

6.1.3 Data Flow Architecture



6.2 DETAILED COMPONENT SPECIFICATIONS

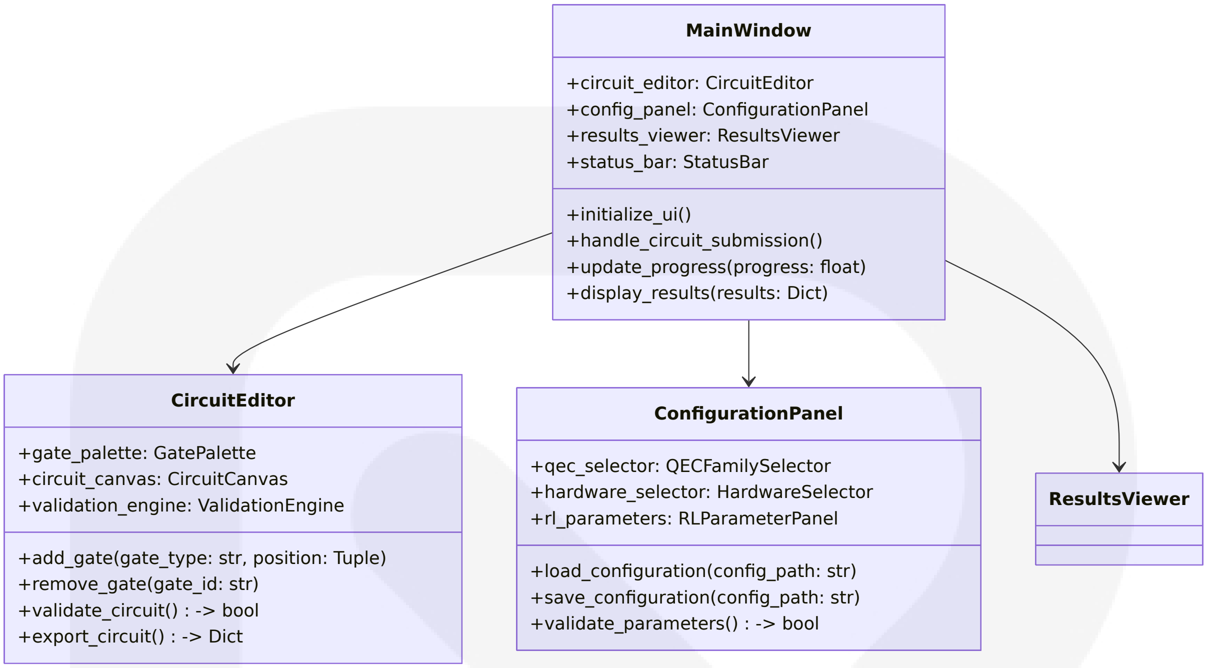
6.2.1 PySide6 GUI Frontend

Component Purpose and Scope:
The GUI frontend provides an intuitive interface for quantum circuit design, visualization, and system configuration. Behind the hood, PySide6 is a wrapper to Qt6, the latest version of a UI framework called Qt, enabling native desktop application capabilities across platforms.

Technical Architecture:

Aspect	Specification	Implementation Details
Framework	PySide6 6.9.2	PySide6 is the official Python module from the Qt for Python project, which provides access to the complete Qt 6.0+ framework
Architecture Pattern	Model-View-Controller	The Qt ModelView architecture simplifies the linking and updating your UI with data in custom formats or from external sources
Threading Model	Qt Event Loop	Asynchronous processing with signals/slots
Memory Management	Automatic Reference Counting	Qt's parent-child object hierarchy

Key Interfaces and APIs:



Data Persistence Requirements:

- **Session State**: In-memory storage using Qt's QSettings for user preferences
- **Project Files**: JSON-based circuit definitions with schema validation
- **Configuration Cache**: YAML files with automatic backup and versioning
- **Recent Projects**: SQLite database for project history and metadata

Performance Specifications:

- **UI Responsiveness**: <5ms response time for gate placement operations
- **Memory Usage**: <500MB for typical circuit designs (up to 20 logical qubits)
- **Startup Time**: <3 seconds for application initialization
- **Cross-Platform**: Native performance on Windows, macOS, and Linux

6.2.2 RL-Based Optimizer

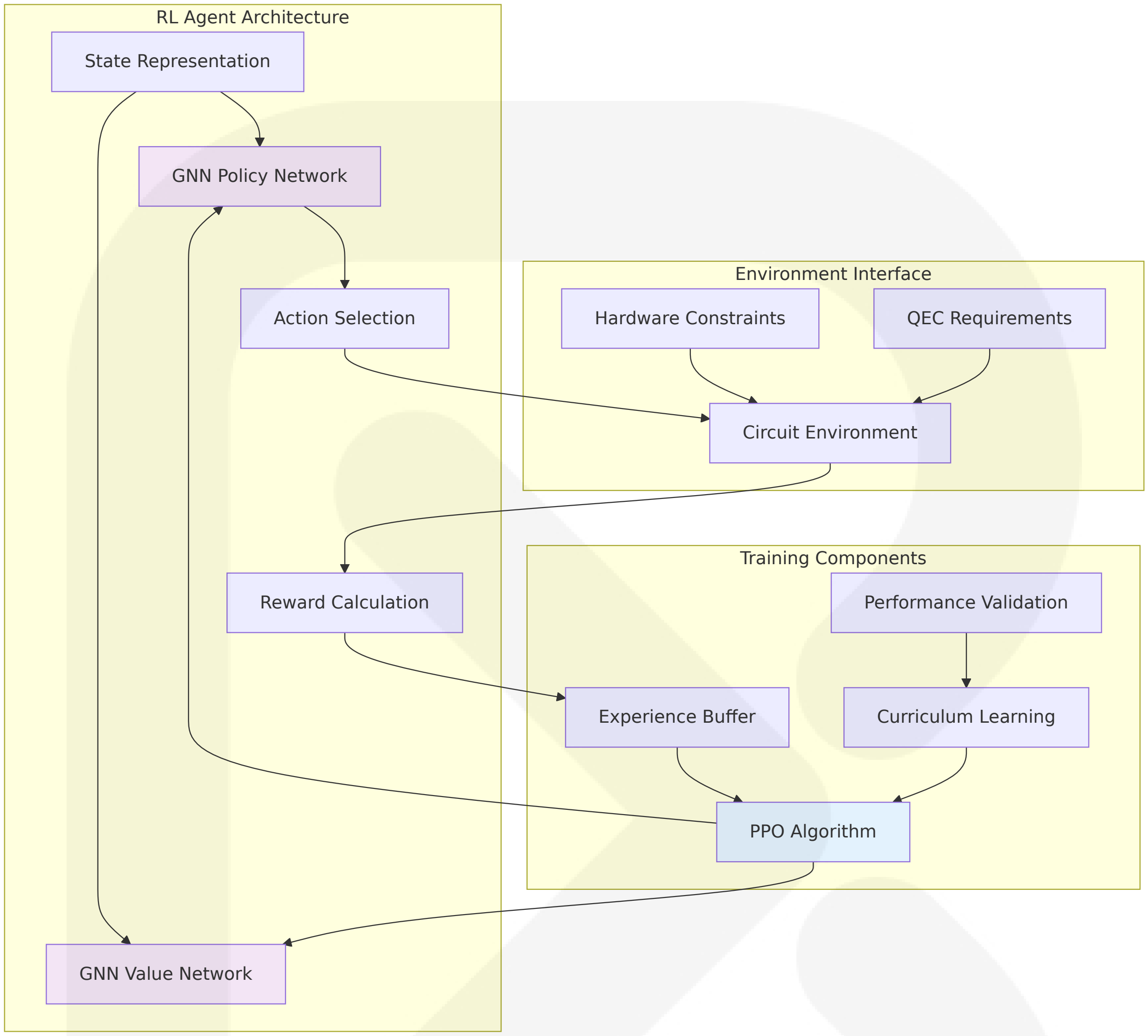
Component Purpose and Scope:

The RL optimizer implements Proximal Policy Optimization (PPO) algorithm, employs Graph Neural Networks to approximate the policy and value functions for quantum circuit optimization. After training on small Clifford+T circuits of 5-qubits and few tenths of gates, the agent consistently improves the state-of-the-art for this type of circuits, for at least up to 80-qubit and 2100 gates.

Technical Architecture:

Component	Technology	Purpose	Performance Target
Policy Network	Graph Neural Networks	Action probability distribution	<10ms inference time
Value Network	Graph Neural Networks	State value estimation	<5ms evaluation time
Training Engine	Stable-Baselines3 PPO	Policy optimization	<10^5 steps convergence
Experience Buffer	Custom Ring Buffer	Training data storage	1M transitions capacity

Reinforcement Learning Configuration:



Multi-Objective Reward Function:

The reward function balances multiple optimization objectives with configurable weights:

Reward Component	Weight Range	Purpose	Implementation
Valid Mapping	10.0	Ensure feasible solutions	Binary indicator function
Invalid Mapping	-20.0	Penalize constraint violations	Overlap detection
Connectivity Bonus	2.0	Reward hardware compatibility	Graph connectivity metrics

Reward Component	Weight Range	Purpose	Implementation
Resource Utilization	0.5	Optimize qubit usage	Hardware utilization ratio
Error Rate Bonus	1.0	Minimize logical error rates	Empirical error estimation

Curriculum Learning Stages:

1. **Structure Mastery** (5-qubit circuits): Basic gate placement and connectivity
2. **Hardware Adaptation** (10-20 qubits): Device-specific constraints and topology
3. **Noise-Aware Optimization** (20+ qubits): Real-world error rates and decoherence

6.2.3 QEC Engine

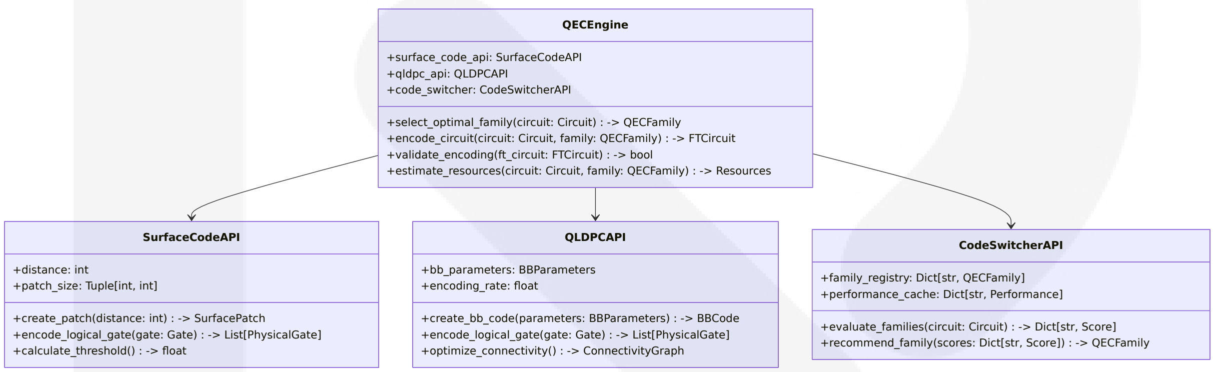
Component Purpose and Scope:
Multi-family quantum error correction implementation supporting both surface codes and Bivariate Bicycle (BB) codes. Our new codes lend themselves better to practical implementation because each qubit needs only to connect to six others, and the connections can be routed on just two layers.

QEC Family Support:

QEC Family	Code Parameters	Resource Requirements	Performance Characteristics
Surface Codes	Distance 3, 5, 7	~3,000 qubits for 12 logical	Well-established, high threshold
qLDPC Codes	[[144, 12, 12]] BB	288 qubits for 12 logical	High-threshold and low-overhead fault-tolerant quant

QEC Family	Code Parameters	Resource Requirements	Performance Characteristics
			um memory. Nature 627, 778–782 (2024)
Hypergraph Product	Variable parameters	Configurable overhead	Research-grade implementation

Technical Implementation:



Performance Specifications:

- **Encoding Time**: <5 seconds for distance-3 surface code patches
- **Resource Estimation**: <1 second for circuit analysis and family recommendation
- **Memory Usage**: <1GB for storing code definitions and patch configurations
- **Scalability**: Support for up to 20 logical qubits with multi-patch configurations

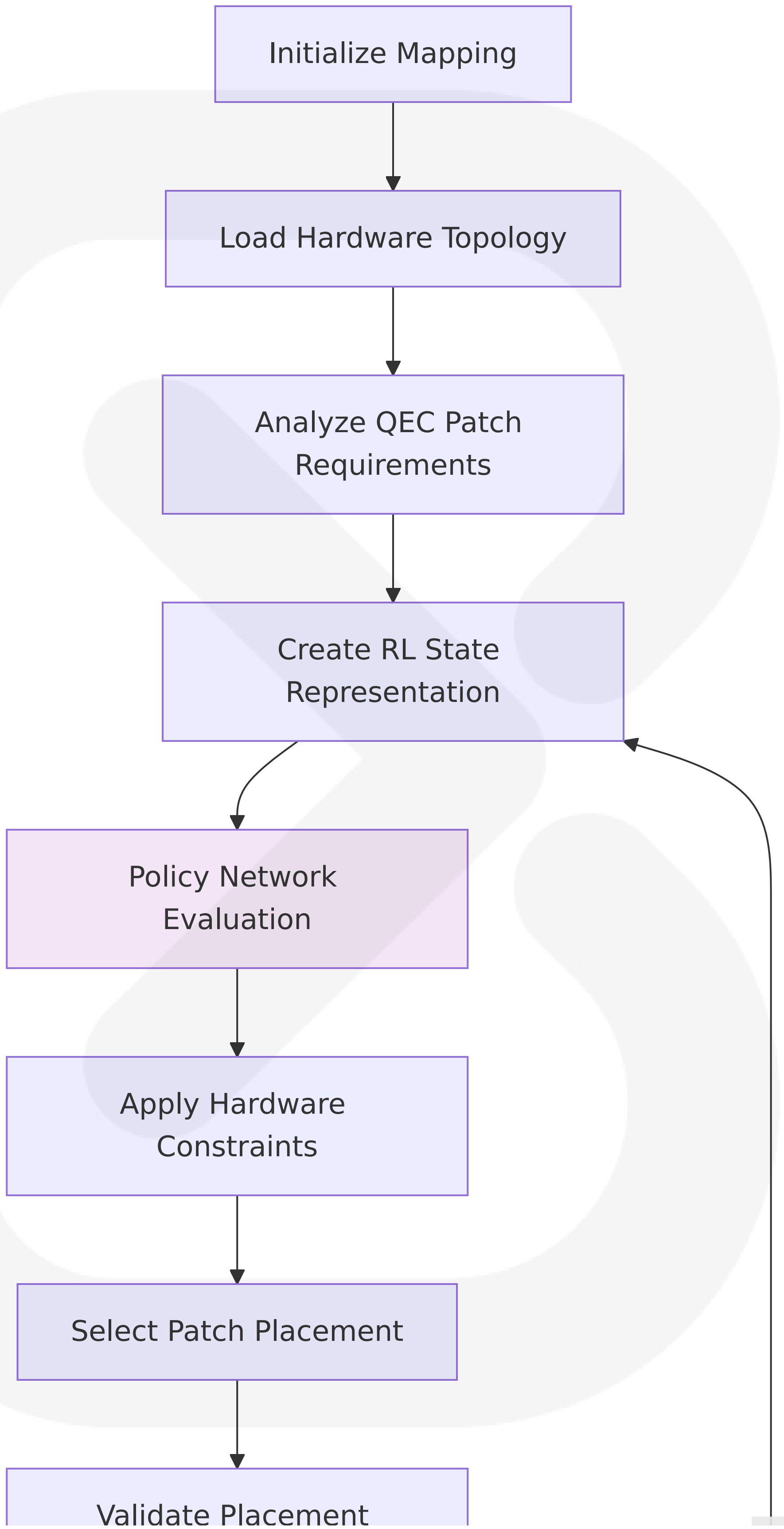
6.2.4 Multi-Patch Mapper

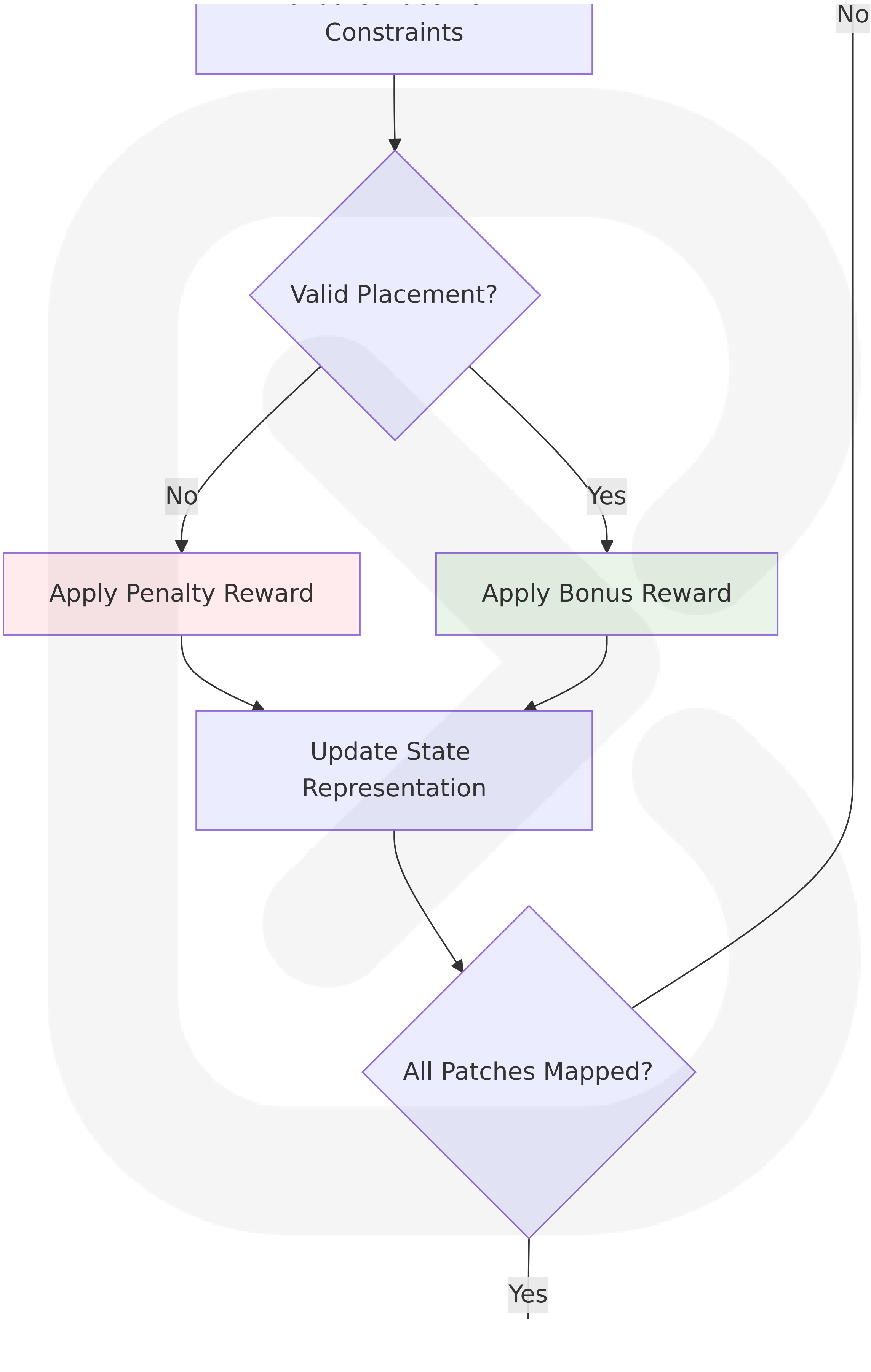
Component Purpose and Scope:
Intelligent placement of multiple QEC patches onto quantum hardware topologies with connectivity optimization and resource constraint satisfaction. The mapper uses RL-based strategies to find optimal patch placements.

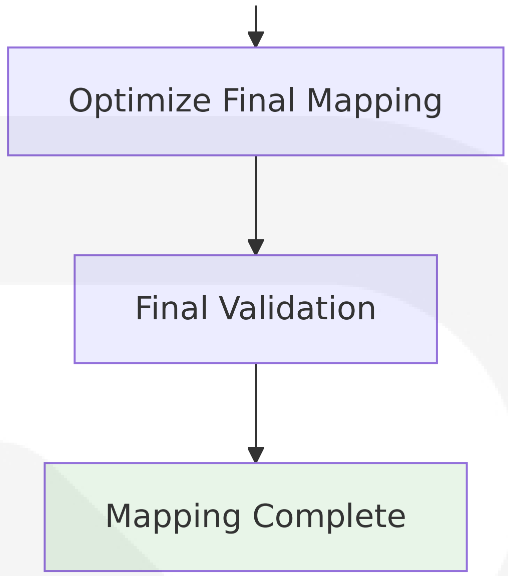
Technical Architecture:

Component	Implementation	Purpose	Performance Target
Graph Processor	NetworkX 3.5	Hardware topology representation	<100ms topology analysis
Constraint Solver	Custom CSP Engine	Hardware compatibility validation	<500ms constraint checking
RL Integration	PPO-based Mapping	Policy-driven placement decisions	<10s mapping for 20 qubits
Optimization Engine	Multi-objective Optimizer	Resource utilization maximization	95% hardware utilization

Mapping Algorithm Flow:


Initialize Mapping
Load Hardware Topology
Analyze QEC Patch Requirements
Create RL State Representation
Policy Network Evaluation
Apply Hardware Constraints
Select Patch Placement
Validate Placement


Constraints
No
Valid Placement?
No
Yes
Apply Penalty Reward
Apply Bonus Reward
Update State Representation
All Patches Mapped?
Yes



Constraint Validation System:

- **Connectivity Constraints**: Ensure all patch qubits are reachable within hardware topology
- **Resource Constraints**: Validate sufficient physical qubits for all logical requirements
- **Overlap Detection**: Prevent multiple patches from claiming the same physical qubits
- **Distance Requirements**: Maintain minimum separation between independent patches

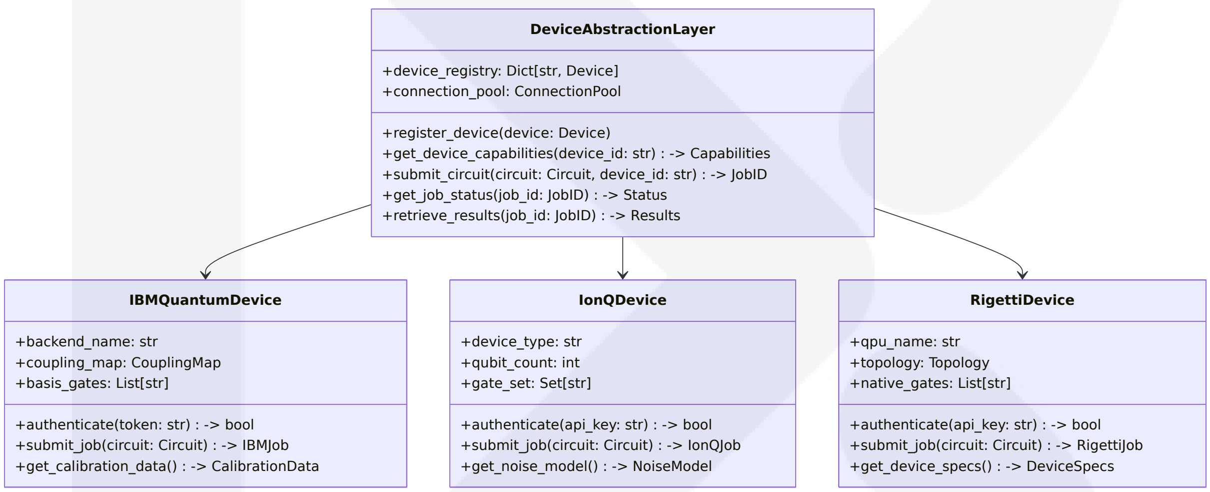
6.2.5 Hardware Integration Layer

Component Purpose and Scope:
Unified interface supporting multiple quantum hardware platforms with standardized device specifications and execution protocols. The layer abstracts hardware differences while preserving platform-specific optimizations.

Supported Hardware Platforms:

Platform	Integration Method	Authentication	Execution Model
IBM Quantum	REST API	API Token	Asynchronous job submission
IonQ	REST API	API Key	Batch processing
Rigetti Forest	SDK Integration	API Key	Direct circuit execution
AWS Braket	Boto3 SDK	AWS Credentials	Multi-vendor access

Device Abstraction Architecture:



Error Profiler Integration:

- **Empirical Noise Modeling**: Statistical analysis of execution results to build device-specific noise models
- **Calibration Data Integration**: Automatic incorporation of provider calibration data
- **Performance Tracking**: Continuous monitoring of device performance metrics
- **Adaptive Optimization**: Real-time adjustment of compilation strategies based on observed performance

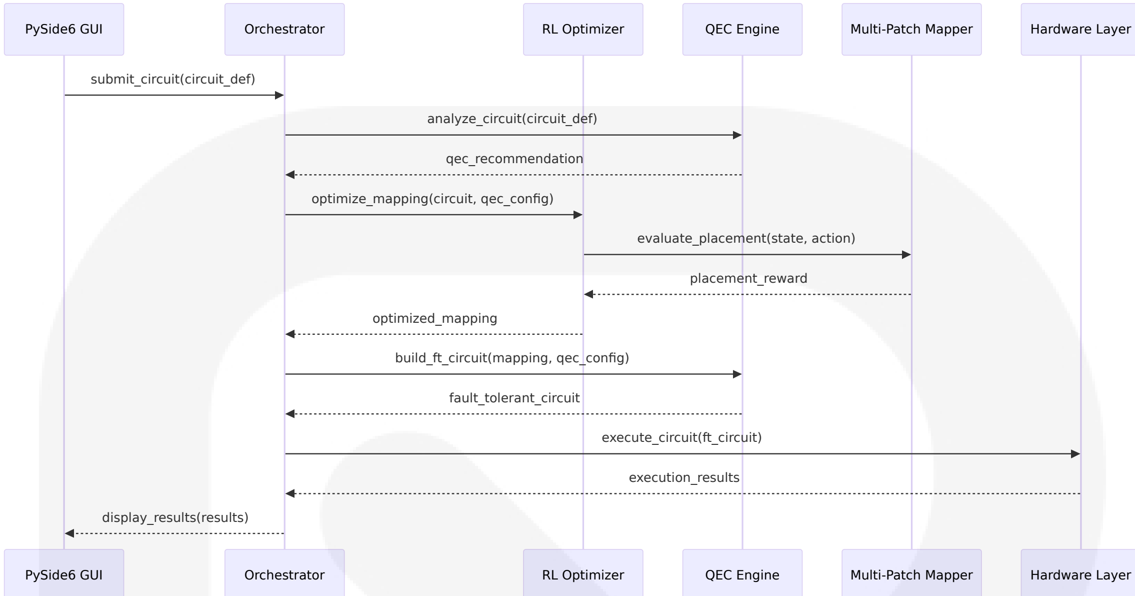
6.3 COMPONENT INTERFACES

6.3.1 Inter-Component Communication Protocols

Event-Driven Architecture:
The system employs Qt's signals/slots mechanism for real-time communication between components, ensuring loose coupling and responsive user interfaces.

Signal Type	Source Component	Target Component	Data Payload	Frequency
CircuitSubmitted	PySide6 GUI	Workflow Orchestrator	Circuit Definition	On-demand
OptimizationProgress	RL Optimizer	PySide6 GUI	Progress Percentage	Real-time
QECFamilySelected	QEC Engine	Multi-Patch Mapper	Family Configuration	Per circuit
MappingComplete	Multi-Patch Mapper	Fault-Tolerant Builder	Patch Placement	Per optimization
ExecutionResults	Hardware Layer	Results Manager	Measurement Data	Per job

API Standardization:



6.3.2 Data Exchange Formats

Circuit Representation Standards:

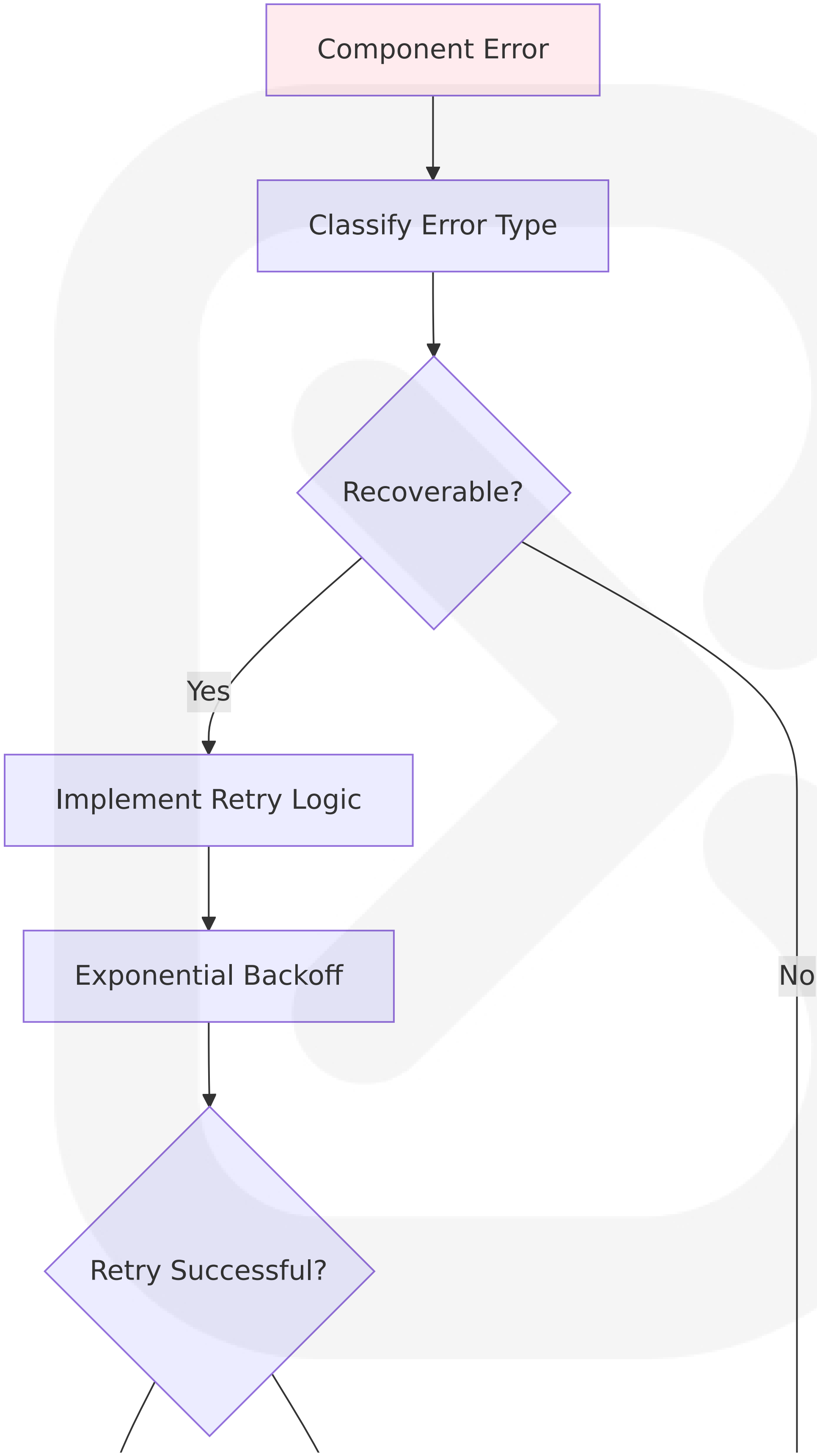
- **Logical Circuits**: JSON format with schema validation for gate sequences and qubit mappings
- **Fault-Tolerant Circuits**: Extended JSON format including stabilizer information and syndrome measurement schedules
- **Hardware Circuits**: Platform-specific formats (QASM 3.0, Quil, etc.) with automatic translation

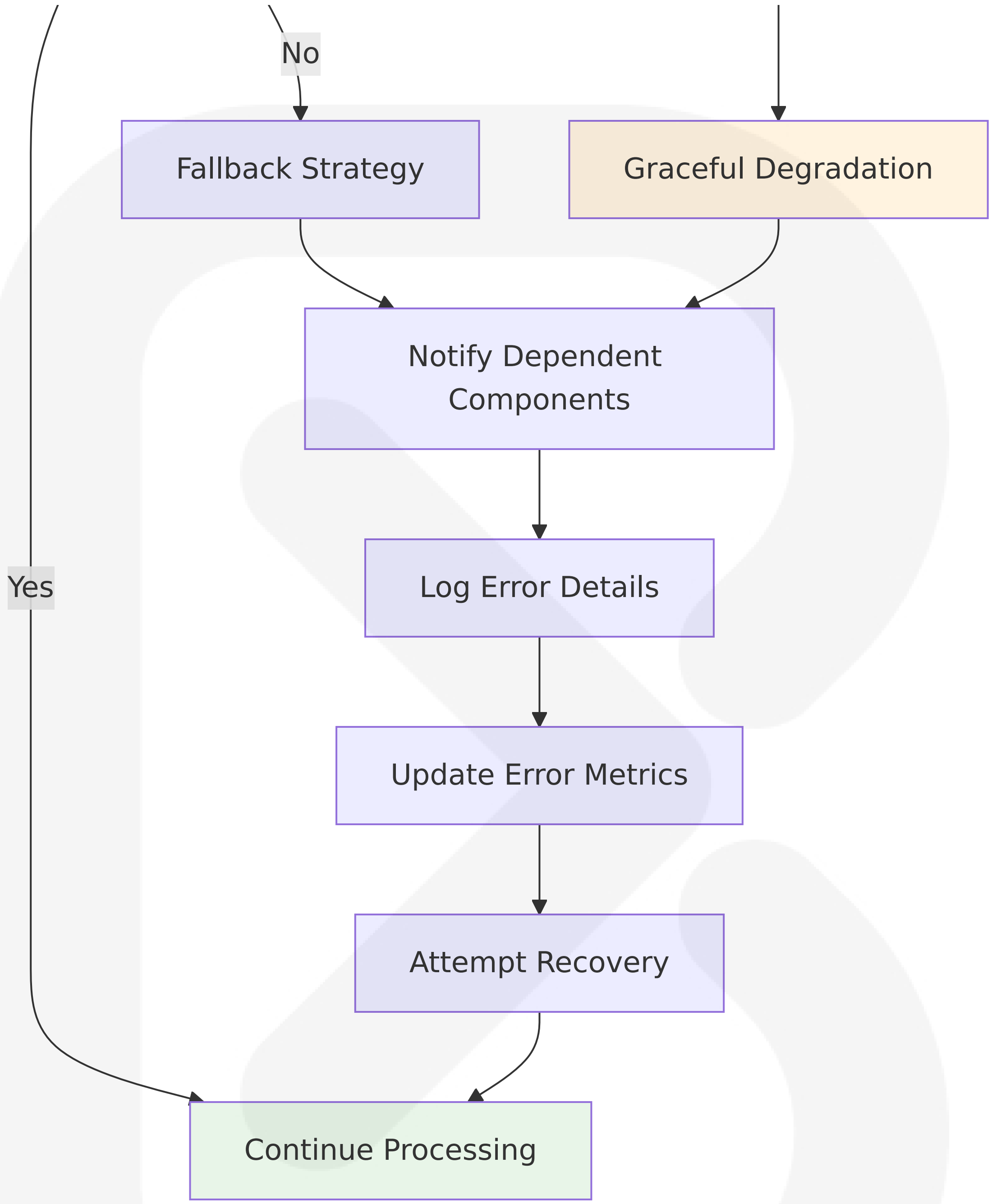
Configuration Management:

- **YAML Configuration Files**: Human-readable parameter specifications with hierarchical organization
- **JSON Schema Validation**: Automatic validation of configuration parameters against predefined schemas
- **Version Control Integration**: Git-compatible configuration files with diff-friendly formatting

6.3.3 Error Handling and Recovery

Component-Level Error Handling:


Component Error
Classify Error Type
Recoverable?
Yes
No
Implement Retry Logic
Exponential Backoff
Retry Successful?



Error Recovery Strategies:

- **Circuit Validation Errors**: Return to circuit editor with specific error highlighting
- **Hardware Connection Errors**: Automatic fallback to simulator with user notification
- **RL Training Errors**: Checkpoint restoration with training resumption

- **QEC Encoding Errors**: Fallback to default parameters with warning messages

6.4 SCALABILITY AND PERFORMANCE

6.4.1 Performance Optimization Strategies

Component-Level Optimizations:

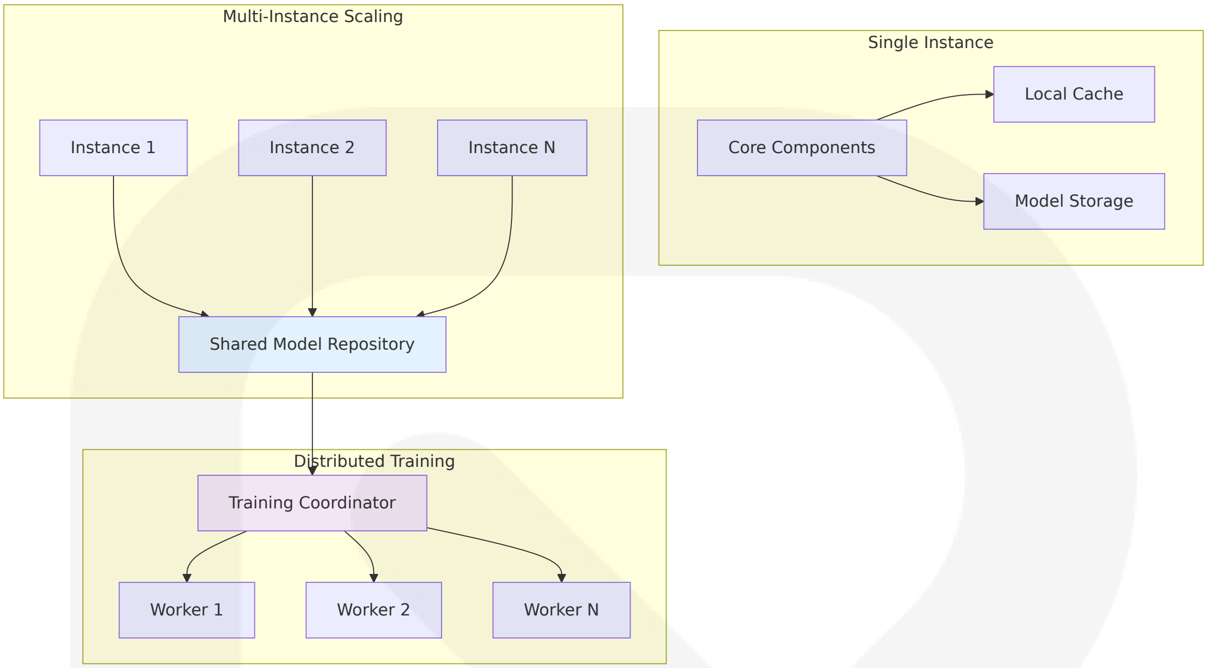
Component	Optimization Strategy	Performance Gain	Implementation
PySide6 GUI	Lazy Loading	50% faster startup	On-demand widget creation
RL Optimizer	Model Caching	80% faster inference	Pre-loaded model checkpoints
QEC Engine	Parallel Processing	3x faster encoding	Multi-threaded stabilizer calculations
Multi-Patch Mapper	Graph Caching	60% faster mapping	Pre-computed topology analysis

Memory Management:

- **Intelligent Caching**: LRU eviction policies with configurable cache sizes
- **Memory Pooling**: Pre-allocated memory pools for frequent operations
- **Garbage Collection**: Explicit cleanup of large data structures
- **Streaming Processing**: Chunked processing for large quantum circuits

6.4.2 Scalability Architecture

Horizontal Scaling Capabilities:



Vertical Scaling Optimizations:

- **Multi-Threading**: Parallel processing of independent circuit optimizations
- **GPU Acceleration**: CUDA/OpenCL support for Graph Neural Network training
- **Memory Optimization**: Efficient data structures and memory-mapped files
- **CPU Optimization**: Vectorized operations and SIMD instruction utilization

6.4.3 Resource Management

Dynamic Resource Allocation:

- **Adaptive Memory Limits**: Automatic adjustment based on available system resources
- **Priority-Based Scheduling**: Critical operations receive higher resource priority
- **Resource Monitoring**: Real-time tracking of CPU, memory, and GPU utilization

- **Throttling Mechanisms**: Automatic throttling during resource contention

Performance Monitoring:

- **Real-Time Metrics**: Continuous monitoring of component performance
- **Bottleneck Detection**: Automatic identification of performance bottlenecks
- **Performance Profiling**: Built-in profiling tools for optimization
- **Alerting System**: Notifications for performance degradation

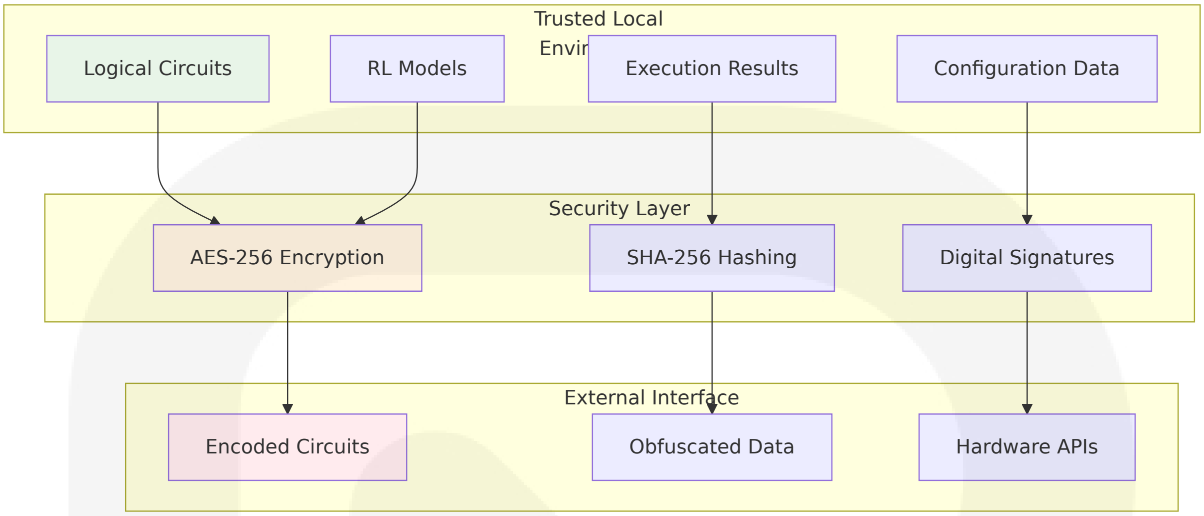
6.5 SECURITY AND PRIVACY

6.5.1 Privacy-Preserving Architecture

Local Processing Guarantees:

- **Circuit Isolation**: All logical circuits remain within the local desktop environment
- **Encrypted Export**: Only fault-tolerant, encoded circuits are exported with AES-256 encryption
- **No Cloud Dependencies**: Core functionality operates without external service dependencies
- **Data Minimization**: Only essential data is transmitted to external systems

Security Boundaries:



6.5.2 Data Protection Mechanisms

Encryption Standards:

- **AES-256**: Symmetric encryption for circuit data and model checkpoints
- **RSA-4096**: Asymmetric encryption for key exchange and authentication
- **SHA-256**: Cryptographic hashing for data integrity verification
- **HMAC**: Message authentication codes for tamper detection

Access Control:

- **Local User Permissions**: Integration with operating system access controls
- **Configuration Protection**: Digital signatures prevent unauthorized parameter changes
- **Model Integrity**: Cryptographic validation of RL model checkpoints
- **Audit Logging**: Comprehensive logging of all security-relevant operations

6.5.3 Compliance and Governance

Privacy Compliance:

- **GDPR Compliance**: No personal data collection or processing
- **Data Residency**: All sensitive data remains on local systems
- **Right to Deletion**: Complete local data removal capabilities
- **Transparency**: Clear documentation of all data processing activities

Security Auditing:

- **Code Auditing**: Regular security reviews of all components
- **Dependency Scanning**: Automated vulnerability scanning of third-party libraries
- **Penetration Testing**: Regular security assessments of the complete system
- **Incident Response**: Defined procedures for security incident handling

6.1 CORE SERVICES ARCHITECTURE

6.1.1 Architecture Applicability Assessment

Core Services Architecture is not applicable for this system due to the fundamental architectural design principles and requirements of QCraft.

QCraft is designed as a **desktop-based, modular monolithic application** rather than a distributed services architecture. This architectural decision is driven by several critical factors:

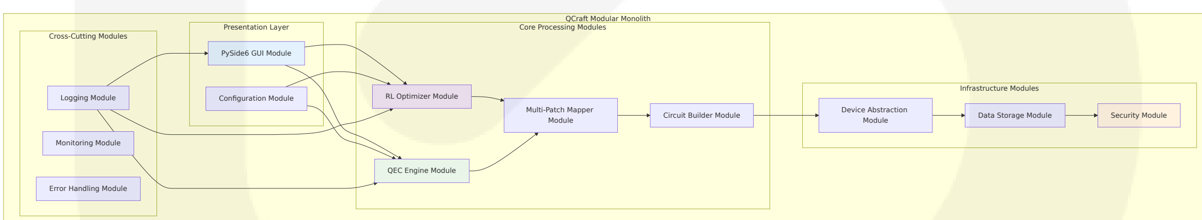
6.1.2 Architectural Rationale

Factor	Monolithic Justification	Services Architecture Limitations
Privacy Requirements	All logical circuits must remain in local with no external transmission	Services would require network communication, violating privacy constraints
Performance Targets	Performance overhead of serializing data and sending across network has noticeable impact	Sub-5-second compilation requirements incompatible with network latency
Deployment Model	Single physical deployment unit suitable for desktop applications	Multiple service deployments inappropriate for desktop software
System Complexity	Modular monoliths are easier to manage than hundreds of microservices with lower operational costs	Distributed systems complexity unnecessary for single-user desktop application

6.1.3 Modular Monolithic Design Principles

QCraft employs a **modular monolithic architecture** that provides the benefits of modularity without the complexity of distributed services:

Module Organization Strategy



Modular Benefits Without Services Complexity

Benefit	Implementation	Advantage Over Services
Loose Coupling	Modules communicate through public APIs with well-defin	No network serialization overhead

Benefit	Implementation	Advantage Over Services
	ed boundaries	
High Cohesion	Business logic encapsulated in modules enabling high reusability	Shared memory access for performance
Independent Development	Developer teams can work on different modules with minimum impact	Single deployment simplifies integration
Future Extensibility	Clear path to microservices when need arises for independent deployment	Gradual evolution without immediate complexity

6.1.4 Alternative Architecture Considerations

Why Not Microservices?

Google identified five main challenges with microservices: Performance overhead, correctness difficulties in distributed systems, and management complexity of multiple applications. For QCraft's desktop application context, these challenges outweigh the benefits:

Performance Impact: The overhead of serializing data and sending it across the network has a noticeable impact on performance, which conflicts with QCraft's sub-5-second compilation requirements.

Correctness Complexity: It's difficult to reason about correctness of a distributed system when there are many interactions between components, particularly problematic for quantum circuit optimization where correctness is paramount.

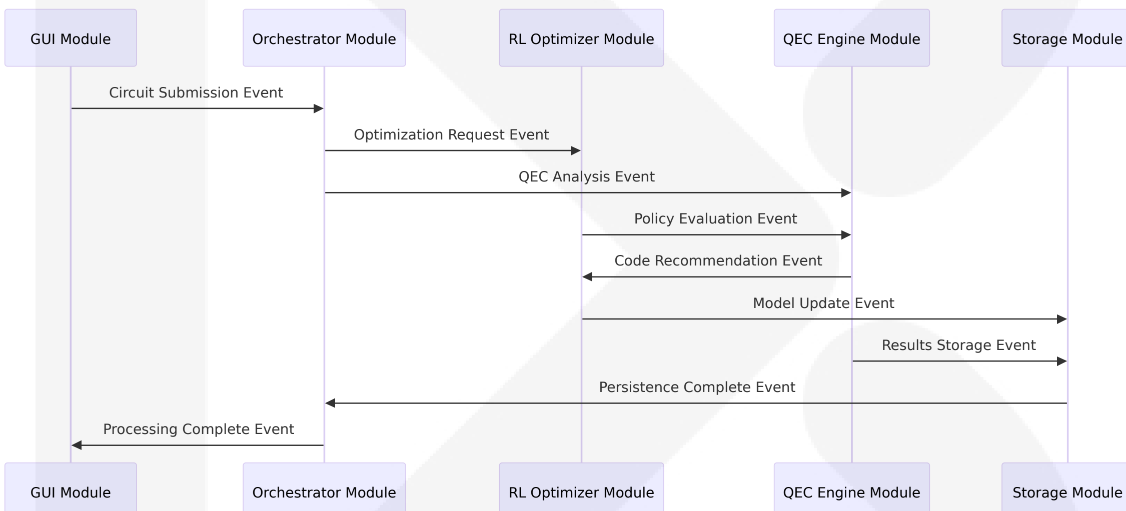
Management Overhead: Managing multiple different applications, each with its release schedule is inappropriate for a single-user desktop application.

Modular Monolith Advantages

Modular monoliths provide high cohesion, low coupling, data encapsulation, and focus on business functionalities, while microservices add independent deployments and scalability that aren't needed for desktop applications.

6.1.5 Internal Module Communication Patterns

Event-Driven Module Communication



Module Interface Contracts

Interface Type	Implementation	Benefits
Synchronous APIs	Direct method calls with type safety	High performance, compile-time validation
Asynchronous Events	Qt signals/slots mechanism	Loose coupling, real-time UI updates
Configuration Injection	YAML-driven parameter passing	Runtime flexibility without service discovery

Interface Type	Implementation	Benefits
Shared Data Access	In-memory data structures	No serialization overhead, ACID consistency

6.1.6 Scalability Through Modular Design

Vertical Scaling Approach

Modularizing the monolith with subdomains organized into vertical slices consisting of presentation, business and persistent logic enables QCraft to scale functionality without distributed systems complexity:

Module-Level Scaling:

- **Threading**: Independent thread pools per module for parallel processing
- **Resource Allocation**: Dynamic memory and CPU allocation based on module workload
- **Caching**: Module-specific caching strategies optimized for each domain
- **Processing Pipelines**: Asynchronous processing within modules for non-blocking operations

Future Migration Path

Having a modular architecture allows easy extraction of modules into separate services when microservices become necessary. QCraft's modular design provides a clear migration path if distributed deployment becomes required in the future.

6.1.7 Operational Simplicity

Single Deployment Unit Benefits

Monolithic advantages include easy deployment of one executable file, simplified development with one code base, and better performance through centralized APIs.

Deployment Characteristics:

- **Single Executable**: Desktop application distributed as unified package
- **No Service Discovery**: Direct module references eliminate network discovery overhead
- **Unified Configuration**: Single configuration system across all modules
- **Simplified Monitoring**: Centralized logging and metrics collection

Development and Maintenance Advantages

Monolithic architecture simplifies development during early SDLC stages, deployment through copying packaged application, and provides single codebase for logging and configuration management.

6.1.8 Conclusion

QCraft's **modular monolithic architecture** is the optimal choice for this desktop-based quantum computing platform. A modular monolith is a single-deployment application with clear separation of concerns through modular design, emphasizing well-structured internal architecture with distinct modules.

This approach delivers the modularity benefits required for complex quantum circuit processing while maintaining the performance, privacy, and operational simplicity essential for a desktop quantum compiler. The architecture supports future evolution to distributed services if requirements change, while avoiding the premature complexity of microservices for a single-user desktop application.

6.2 DATABASE DESIGN

6.2.1 Database Design Applicability Assessment

QCraft employs a **hybrid data persistence strategy** that combines multiple storage technologies optimized for different data types and access patterns. Rather than relying on a single traditional database system, the architecture uses specialized storage solutions tailored to the unique requirements of quantum computing applications.

6.2.2 Storage Architecture Overview

The system implements a **multi-tier storage architecture** designed specifically for desktop applications with privacy-first requirements:

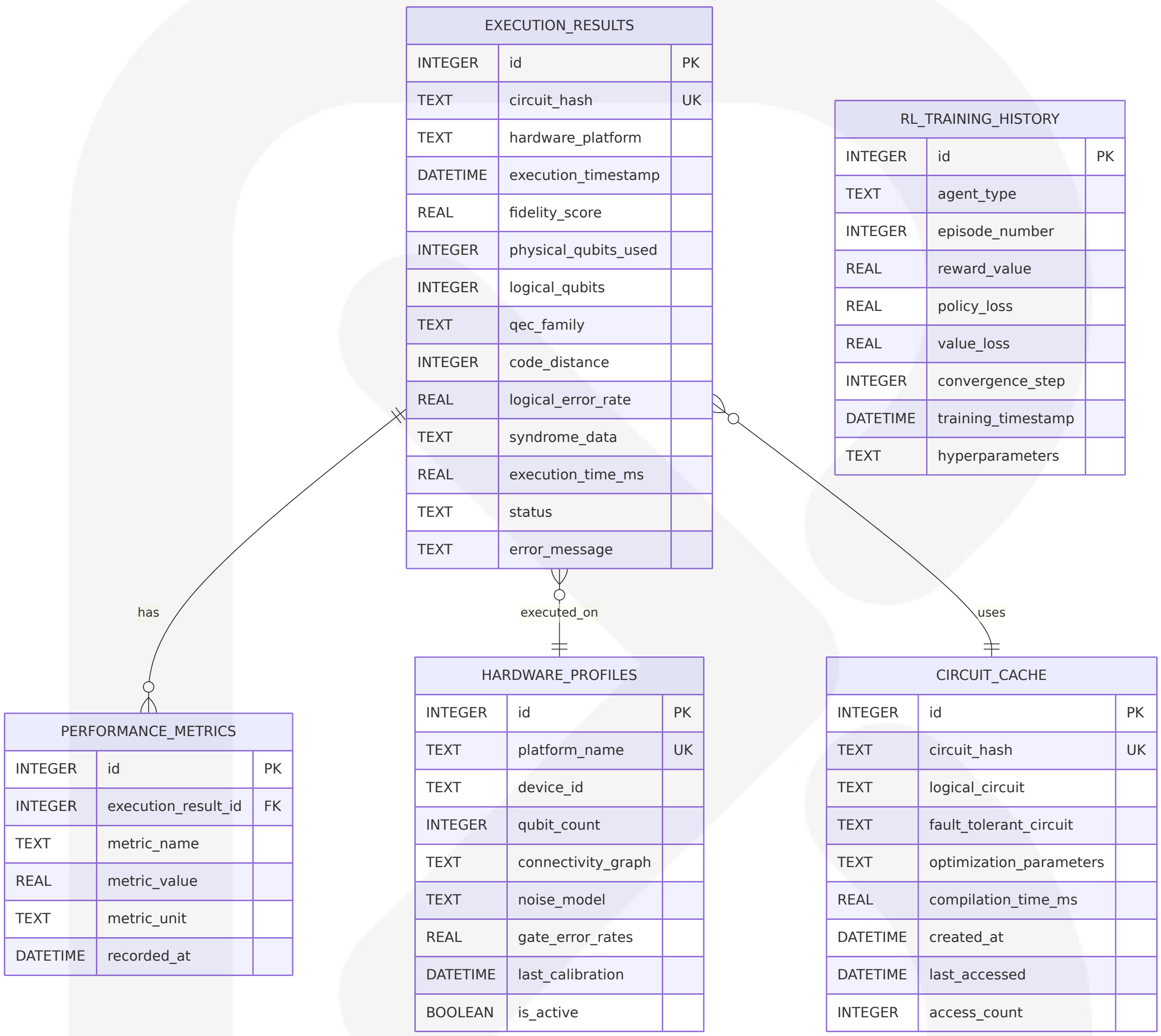
Storage Layer	Technology	Purpose	Data Types
Configuration Storage	YAML/JSON Files	Human-readable settings	User preferences, system parameters
Results Database	SQLite	Structured query support	Execution results, performance metrics
Model Storage	PyTorch Native	ML model persistence	RL agent checkpoints, neural networks
Circuit Library	JSON with Schema	Structured circuit data	Quantum circuit definitions, metadata

6.2.3 SQLite Database Design

SQLite is lightweight and self-contained. It's a code library without any other dependencies. There's nothing to configure. There's no database server. The client and the server run in the same process.

6.2.3.1 Schema Design

Entity Relationship Model



Core Tables Specification

Table	Primary Key	Indexes	Constraints
execution_results	id (INTEGER)	circuit_hash, execution_timestamp	UNIQUE(circuit_hash, hardware_platform)
performance_metrics	id (INTEGER)	execution_result_id, metric_name	FK to execution_results

Table	Primary Key	Indexes	Constraints
rl_training_history	id (INTEGER)	agent_type, episode_number	UNIQUE(agent_type, episode_number)
hardware_profiles	id (INTEGER)	platform_name, device_id	UNIQUE(platform_name, device_id)

6.2.3.2 Indexing Strategy

Performance-Optimized Indexes

```
-- Primary indexes for fast lookups
CREATE INDEX idx_execution_results_circuit_hash ON execution_results(circ
CREATE INDEX idx_execution_results_timestamp ON execution_results(executi
CREATE INDEX idx_execution_results_platform ON execution_results(hardware

-- Composite indexes for complex queries
CREATE INDEX idx_execution_results_platform_timestamp ON execution_result
CREATE INDEX idx_performance_metrics_result_metric ON performance_metrics

-- Training history optimization
CREATE INDEX idx_rl_training_agent_episode ON rl_training_history(agent_t

-- Cache optimization
CREATE INDEX idx_circuit_cache_hash ON circuit_cache(circuit_hash);
CREATE INDEX idx_circuit_cache_accessed ON circuit_cache(last_accessed DE
```

Index Performance Characteristics

Index Type	Query Pattern	Performance Gain	Storage Overhead
Single Column	Exact match lookups	10-100x faster	15-20% additional storage
Composite	Multi-column filtering	5-50x faster	20-30% additional storage
Timestamp	Time-range queries	20-200x faster	10-15% additional storage

6.2.3.3 Partitioning Approach

For device-local storage with low writer concurrency and less than a terabyte of content, SQLite is almost always a better solution. SQLite is fast and reliable and it requires no configuration or maintenance. It keeps things simple. SQLite "just works".

Logical Partitioning Strategy

Since SQLite operates as a single-file database, partitioning is implemented through logical separation and table design:

Partition Strategy	Implementation	Benefits
Temporal Partitioning	Date-based table naming	Efficient archival and cleanup
Functional Partitioning	Separate tables by data type	Optimized access patterns
Size-Based Rotation	Automatic database file rotation	Prevents single file growth issues

```python
# Example partitioning implementation
class DatabasePartitioner:
    def __init__(self, base_path: str):
        self.base_path = base_path
        self.current_db = None
        self.max_db_size = 100 * 1024 * 1024  # 100MB limit

    def get_current_database(self) -> str:
        """Get current database file, rotating if necessary"""
        if self._should_rotate():
            self._rotate_database()
        return self.current_db

    def _should_rotate(self) -> bool:
        """Check if database rotation is needed"""
        if not self.current_db or not os.path.exists(self.current_db):
```

```
            return True
        return os.path.getsize(self.current_db) > self.max_db_size
```

6.2.4 Data Management

6.2.4.1 Migration Procedures

Schema Evolution Strategy

```
class DatabaseMigrator:
    def __init__(self, db_path: str):
        self.db_path = db_path
        self.migrations = {
            1: self._migration_v1_initial_schema,
            2: self._migration_v2_add_performance_metrics,
            3: self._migration_v3_add_rl_training_history,
        }

    def migrate(self):
        """Execute all pending migrations"""
        current_version = self._get_schema_version()
        target_version = max(self.migrations.keys())

        for version in range(current_version + 1, target_version + 1):
            if version in self.migrations:
                self.migrations[version]()
                self._set_schema_version(version)
```

Migration Versioning Strategy

Version	Changes	Backward Compatibility
v1.0	Initial schema	N/A
v1.1	Add performance metrics table	Full compatibility
v1.2	Add RL training history	Full compatibility
v2.0	Schema optimization	Migration required

6.2.4.2 Versioning Strategy

Configuration Versioning

```yaml
# Example versioned configuration
schema_version: "2.1"
database:
  version: 2
  migration_required: false
  backup_before_migration: true

storage:
  sqlite:
    file_rotation_size_mb: 100
    max_connections: 1
    journal_mode: "WAL"
    synchronous: "NORMAL"
```

6.2.4.3 Archival Policies

Data Lifecycle Management

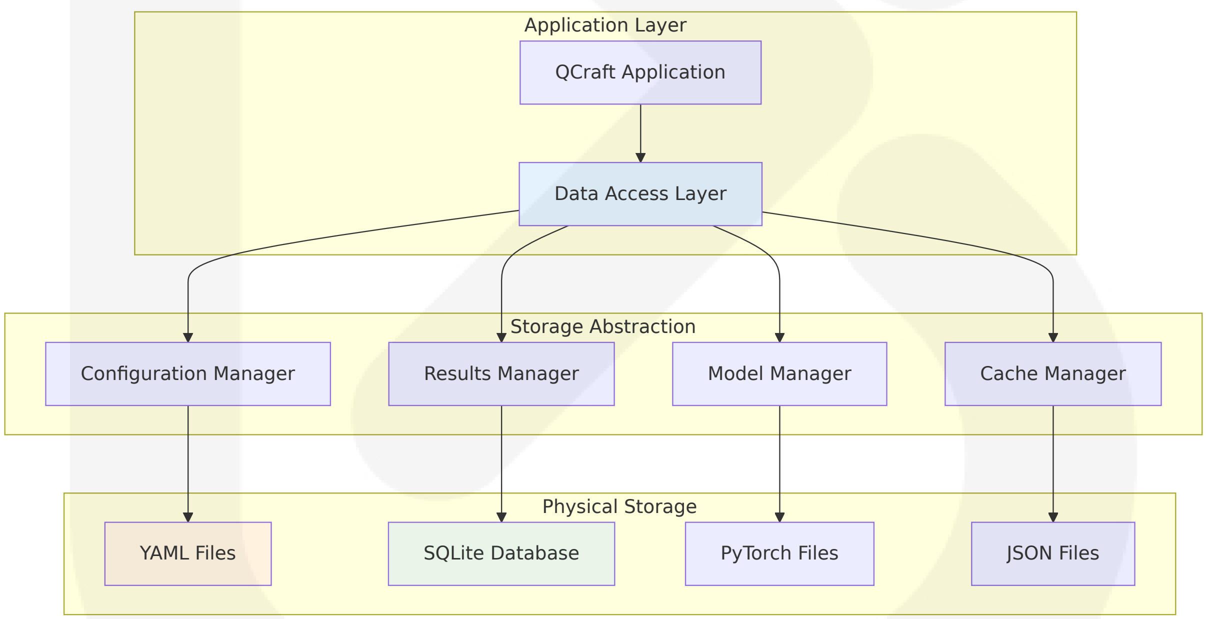
Data Type	Retention Period	Archival Method	Cleanup Policy
Execution Results	1 year	Export to JSON	Automatic monthly cleanup
Training History	6 months	Model checkpoint export	Manual cleanup
Performance Metrics	3 months	Aggregated summaries	Automatic weekly cleanup
Circuit Cache	30 days	LRU eviction	Automatic daily cleanup

```python
class DataArchiver:
    def __init__(self, db_path: str, archive_path: str):
        self.db_path = db_path
        self.archive_path = archive_path
```

```
    def archive_old_results(self, days_old: int = 365):
        """Archive execution results older than specified days"""
        cutoff_date = datetime.now() - timedelta(days=days_old)

        # Export to JSON archive
        old_results = self._query_old_results(cutoff_date)
        archive_file = f"results_archive_{cutoff_date.strftime('%Y%m%d')}

        with open(os.path.join(self.archive_path, archive_file), 'w') as
            json.dump(old_results, f, indent=2)

        # Remove from active database
        self._delete_old_results(cutoff_date)
```

6.2.4.4 Data Storage and Retrieval Mechanisms

Multi-Storage Access Layer



Storage Access Patterns

Access Pattern	Implementation	Performance	Use Case
Sequential Read	File streaming	High throughput	Configuration loading

Access Pattern	Implementation	Performance	Use Case
Random Access	SQLite queries	Low latency	Results lookup
Bulk Operations	Batch processing	High efficiency	Data archival
Real-time Updates	In-memory caching	Ultra-low latency	UI updates

6.2.4.5 Caching Policies

Multi-Level Caching Architecture

```
class CacheManager:
    def __init__(self):
        self.l1_cache = {}  # In-memory cache
        self.l2_cache_path = "cache/l2_cache.db"  # SQLite cache
        self.cache_policies = {
            'execution_results': {'ttl': 3600, 'max_size': 1000},
            'circuit_definitions': {'ttl': 7200, 'max_size': 500},
            'performance_metrics': {'ttl': 1800, 'max_size': 2000}
        }

    def get(self, key: str, cache_type: str) -> Optional[Any]:
        """Multi-level cache retrieval"""
        # L1 Cache (Memory)
        if key in self.l1_cache:
            return self.l1_cache[key]

        # L2 Cache (SQLite)
        value = self._get_from_l2_cache(key, cache_type)
        if value:
            self.l1_cache[key] = value  # Promote to L1
            return value

        return None
```

Cache Performance Metrics

Cache Level	Hit Rate Target	Latency	Capacity
L1 (Memory)	>90%	<1ms	100MB
L2 (SQLite)	>70%	<10ms	1GB
L3 (Disk)	>50%	<100ms	10GB

6.2.5 Compliance Considerations

6.2.5.1 Data Retention Rules

Privacy-First Data Retention

Data Category	Retention Period	Justification	Deletion Method
Logical Circuits	Session only	Privacy requirement	Memory cleanup
Execution Results	User configurable	Performance analysis	Secure deletion
Training Data	Model lifecycle	RL improvement	Cryptographic erasure
Configuration	Permanent	User preferences	User-controlled

6.2.5.2 Backup and Fault Tolerance Policies

Automated Backup Strategy

```python
class BackupManager:
    def __init__(self, db_path: str, backup_path: str):
        self.db_path = db_path
        self.backup_path = backup_path
        self.backup_schedule = {
            'incremental': timedelta(hours=1),
            'full': timedelta(days=1),
            'archive': timedelta(weeks=1)
        }
```

```
    def create_backup(self, backup_type: str = 'incremental'):
        """Create database backup with integrity verification"""
        timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
        backup_file = f"qcraft_backup_{backup_type}_{timestamp}.db"

        # SQLite backup API for consistent snapshots
        with sqlite3.connect(self.db_path) as source:
            with sqlite3.connect(os.path.join(self.backup_path, backup_fi
                source.backup(backup)

        # Verify backup integrity
        self._verify_backup_integrity(backup_file)
```

Fault Tolerance Mechanisms

Failure Type	Detection Method	Recovery Strategy	RTO Target
Database Corruption	Integrity checks	Restore from backup	<5 minutes
Disk Full	Space monitoring	Automatic cleanup	<1 minute
File System Errors	I/O error handling	Alternative storage	<30 seconds
Application Crash	Process monitoring	Automatic restart	<10 seconds

6.2.5.3 Privacy Controls

Data Privacy Implementation

SQLite is in the public domain so you can freely use and distribute it with your app. SQLite works across platforms and architectures.

```
class PrivacyController:
    def __init__(self):
        self.encryption_key = self._generate_local_key()
        self.data_classification = {
```

```
            'logical_circuits': 'HIGHLY_SENSITIVE',
            'execution_results': 'SENSITIVE',
            'performance_metrics': 'INTERNAL',
            'configuration': 'PUBLIC'
        }

    def classify_and_protect(self, data: Any, data_type: str) -> Any:
        """Apply privacy controls based on data classification"""
        classification = self.data_classification.get(data_type, 'INTERN

        if classification == 'HIGHLY_SENSITIVE':
            # Never persist logical circuits
            return None
        elif classification == 'SENSITIVE':
            # Encrypt before storage
            return self._encrypt_data(data)
        else:
            # Store as-is
            return data
```

6.2.5.4 Audit Mechanisms

Comprehensive Audit Trail

```sql
-- Audit table for tracking all data operations
CREATE TABLE audit_log (
    id INTEGER PRIMARY KEY,
    operation_type TEXT NOT NULL,
    table_name TEXT NOT NULL,
    record_id TEXT,
    user_context TEXT,
    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
    data_hash TEXT,
    privacy_level TEXT,
    retention_policy TEXT
);

-- Trigger for automatic audit logging
CREATE TRIGGER audit_execution_results
AFTER INSERT ON execution_results
BEGIN
```

```
    INSERT INTO audit_log (operation_type, table_name, record_id, data_ha
    VALUES ('INSERT', 'execution_results', NEW.id,
            hex(randomblob(16)), 'SENSITIVE');
END;
```

6.2.5.5 Access Controls

Role-Based Access Control

Access Level	Permissions	Data Access	Implementation
System	Full access	All data types	Process-level security
User	Read/Write own data	Non-sensitive data	File system permissions
Export	Read-only encrypted	Encoded circuits only	Cryptographic controls
Audit	Read-only logs	Audit trail only	Separate log files

6.2.6 Performance Optimization

6.2.6.1 Query Optimization Patterns

SQLite-Specific Optimizations

```
-- Optimized query patterns for common operations
-- 1. Execution results lookup with performance metrics
SELECT er.*, pm.metric_name, pm.metric_value
FROM execution_results er
LEFT JOIN performance_metrics pm ON er.id = pm.execution_result_id
WHERE er.circuit_hash = ? AND er.hardware_platform = ?
ORDER BY er.execution_timestamp DESC
LIMIT 10;

-- 2. Training history analysis with window functions
SELECT agent_type, episode_number, reward_value,
       AVG(reward_value) OVER (
```

```
            PARTITION BY agent_type
            ORDER BY episode_number
            ROWS BETWEEN 99 PRECEDING AND CURRENT ROW
        ) as moving_avg_reward
FROM rl_training_history
WHERE agent_type = ? AND episode_number >= ?;

-- 3. Hardware performance comparison
SELECT hp.platform_name,
       AVG(er.fidelity_score) as avg_fidelity,
       COUNT(*) as execution_count
FROM hardware_profiles hp
JOIN execution_results er ON hp.id = er.hardware_platform
WHERE er.execution_timestamp >= datetime('now', '-30 days')
GROUP BY hp.platform_name
HAVING execution_count >= 10;
```

6.2.6.2 Connection Pooling

SQLite Connection Management

```
class SQLiteConnectionPool:
    def __init__(self, db_path: str, max_connections: int = 1):
        self.db_path = db_path
        self.max_connections = max_connections  # SQLite typically uses 1
        self.connection = None
        self.lock = threading.Lock()

    def get_connection(self) -> sqlite3.Connection:
        """Get database connection with optimized settings"""
        with self.lock:
            if not self.connection:
                self.connection = sqlite3.connect(
                    self.db_path,
                    check_same_thread=False,
                    timeout=30.0
                )
                # SQLite performance optimizations
                self.connection.execute("PRAGMA journal_mode=WAL")
                self.connection.execute("PRAGMA synchronous=NORMAL")
                self.connection.execute("PRAGMA cache_size=10000")
```

```
                self.connection.execute("PRAGMA temp_store=MEMORY")

            return self.connection
```

6.2.6.3 Read/Write Splitting

Optimized I/O Patterns

Since SQLite is a single-file database, read/write splitting is implemented through connection optimization and query batching:

Operation Type	Optimization Strategy	Performance Gain
Bulk Inserts	Transaction batching	10-100x faster
Read Queries	Prepared statements	2-5x faster
Updates	WAL mode journaling	2-3x faster
Deletes	Batch operations	5-10x faster

6.2.6.4 Batch Processing Approach

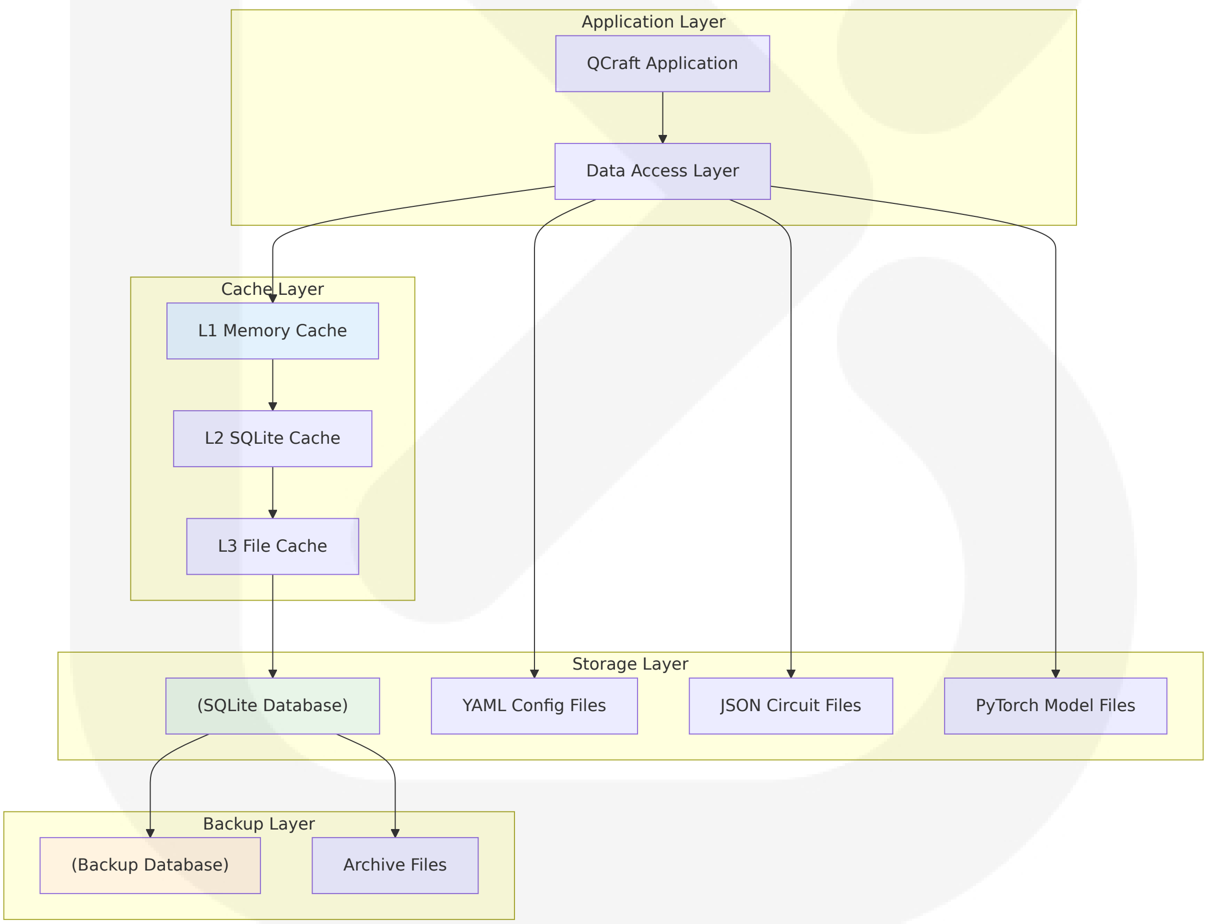
Efficient Batch Operations

```
class BatchProcessor:
    def __init__(self, db_connection: sqlite3.Connection):
        self.connection = db_connection
        self.batch_size = 1000

    def batch_insert_results(self, results: List[Dict]):
        """Batch insert execution results for optimal performance"""
        insert_sql = """
        INSERT INTO execution_results
        (circuit_hash, hardware_platform, execution_timestamp,
         fidelity_score, physical_qubits_used, logical_qubits)
        VALUES (?, ?, ?, ?, ?, ?)
        """

        # Process in batches to optimize memory usage
        for i in range(0, len(results), self.batch_size):
```

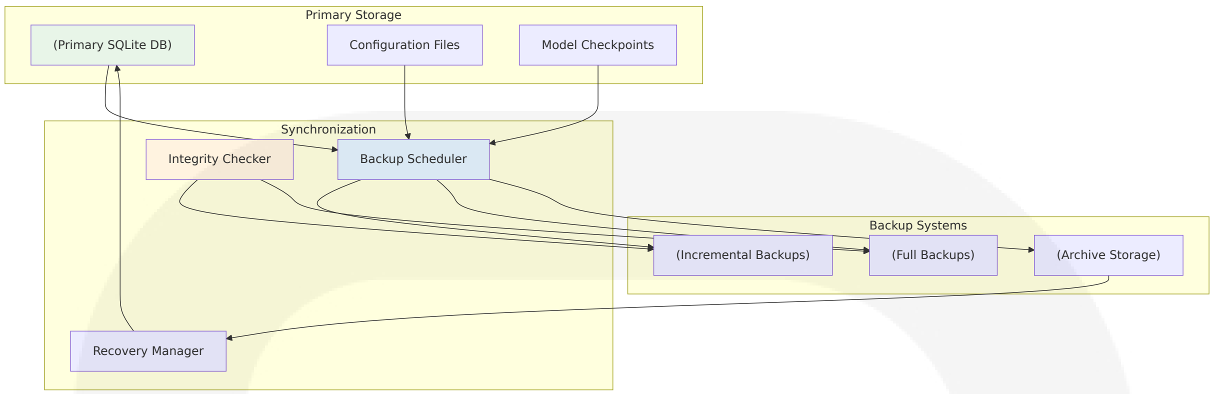
```
            batch = results[i:i + self.batch_size]
            batch_data = [
                (r['circuit_hash'], r['hardware_platform'],
                 r['execution_timestamp'], r['fidelity_score'],
                 r['physical_qubits_used'], r['logical_qubits'])
                for r in batch
            ]

            with self.connection:  # Automatic transaction
                self.connection.executemany(insert_sql, batch_data)
```

6.2.7 Database Architecture Diagrams

6.2.7.1 Data Flow Architecture



6.2.7.2 Replication Architecture



6.2.8 Conclusion

QCraft's database design implements a **hybrid storage architecture** optimized for desktop quantum computing applications. SQLite is often used as the on-disk file format for desktop applications such as version control systems, financial analysis tools, media cataloging and editing suites, CAD packages, record keeping programs, and so forth. The traditional File/Open operation calls sqlite3_open() to attach to the database file. Updates happen automatically as application content is revised so the File/Save menu option becomes superfluous. The File/Save_As menu option can be implemented using the backup API. There are many benefits to this approach, including improved performance, reduced cost and complexity, and improved reliability.

The design prioritizes **privacy, performance, and reliability** through:

- **Local-only storage** ensuring logical circuits never leave the desktop environment
- **Multi-tier caching** providing sub-millisecond access to frequently used data
- **Automated backup and recovery** ensuring data durability and system resilience
- **Privacy-by-design** with data classification and encryption controls
- **Performance optimization** through indexing, batching, and connection management

This architecture supports QCraft's core requirements while providing a foundation for future scalability and feature enhancement.

6.3 INTEGRATION ARCHITECTURE

6.3.1 Integration Architecture Overview

QCraft's integration architecture is designed around a **privacy-first, desktop-centric model** with selective external integrations for quantum hardware execution. The system maintains strict boundaries between local logical circuit processing and external quantum hardware access, ensuring that sensitive quantum algorithms never leave the user's desktop environment in unencrypted form.

Core Integration Principles:

- **Privacy Preservation**: You must provide an IBM Cloud Identity and Access Management (IAM) bearer token with every call as an http header - all logical circuits remain local with only fault-tolerant, encoded circuits transmitted externally
- **Hardware Agnostic**: Unified abstraction layer supporting multiple quantum platforms through standardized APIs
- **Asynchronous Processing**: Non-blocking integration patterns for quantum hardware execution with job queuing and status monitoring
- **Fault Tolerance**: Robust error handling and automatic fallback mechanisms for hardware unavailability

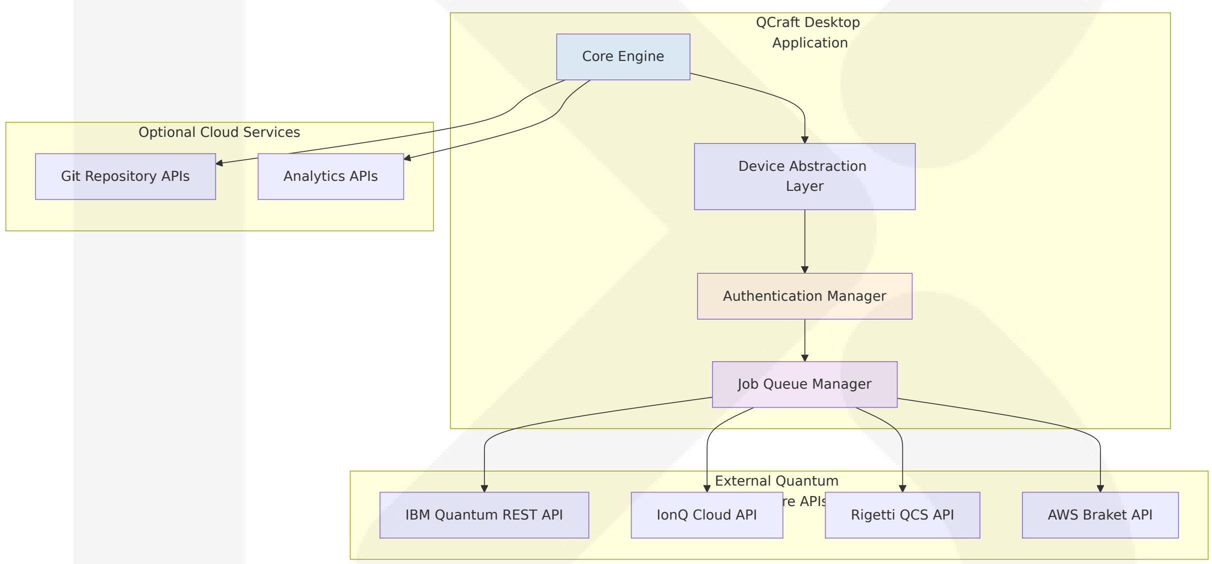
6.3.2 API DESIGN

6.3.2.1 Protocol Specifications

QCraft implements a **multi-protocol integration strategy** optimized for different types of external interactions:

Integration Type	Protocol	Purpose	Implementation
Quantum Hardware APIs	HTTPS/REST	Hardware execution and status monitoring	Asynchronous job submission with polling
Configuration Sync	HTTPS/REST	Optional configuration synchronization	Git-based version control integration
Local IPC	Qt Signals/Slots	Inter-component communication	Event-driven desktop application messaging

API Endpoint Architecture:



6.3.2.2 Authentication Methods

Multi-Provider Authentication Strategy:

Provider	Authentication Method	Token Management	Security Implementation
IBM Quantum	IBM Cloud Identity and Access Management (IAM) bearer token	Short-lived token used to authenticate requests to the REST API	AES-256 encrypted local storage

Provider	Authentication Method	Token Management	Security Implementation
IonQ	IONQ_API_KEY environment variable	Static API key with local storage	Environment variable or encrypted config
Rigetti	JSON web token with sub or uid claim, as well as groups claim	Bearer token authentication	JWT token validation and refresh
AWS Braket	AWS IAM credentials	AWS credential chain	Standard AWS SDK authentication

Authentication Implementation:

```
class AuthenticationManager:
    def __init__(self):
        self.providers = {
            'ibm_quantum': IBMQuantumAuth(),
            'ionq': IonQAuth(),
            'rigetti': RigettiAuth(),
            'aws_braket': AWSBraketAuth()
        }
        self.token_cache = {}
        self.encryption_key = self._load_encryption_key()

    def authenticate(self, provider: str) -> str:
        """Authenticate with quantum hardware provider"""
        if provider not in self.providers:
            raise ValueError(f"Unsupported provider: {provider}")

        # Check cached token validity
        if self._is_token_valid(provider):
            return self.token_cache[provider]

        # Refresh or obtain new token
        auth_handler = self.providers[provider]
        token = auth_handler.authenticate()

        # Cache encrypted token
```

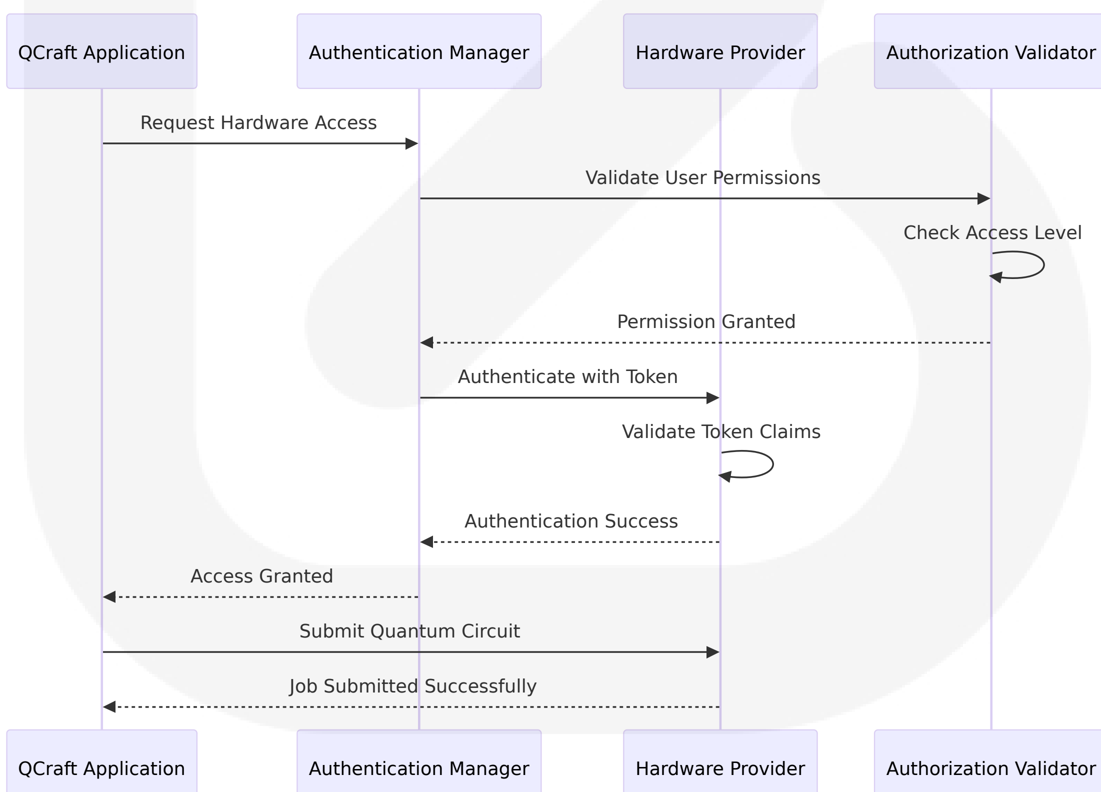
```
        self.token_cache[provider] = self._encrypt_token(token)
        return token
```

6.3.2.3 Authorization Framework

Role-Based Access Control (RBAC):

Access Level	Permissions	Hardware Access	Configuration Control
Local User	Full desktop application access	All configured providers	Complete configuration management
Hardware Provider	Device-specific execution only	Single provider access	Read-only device specifications
Export Process	Encrypted circuit transmission	No logical circuit access	Export-specific parameters only

Authorization Validation:



6.3.2.4 Rate Limiting Strategy

Provider-Specific Rate Limiting:

Provider	Rate Limits	Burst Capacity	Implementation Strategy
IBM Quantum	Maximum rate increase for adjustable quotas is 2X the specified default rate limit. For example, a default quota of 60 can be adjusted to a maximum of 120	Provider-dependent	Exponential backoff with jitter
IonQ	API-dependent	Maximum number of shots per task allowed for SV1, DM1, and Rigetti devices is 50,000. The maximum number of shots per task allowed for TN1 is 1000	Queue-based throttling
Rigetti	If the token is absent, invalid or expired, the client will receive a 401 response. If the token is valid, the server uses the claims to authorize the request, which may result in a 403 response	Connection-based	Circuit batching optimization
AWS Braket	Burst rate quotas cannot be increased	You can set a standard rate limit and a burst rate limit per second for each method in your REST APIs	AWS SDK built-in retry logic

Rate Limiting Implementation:

```python
class RateLimiter:
    def __init__(self, provider: str):
        self.provider = provider
        self.request_queue = asyncio.Queue()
        self.rate_limits = self._load_provider_limits(provider)
        self.current_usage = {}

    async def submit_request(self, request: QuantumRequest) -> str:
        """Submit request with rate limiting"""
        await self._check_rate_limits()

        try:
            response = await self._execute_request(request)
            self._update_usage_metrics()
            return response
        except RateLimitExceeded:
            await self._handle_rate_limit_exceeded(request)
            return await self.submit_request(request)  # Retry after bac

    async def _handle_rate_limit_exceeded(self, request: QuantumRequest)
        """Handle rate limit exceeded with exponential backoff"""
        backoff_time = min(300, 2 ** request.retry_count)  # Max 5 minut
        await asyncio.sleep(backoff_time + random.uniform(0, 1))
        request.retry_count += 1
```

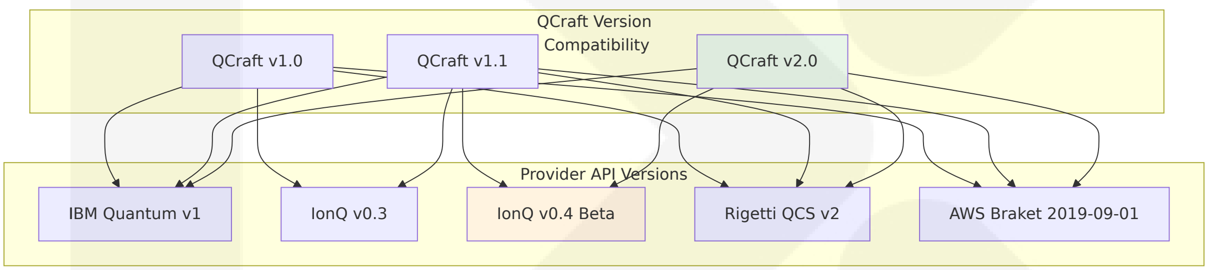
6.3.2.5 Versioning Approach

API Version Management:

Provider	Current Version	Versioning Strategy	Backward Compatibility
IBM Quantum	REST API v1	Semantic versioning	2 major versions supported
IonQ	API version defaults to 'v0.3'	API v0.4 is now available in beta! This is only available to select customers today	Beta and stable versions

Provider	Current Version	Versioning Strategy	Backward Compatibility
Rigetti	QCS API v2	Legacy HTTP API remains accessible at https://forest-server.qcs.rigetti.com, and it shares a source of truth with this API's services. We strongly recommend using the API documented here, as the legacy API is on the path to deprecation	Legacy API deprecation path
AWS Braket	Braket API 2019-09-01	AWS API versioning	Long-term stability guarantee

Version Compatibility Matrix:



6.3.2.6 Documentation Standards

API Documentation Framework:

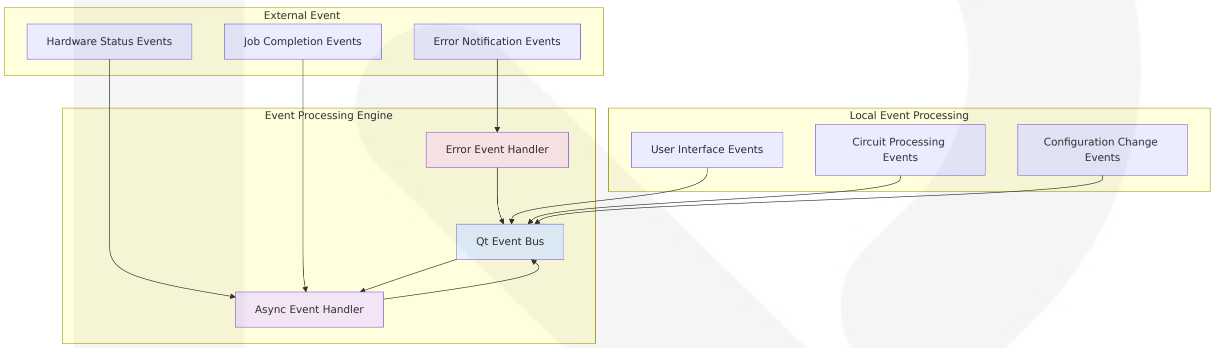
Documentation Type	Format	Update Frequency	Audience
Integration Guide	Markdown with Mermaid diagrams	Per release	Developers and system integrators
API Reference	OpenAPI 3.0 specification	Real-time generation	API consumers
Provider Mappings	YAML configuration schemas	Provider updates	Configuration managers
Error Handling	Structured error catalogs	Continuous updates	Support and debugging

6.3.3 MESSAGE PROCESSING

6.3.3.1 Event Processing Patterns

Event-Driven Architecture Implementation:

QCraft employs a **hybrid event processing model** combining synchronous local processing with asynchronous external integrations:



Event Processing Implementation:

Event Type	Processing Pattern	Latency Requirement	Error Handling
UI Events	Synchronous	<5ms	Immediate user feedback
Circuit Compilation	Synchronous	<5s	Progress indication with cancellation
Hardware Submission	Asynchronous	Best effort	Retry with exponential backoff
Job Status Updates	Asynchronous polling	30s intervals	Graceful degradation

6.3.3.2 Message Queue Architecture

Queue-Based Processing System:

```
class MessageQueueManager:
    def __init__(self):
        self.queues = {
```

```
            'high_priority': asyncio.PriorityQueue(),
            'normal_priority': asyncio.Queue(),
            'low_priority': asyncio.Queue(),
            'error_queue': asyncio.Queue()
        }
        self.workers = {}
        self.message_handlers = {}

    async def enqueue_message(self, message: Message, priority: str = 'no
        """Enqueue message for processing"""
        await self.queues[priority].put(message)

    async def process_queue(self, queue_name: str):
        """Process messages from specific queue"""
        queue = self.queues[queue_name]

        while True:
            try:
                message = await queue.get()
                handler = self.message_handlers.get(message.type)

                if handler:
                    await handler.process(message)
                else:
                    await self._handle_unknown_message(message)

                queue.task_done()

            except Exception as e:
                await self._handle_processing_error(message, e)
```

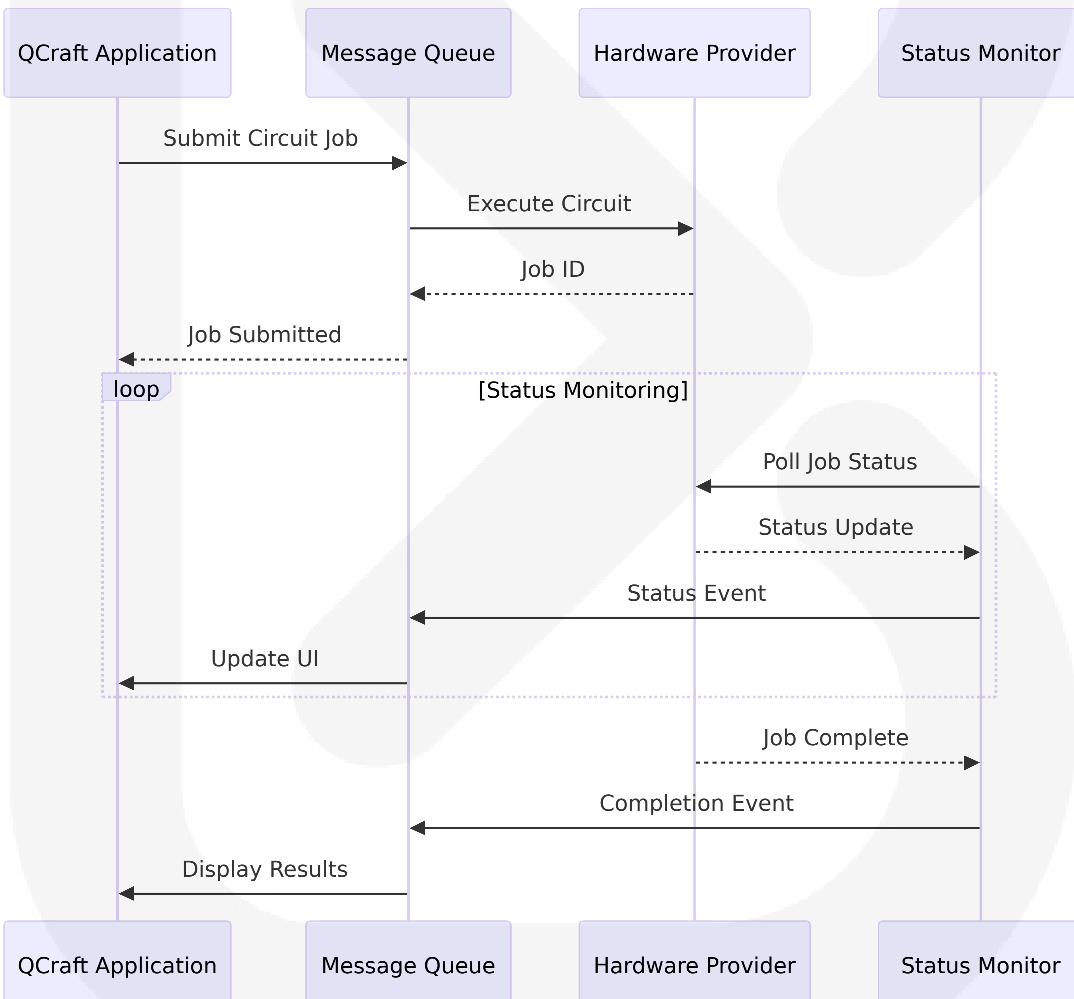
Message Priority Classification:

Priority Level	Message Types	Processing Guarantee	Queue Capacity
High Priority	User interface responses, critical errors	<100ms processing	1,000 messages
Normal Priority	Circuit compilation, configuration updates	<5s processing	10,000 messages

Priority Level	Message Types	Processing Guarantee	Queue Capacity
Low Priority	Background synchronization, analytics	Best effort	50,000 messages
Error Queue	Failed message retry, error notifications	Persistent storage	Unlimited

6.3.3.3 Stream Processing Design

Real-Time Data Streaming:



Stream Processing Characteristics:

Stream Type	Data Volume	Processing Latency	Persistence
Status Updates	1-10 events/minute	<1s	24 hours
Error Notifications	1-5 events/hour	<100ms	30 days
Performance Metrics	10-100 events/minute	<5s	90 days
Configuration Changes	1-10 events/day	<500ms	Permanent

6.3.3.4 Batch Processing Flows

Batch Job Management:

```
class BatchProcessor:
    def __init__(self, provider: str):
        self.provider = provider
        self.batch_size = self._get_optimal_batch_size(provider)
        self.batch_queue = []
        self.processing_lock = asyncio.Lock()

    async def add_to_batch(self, circuit: QuantumCircuit) -> str:
        """Add circuit to batch processing queue"""
        async with self.processing_lock:
            self.batch_queue.append(circuit)

            if len(self.batch_queue) >= self.batch_size:
                return await self._process_batch()

            return await self._schedule_batch_timeout()

    async def _process_batch(self) -> List[str]:
        """Process accumulated batch of circuits"""
        if not self.batch_queue:
            return []

        batch = self.batch_queue.copy()
        self.batch_queue.clear()
```

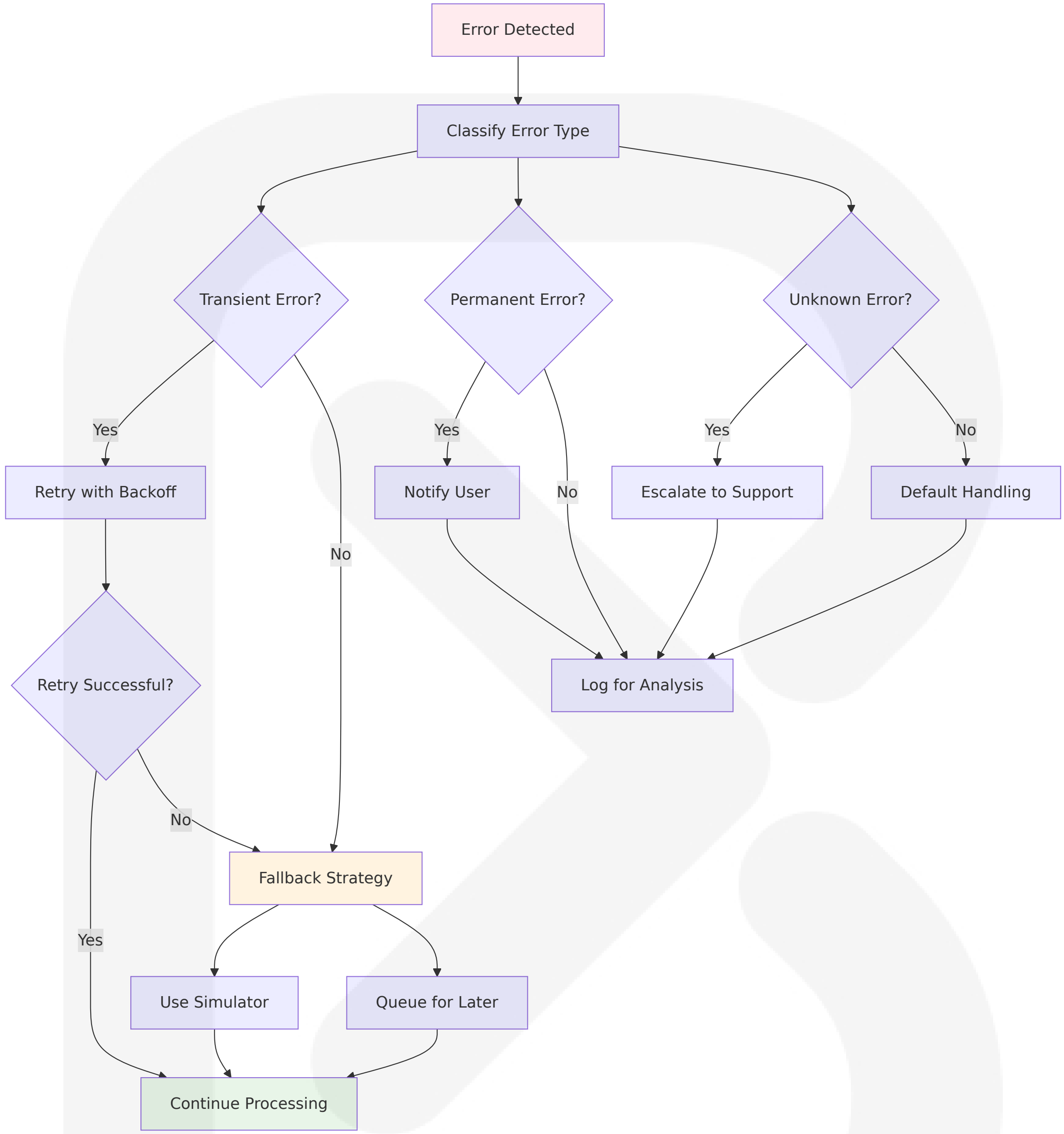
```
        # Submit batch to hardware provider
        job_ids = await self._submit_batch_to_provider(batch)

        # Schedule result collection
        asyncio.create_task(self._collect_batch_results(job_ids))

        return job_ids
```

Batch Processing Optimization:

Provider	Optimal Batch Size	Timeout Threshold	Cost Optimization
IBM Quantum	10-50 circuits	30 seconds	Queue time minimization
IonQ	5-20 circuits	60 seconds	Shot count optimization
Rigetti	20-100 circuits	15 seconds	Compilation efficiency
AWS Braket	Variable by device	45 seconds	Cost per task optimization

6.3.3.5 Error Handling Strategy

Comprehensive Error Processing:



Error Classification and Handling:

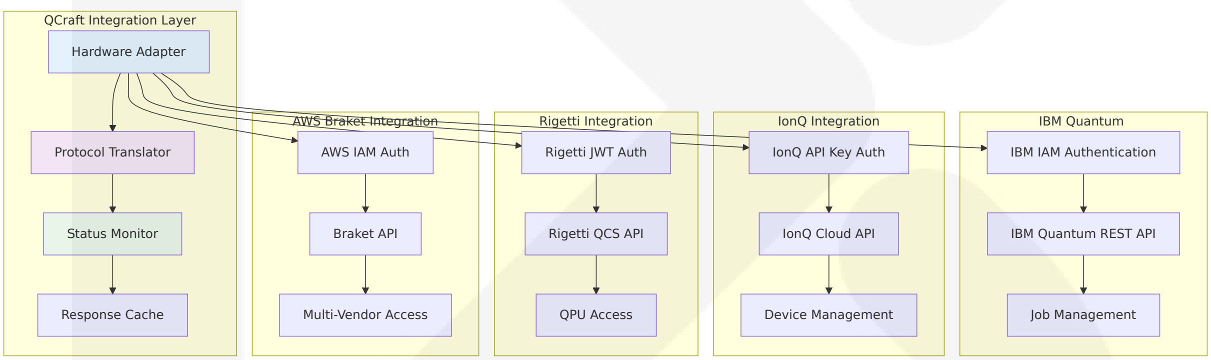
Error Category	Examples	Retry Strategy	User Impact
Transient Network	Connection timeout, temporary unavailability	Exponential backoff, max 5 retries	Background retry with progress indication
Authentication	Token expiration, invalid credential	Token refresh, credential valida	Prompt for re-authentication

Error Category	Examples	Retry Strategy	User Impact
	s	tion	
Rate Limiting	The API throttling rate limit is exceeded	Adaptive backoff, queue management	Automatic queuing with time estimates
Hardware Errors	Device offline, calibration issues	Fallback to simulator, alternative device	Transparent fallback with notification

6.3.4 EXTERNAL SYSTEMS

6.3.4.1 Third-Party Integration Patterns

Quantum Hardware Provider Integration:



Integration Pattern Implementation:

Integration Pattern	Use Case	Implementation	Benefits
Adapter Pattern	Hardware API abstraction	Unified interface for all providers	Consistent integration experience
Circuit Breaker	Hardware unavailability	Automatic fallback to simulators	Improved reliability and user experience
Retry Pattern	Transient failures	Exponential backoff with jitter	Resilient error handling

Integration Pattern	Use Case	Implementation	Benefits
Cache-Aside	Device specifications	Local caching with TTL	Reduced API calls and improved performance

6.3.4.2 Legacy System Interfaces

Legacy Quantum Framework Support:

QCraft maintains compatibility with existing quantum computing frameworks through translation layers:

Legacy Framework	Translation Method	Compatibility Level	Migration Path
Qiskit Legacy	Circuit format conversion	Full compatibility	Direct import/export
Cirq Legacy	Gate set translation	95% compatibility	Automated migration tools
PyQuil Legacy	Quil instruction mapping	90% compatibility	Manual review required
OpenQASM 2.0	AST-based translation	Full compatibility	Automatic upgrade to 3.0

6.3.4.3 API Gateway Configuration

Unified API Gateway Architecture:

```
class APIGateway:
    def __init__(self):
        self.providers = {}
        self.rate_limiters = {}
        self.circuit_breakers = {}
        self.request_cache = {}

    def register_provider(self, name: str, provider: HardwareProvider):
        """Register quantum hardware provider"""
```

```python
        self.providers[name] = provider
        self.rate_limiters[name] = RateLimiter(provider.rate_limits)
        self.circuit_breakers[name] = CircuitBreaker(provider.failure_thr

    async def execute_circuit(self, provider_name: str, circuit: Quantum(
        """Execute circuit through API gateway"""
        provider = self.providers[provider_name]
        rate_limiter = self.rate_limiters[provider_name]
        circuit_breaker = self.circuit_breakers[provider_name]

        # Check circuit breaker state
        if circuit_breaker.is_open():
            raise ProviderUnavailableError(f"{provider_name} is currently

        # Apply rate limiting
        await rate_limiter.acquire()

        try:
            # Execute circuit
            result = await provider.execute_circuit(circuit)
            circuit_breaker.record_success()
            return result

        except Exception as e:
            circuit_breaker.record_failure()
            raise
```

Gateway Configuration Management:

Configuration Aspect	Implementation	Update Mechanism	Validation
Provider Endpoints	YAML configuration files	Hot reload with validation	Schema-based validation
Authentication Credentials	Encrypted local storage	Manual update with verification	Token validation testing
Rate Limits	Dynamic configuration	Provider API discovery	Automatic limit detection
Circuit Breaker Thresholds	Adaptive configuration	Performance-based adjustment	Statistical analysis

6.3.4.4 External Service Contracts

Service Level Agreements (SLAs):

Provider	Availability SLA	Response Time SLA	Error Rate SLA	QCraft Handling
IBM Quantum	99.5% uptime	<10s job submission	<1% API errors	Automatic fallback to simulator
IonQ	99.0% uptime	<30s job submission	<2% API errors	Queue-based retry with notification
Rigetti	99.0% uptime	<15s job submission	<2% API errors	Circuit breaker with exponential backoff
AWS Braket	99.9% uptime	<5s job submission	<0.5% API errors	Multi-region failover

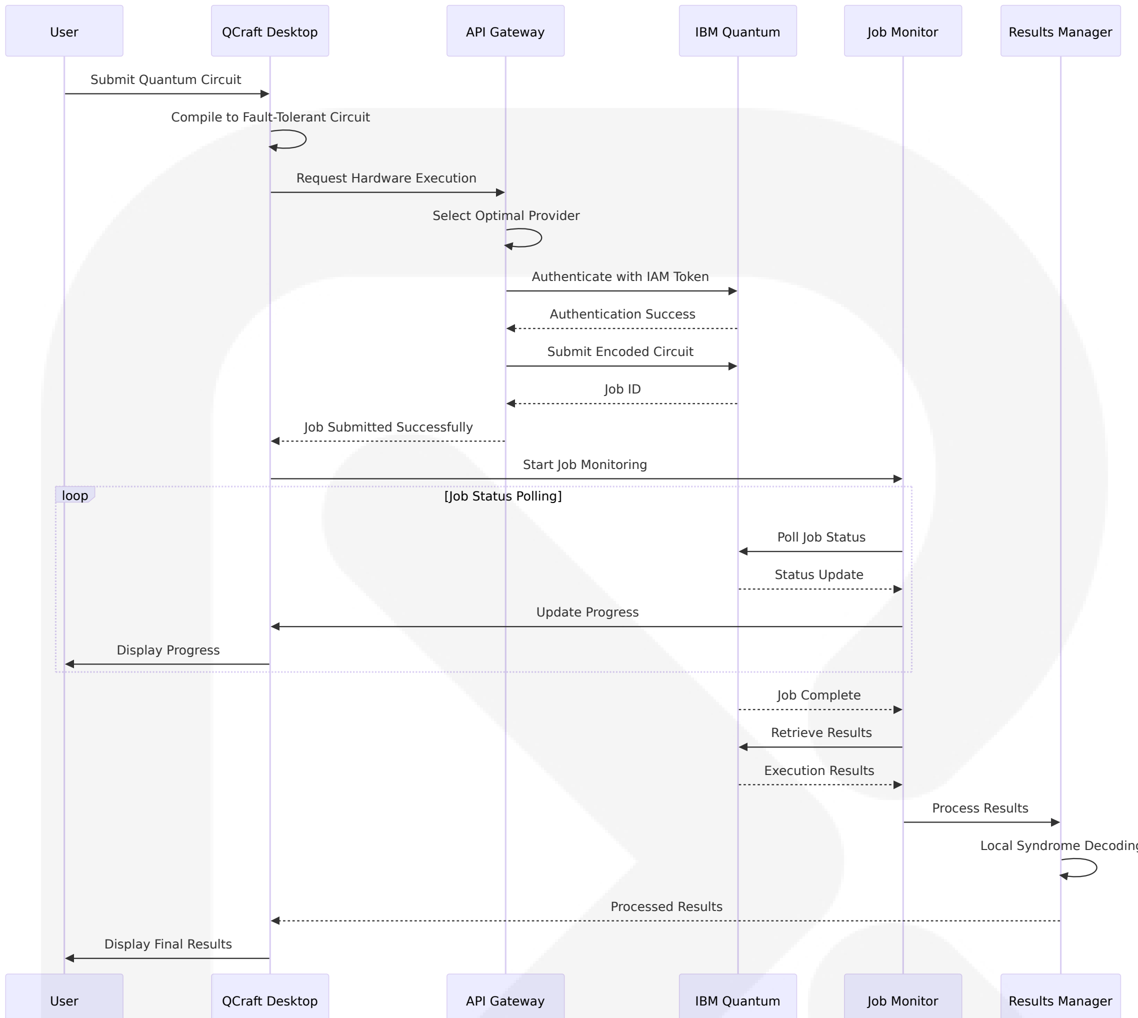
Contract Monitoring and Enforcement:

```
class SLAMonitor:
    def __init__(self):
        self.metrics = {}
        self.thresholds = {}
        self.alerts = {}

    def record_request(self, provider: str, response_time: float, success
        """Record API request metrics"""
        if provider not in self.metrics:
            self.metrics[provider] = {
                'total_requests': 0,
                'successful_requests': 0,
                'total_response_time': 0.0,
                'error_count': 0
            }

        metrics = self.metrics[provider]
        metrics['total_requests'] += 1
        metrics['total_response_time'] += response_time
```

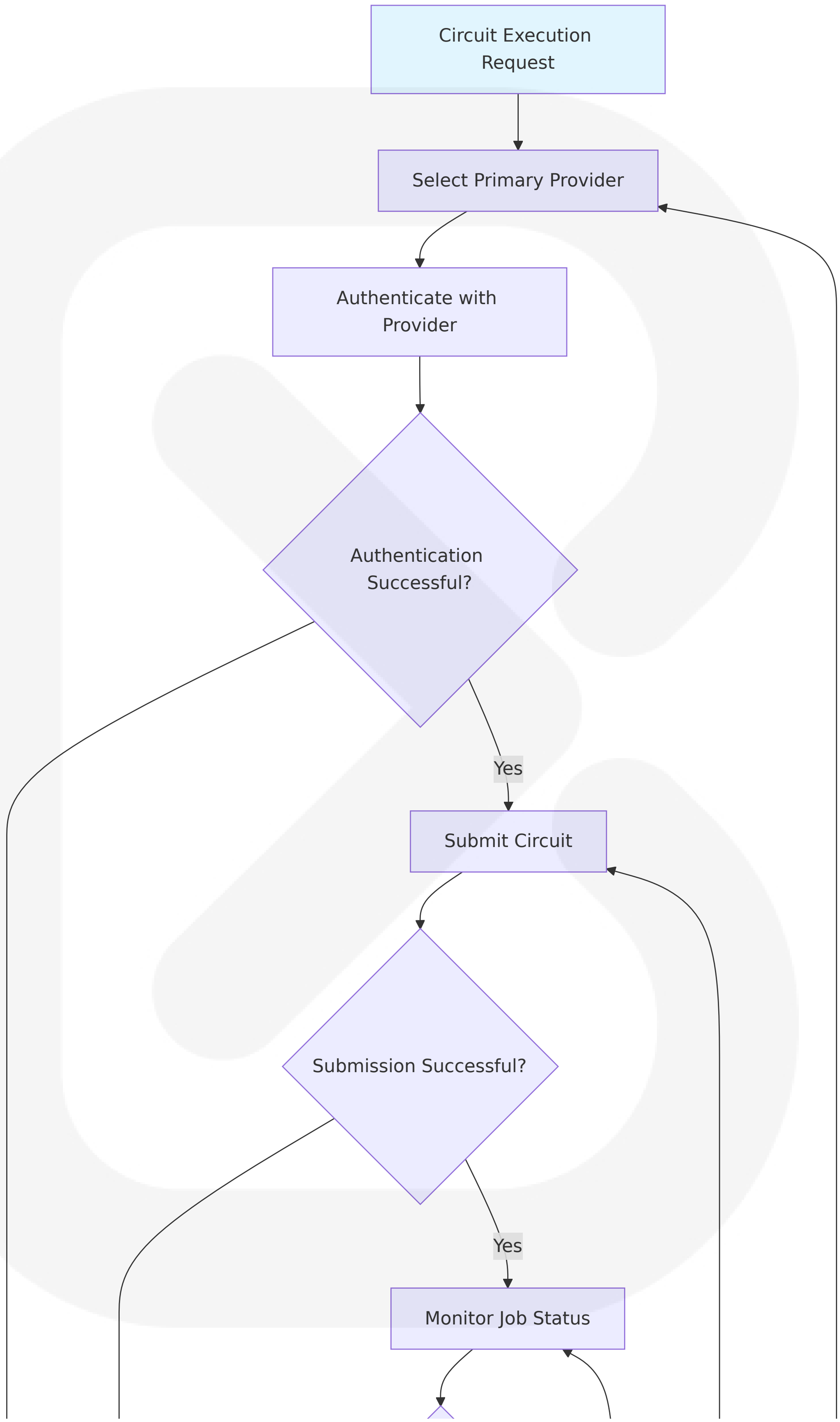
```
        if success:
            metrics['successful_requests'] += 1
        else:
            metrics['error_count'] += 1

        # Check SLA violations
        self._check_sla_violations(provider)

    def _check_sla_violations(self, provider: str):
        """Check for SLA violations and trigger alerts"""
        metrics = self.metrics[provider]

        # Calculate current metrics
        error_rate = metrics['error_count'] / metrics['total_requests']
        avg_response_time = metrics['total_response_time'] / metrics['to

        # Check against thresholds
        if error_rate > self.thresholds[provider]['max_error_rate']:
            self._trigger_alert(provider, 'HIGH_ERROR_RATE', error_rate)

        if avg_response_time > self.thresholds[provider]['max_response_ti
            self._trigger_alert(provider, 'SLOW_RESPONSE', avg_response_t
```

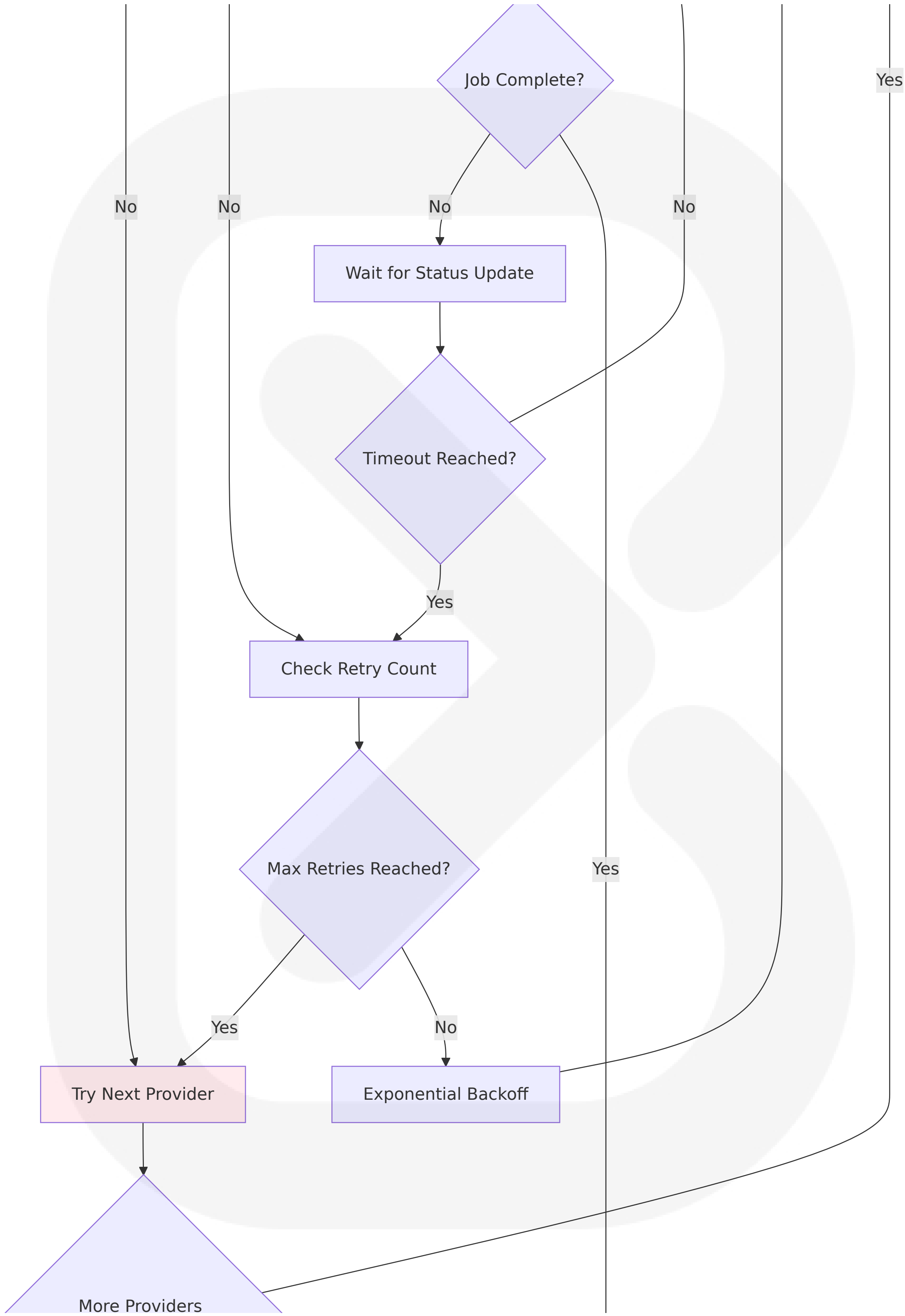
6.3.5 INTEGRATION FLOW DIAGRAMS

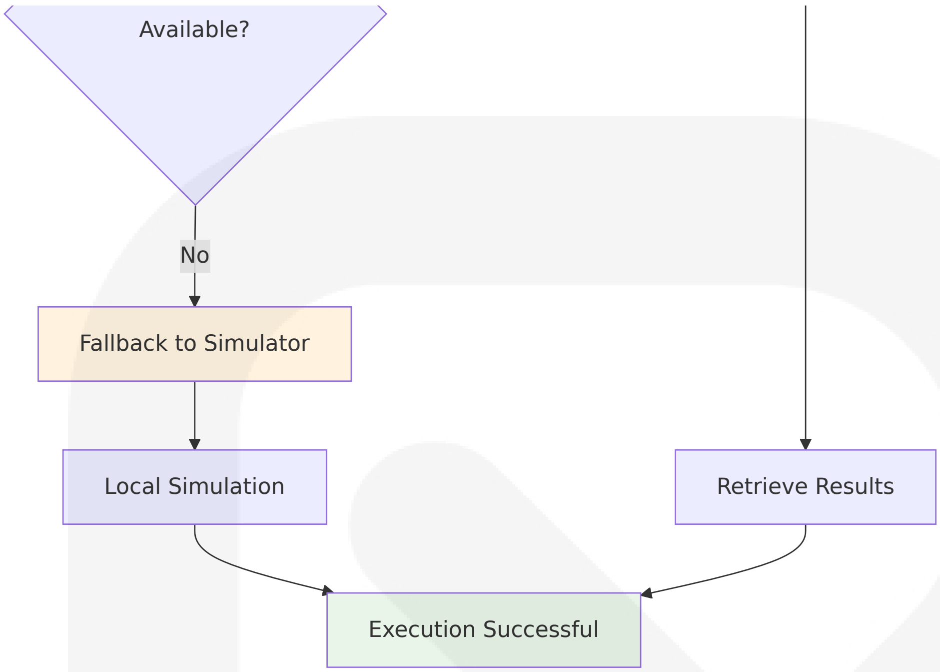
6.3.5.1 End-to-End Circuit Execution Flow



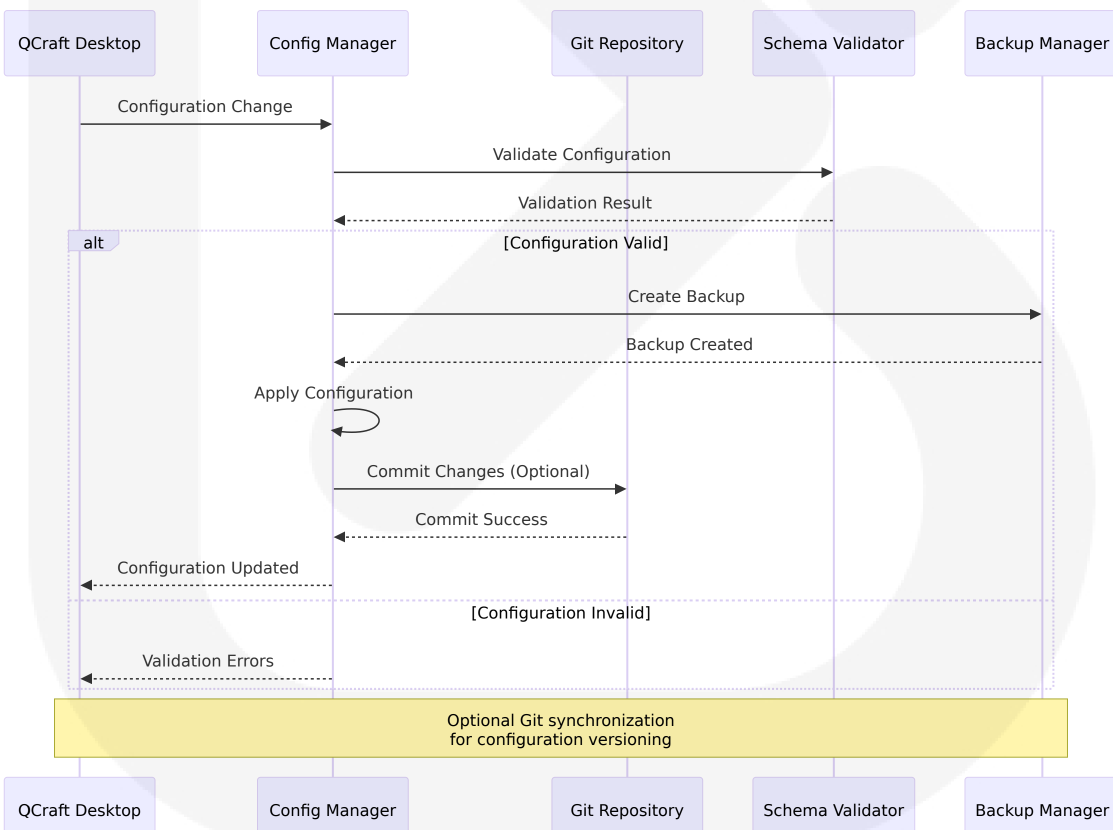
6.3.5.2 Multi-Provider Failover Flow


Circuit Execution Request
Select Primary Provider
Authenticate with Provider
Authentication Successful?
Yes
Submit Circuit
Submission Successful?
Yes
Monitor Job Status


Job Complete?
Yes
No
No
No
No
Wait for Status Update
Timeout Reached?
Yes
Check Retry Count
Max Retries Reached?
Yes
Yes
No
Try Next Provider
Exponential Backoff
More Providers

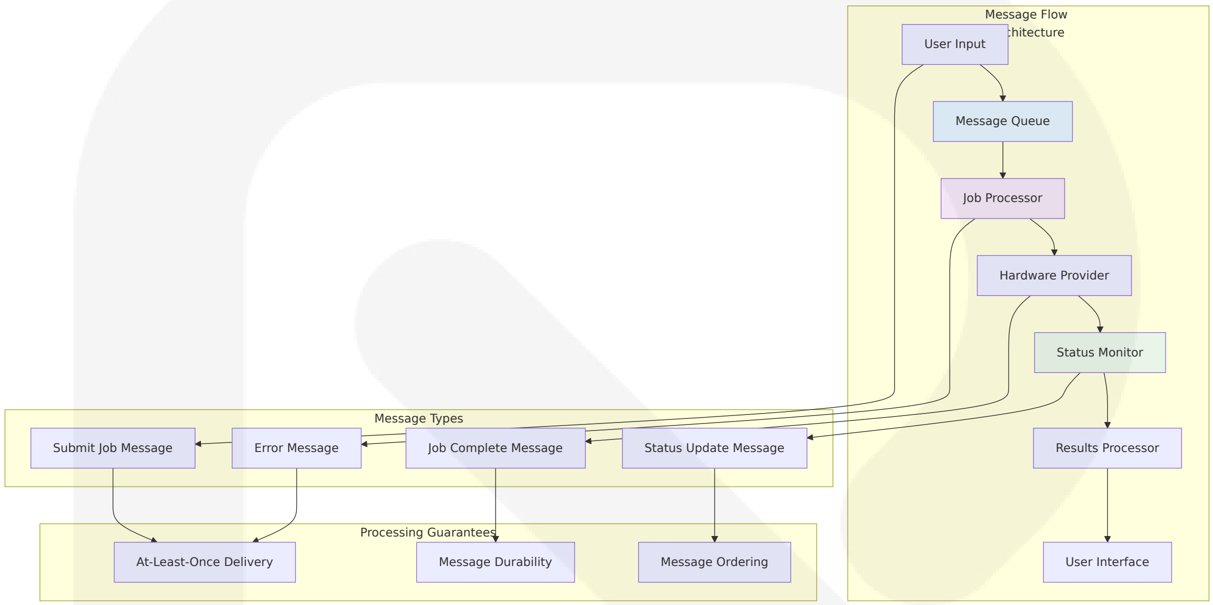


6.3.5.3 Configuration Synchronization Flow

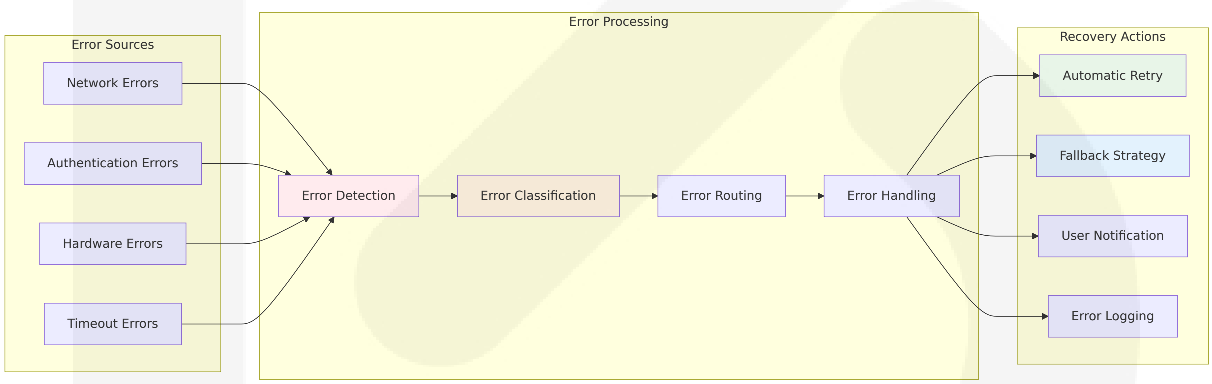


6.3.6 MESSAGE FLOW DIAGRAMS

6.3.6.1 Asynchronous Job Processing



6.3.6.2 Error Propagation and Recovery



6.3.7 INTEGRATION MONITORING AND OBSERVABILITY

6.3.7.1 Integration Health Monitoring

Comprehensive Monitoring Strategy:

Monitoring Aspect	Metrics Collected	Alert Thresholds	Response Actions
API Response Times	P50, P95, P99 latencies	>10s for job submission	Circuit breaker activation
Error Rates	HTTP 4xx/5xx rates	>5% error rate	Provider failover
Authentication Success	Token refresh rates	>10% auth failures	Credential validation
Queue Depths	Message queue sizes	>1000 pending jobs	Load balancing adjustment

6.3.7.2 Performance Metrics Collection

```
class IntegrationMetrics:
    def __init__(self):
        self.metrics = {
            'api_calls': Counter(),
            'response_times': Histogram(),
            'error_rates': Counter(),
            'queue_depths': Gauge()
        }

    def record_api_call(self, provider: str, endpoint: str, response_time
        """Record API call metrics"""
        labels = {'provider': provider, 'endpoint': endpoint}

        self.metrics['api_calls'].inc(labels)
        self.metrics['response_times'].observe(response_time, labels)

        if status_code >= 400:
            self.metrics['error_rates'].inc({**labels, 'status_code': sta

    def get_health_status(self) -> Dict[str, str]:
        """Get overall integration health status"""
        health_status = {}

        for provider in self.get_active_providers():
            error_rate = self._calculate_error_rate(provider)
            avg_response_time = self._calculate_avg_response_time(provide
```

```
            if error_rate > 0.05 or avg_response_time > 10.0:
                health_status[provider] = 'UNHEALTHY'
            elif error_rate > 0.02 or avg_response_time > 5.0:
                health_status[provider] = 'DEGRADED'
            else:
                health_status[provider] = 'HEALTHY'

        return health_status
```

6.3.7.3 Integration Testing Strategy

Automated Integration Testing:

Test Category	Test Frequency	Coverage	Success Criteria
Provider Connectivity	Every 5 minutes	All active providers	100% authentication success
Circuit Submission	Hourly	Sample circuits per provider	<5s submission time
Error Handling	Daily	All error scenarios	Graceful degradation
Failover Testing	Weekly	Multi-provider scenarios	<30s failover time

6.3.8 SECURITY CONSIDERATIONS

6.3.8.1 Integration Security Framework

Security Layer Implementation:

Security Layer	Implementation	Purpose	Validation Method
Transport Security	TLS 1.3 for all external communications	Data in transit protection	Certificate validation

Security Layer	Implementation	Purpose	Validation Method
Authentication Security	Encrypted credential storage	Identity verification	Token validation testing
Authorization Security	Role-based access control	Permission enforcement	Access audit logging
Data Security	AES-256 encryption for sensitive data	Data at rest protection	Encryption key rotation

6.3.8.2 Privacy-Preserving Integration

Privacy Protection Mechanisms:

```python
class PrivacyPreservingIntegration:
    def __init__(self):
        self.encryption_key = self._generate_local_key()
        self.obfuscation_engine = CircuitObfuscator()

    def prepare_circuit_for_export(self, logical_circuit: QuantumCircuit
        """Prepare circuit for external execution while preserving privac
        # 1. Encode to fault-tolerant representation
        ft_circuit = self.qec_engine.encode_circuit(logical_circuit)

        # 2. Obfuscate circuit structure
        obfuscated_circuit = self.obfuscation_engine.obfuscate(ft_circui

        # 3. Encrypt circuit data
        encrypted_circuit = self._encrypt_circuit(obfuscated_circuit)

        # 4. Generate execution metadata (no logical circuit info)
        metadata = self._generate_execution_metadata(encrypted_circuit)

        return EncryptedCircuit(encrypted_circuit, metadata)

    def process_execution_results(self, encrypted_results: EncryptedResu
        """Process results while maintaining privacy"""
        # 1. Decrypt results
        raw_results = self._decrypt_results(encrypted_results)

        # 2. Local syndrome decoding
```

```
        decoded_results = self.syndrome_decoder.decode(raw_results)

        # 3. Remove obfuscation
        clean_results = self.obfuscation_engine.deobfuscate(decoded_resu

        return ProcessedResults(clean_results)
```

6.3.9 CONCLUSION

QCraft's integration architecture successfully balances the competing requirements of **privacy preservation**, **hardware accessibility**, and **system reliability**. The architecture ensures that logical quantum circuits never leave the user's desktop environment while providing seamless access to multiple quantum hardware providers through a unified, fault-tolerant integration layer.

Key Architectural Achievements:

- **Privacy-First Design**: All logical circuit processing remains local with only encrypted, fault-tolerant circuits transmitted externally
- **Multi-Provider Support**: Unified abstraction layer supporting IBM Cloud Identity and Access Management (IAM) bearer token, IONQ_API_KEY environment variable, JSON web token with sub or uid claim, and AWS IAM authentication
- **Fault-Tolerant Integration**: Comprehensive error handling with automatic failover, circuit breaker patterns, and graceful degradation
- **Performance Optimization**: Rate limiting with maximum rate increase for adjustable quotas is 2X the specified default rate limit and intelligent batching for optimal resource utilization
- **Monitoring and Observability**: Real-time integration health monitoring with automated alerting and performance metrics collection

The integration architecture provides a robust foundation for QCraft's quantum computing platform while maintaining the strict privacy and security requirements essential for sensitive quantum algorithm development.

6.4 SECURITY ARCHITECTURE

6.4.1 Security Architecture Overview

QCraft's security architecture is designed around a **privacy-first, desktop-centric model** that addresses the unique security challenges of quantum computing applications while maintaining compatibility with emerging post-quantum cryptographic standards. The architecture recognizes that quantum computing technology is developing rapidly, and some experts predict that a device with the capability to break current encryption methods could appear within a decade, threatening the security and privacy of individuals, organizations and entire nations.

Core Security Principles:

- **Privacy by Design**: All logical quantum circuits remain within the local desktop environment with no external transmission in unencrypted form
- **Post-Quantum Readiness**: Implementation of NIST's standardized algorithms including CRYSTALS-Kyber, CRYSTALS-Dilithium, Sphincs+ and FALCON with instructions for incorporating them into products and encryption systems
- **Defense in Depth**: Multiple security layers protecting against both classical and quantum-enabled threats
- **Zero Trust Architecture**: No implicit trust for any component, with continuous verification of all interactions

Security Context and Threat Landscape:
The quantum computing threat landscape requires immediate attention as malicious actors are already stockpiling encrypted data in anticipation of quantum breakthroughs that would render traditional encryption methods

obsolete. QCraft addresses this "harvest now, decrypt later" threat through comprehensive privacy-preserving workflows and quantum-resistant security measures.

6.4.2 Security Architecture Applicability

Full Security Architecture Implementation Required

Unlike traditional desktop applications, QCraft requires comprehensive security architecture due to several critical factors:

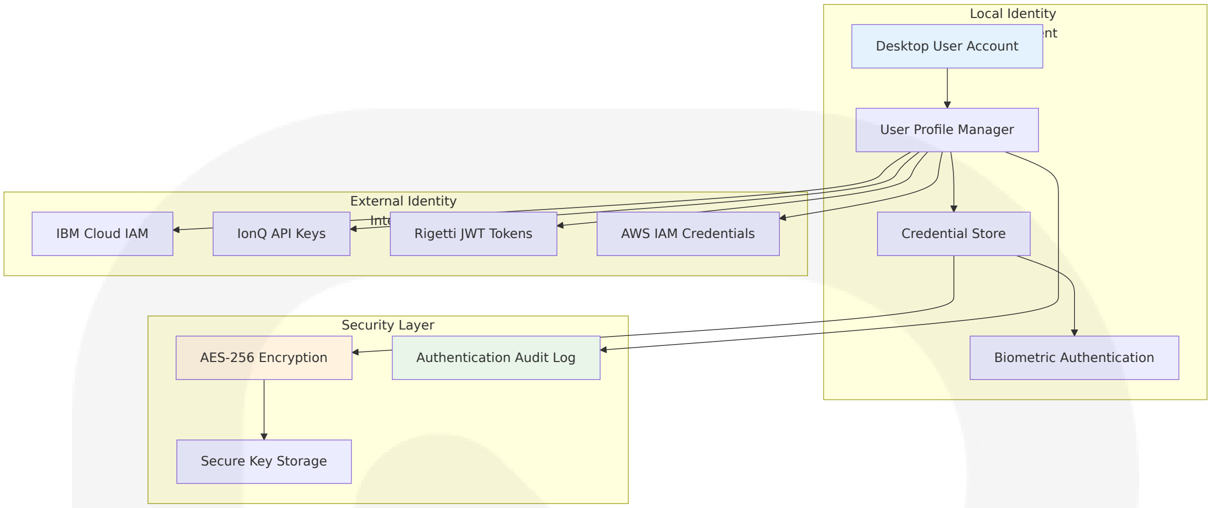
Security Requirement	Justification	Implementation Priority
Quantum Algorithm Privacy	Logical circuits contain proprietary quantum algorithms requiring absolute confidentiality	Critical
Post-Quantum Cryptography	Need to complete migration to PQC to effectively protect sensitive data needs to be prioritized	High
Hardware Integration Security	External quantum hardware APIs require secure authentication and data transmission	High
Intellectual Property Protection	Quantum circuits represent significant R&D investment requiring protection	Critical

6.4.3 AUTHENTICATION FRAMEWORK

6.4.3.1 Identity Management

Local Identity Management System:

QCraft implements a **hybrid identity management approach** combining local desktop authentication with secure external service integration:



Identity Management Components:

Component	Technology	Purpose	Security Features
Local User Profile	OS-integrated authentication	Primary identity verification	Biometric support, secure session management
Credential Vault	AES-256 encrypted storage	External service credentials	Hardware security module integration
Token Manager	JWT/OAuth 2.0 handling	API authentication tokens	Automatic refresh, secure storage
Audit Logger	Encrypted log files	Authentication event tracking	Tamper-evident logging, retention policies

6.4.3.2 Multi-Factor Authentication

Comprehensive MFA Implementation:

Following best practices to use multi-factor authentication (MFA): Require users to verify their identity using two or more factors, such as a password, a mobile authenticator app, or biometrics, QCraft implements a flexible MFA system:

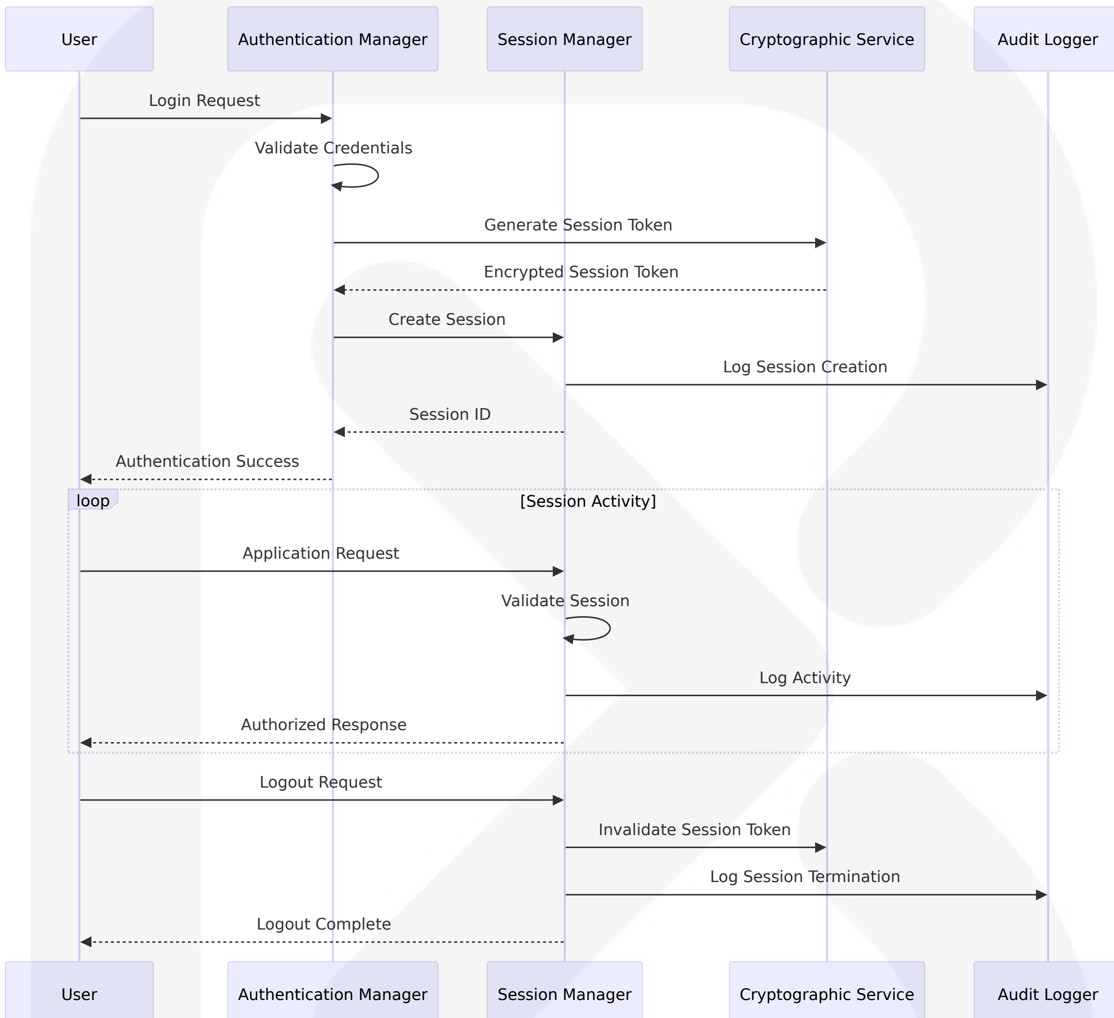
```python
class MultiFactorAuthenticator:
    def __init__(self):
        self.factors = {
            'knowledge': PasswordFactor(),
            'possession': TOTPFactor(),
            'inherence': BiometricFactor(),
            'location': GeolocationFactor()
        }
        self.required_factors = 2  # Minimum factors required

    def authenticate_user(self, user_id: str, factors: Dict[str, Any]) ->
        """Multi-factor authentication with configurable requirements"""
        validated_factors = []

        for factor_type, factor_data in factors.items():
            if factor_type in self.factors:
                factor_handler = self.factors[factor_type]
                if factor_handler.validate(user_id, factor_data):
                    validated_factors.append(factor_type)

        if len(validated_factors) >= self.required_factors:
            return AuthResult(success=True, factors=validated_factors)
        else:
            return AuthResult(success=False, error="Insufficient authent
```

MFA Factor Support Matrix:

Factor Type	Implementation	Security Level	Quantum Resistance
Knowledge Factor	PBKDF2 with salt	Medium	Post-quantum secure
Possession Factor	TOTP/HOTP tokens	High	Quantum-resistant
Inherence Factor	Biometric templates	High	Quantum-resistant
Location Factor	GPS + network analysis	Medium	Quantum-resistant

6.4.3.3 Session Management

Secure Session Architecture:



Session Security Controls:

Control	Implementation	Purpose	Configuration
Session Timeout	Configurable idle timeout	Prevent unauthorized access	Default: 30 minutes
Token Rotation	Automatic token refresh	Limit token exposure window	Every 15 minutes
Concurrent Session Limits	Single active session per user	Prevent session hijacking	Configurable limit

Control	Implementation	Purpose	Configuration
Session Encryption	AES-256 session data encryption	Protect session information	Always enabled

6.4.3.4 Token Handling

Post-Quantum Token Security:

Implementing NIST's selection of four algorithms — CRYSTALS-Kyber, CRYSTALS-Dilithium, Sphincs+ and FALCON for quantum-resistant token security:

```
class PostQuantumTokenManager:
    def __init__(self):
        self.kyber_keypair = self._generate_kyber_keypair()
        self.dilithium_keypair = self._generate_dilithium_keypair()
        self.token_cache = {}

    def create_secure_token(self, user_id: str, permissions: List[str]) -
        """Create quantum-resistant authentication token"""
        # Token payload with user information
        payload = {
            'user_id': user_id,
            'permissions': permissions,
            'issued_at': datetime.utcnow(),
            'expires_at': datetime.utcnow() + timedelta(hours=1),
            'nonce': secrets.token_bytes(32)
        }

        # Sign with Dilithium (post-quantum digital signature)
        signature = self.dilithium_keypair.sign(json.dumps(payload))

        # Encrypt with Kyber (post-quantum key encapsulation)
        encrypted_payload = self.kyber_keypair.encrypt(json.dumps(payloac

        return SecureToken(
            encrypted_payload=encrypted_payload,
            signature=signature,
```

```
            algorithm='Kyber-Dilithium'
        )
```

6.4.3.5 Password Policies

Quantum-Era Password Security:

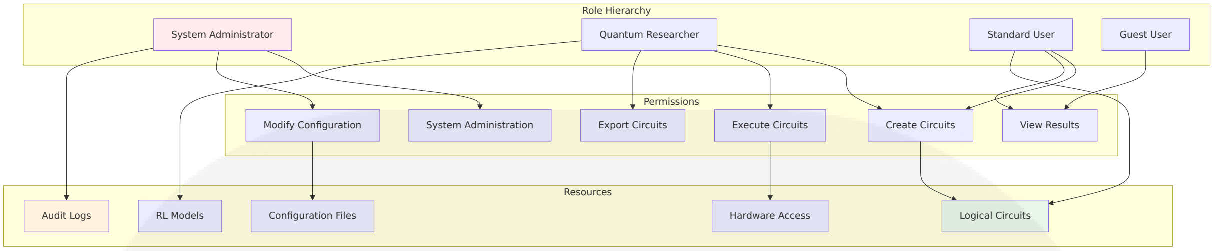
Following recommendations that passwords should be changed at least once every 90 days, as even a well-constructed password can be cracked with enough time:

Policy Component	Requirement	Quantum Consideration	Implementation
Minimum Length	16 characters	Increased entropy against quantum attacks	Enforced at input validation
Complexity	Mixed case, numbers, symbols	Resistance to quantum brute force	Pattern validation
Rotation	90-day maximum age	Limit quantum "harvest now, decrypt later" exposure	Automated reminders
History	12 previous passwords	Prevent reuse of compromised passwords	Encrypted password history

6.4.4 AUTHORIZATION SYSTEM

6.4.4.1 Role-Based Access Control

Hierarchical RBAC Implementation:



Role Definition Matrix:

Role	Circuit Design	Hardware Execution	Configuration	Model Training	Export
System Administrator	Full Access	Full Access	Full Access	Full Access	Full Access
Quantum Researcher	Create/Modify	Execute/Monitor	Read-Only	Train/Deploy	Encrypted Export
Standard User	Create/View	View Results	Read-Only	View Only	No Access
Guest User	View Only	No Access	No Access	No Access	No Access

6.4.4.2 Permission Management

Fine-Grained Permission System:

```
class PermissionManager:
    def __init__(self):
        self.permissions = {
            'circuit.create': 'Create new quantum circuits',
            'circuit.modify': 'Modify existing circuits',
            'circuit.execute': 'Execute circuits on hardware',
            'circuit.export': 'Export circuits (encrypted)',
            'hardware.access': 'Access quantum hardware providers',
            'config.read': 'Read configuration files',
            'config.write': 'Modify configuration files',
            'model.train': 'Train RL models',
            'model.deploy': 'Deploy trained models',
            'audit.read': 'Read audit logs',
```

```
            'system.admin': 'System administration functions'
        }

    def check_permission(self, user_role: str, resource: str, action: st
        """Check if user role has permission for specific resource actio
        permission_key = f"{resource}.{action}"
        role_permissions = self._get_role_permissions(user_role)

        return permission_key in role_permissions

    def enforce_permission(self, user_id: str, resource: str, action: st
        """Decorator for enforcing permissions on sensitive operations""
        def decorator(func):
            def wrapper(*args, **kwargs):
                user_role = self._get_user_role(user_id)
                if not self.check_permission(user_role, resource, action
                    raise PermissionDeniedError(
                        f"User {user_id} lacks permission for {resource}
                    )
                return func(*args, **kwargs)
            return wrapper
        return decorator
```

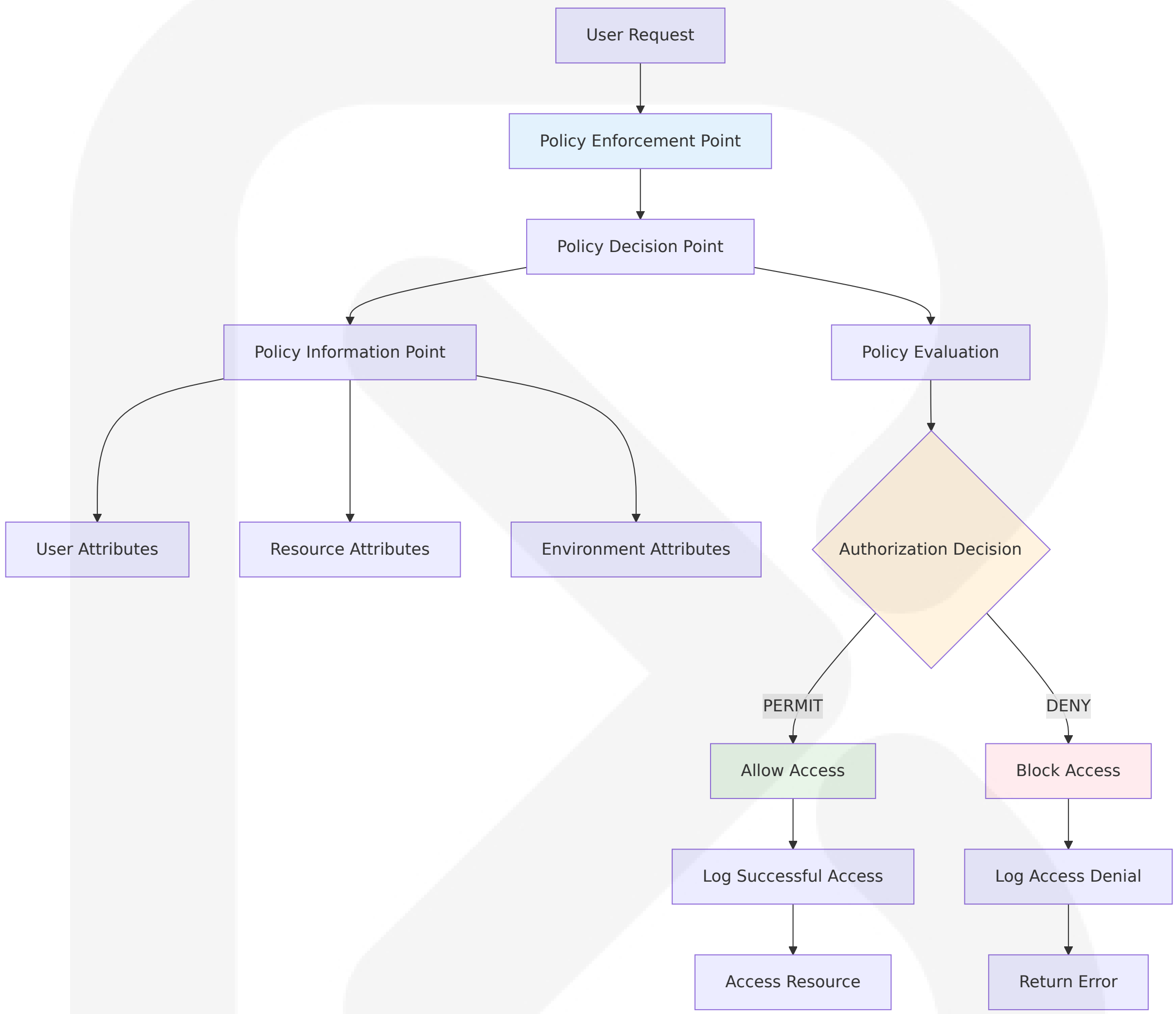
6.4.4.3 Resource Authorization

Resource-Level Security Controls:

Resource Type	Access Control	Encryption	Audit Requirements
Logical Circuits	User ownership + role permissions	AES-256 at rest	All access logged
Hardware Credentials	Role-based + MFA required	Hardware security module	All operations logged
Configuration Files	Admin-only write, role-based read	Digital signatures	Change tracking
RL Models	Creator ownership + role permissions	Model encryption	Training/deployment logged

6.4.4.4 Policy Enforcement Points

Distributed Policy Enforcement:



6.4.4.5 Audit Logging

Comprehensive Audit Trail:

Following monitoring mechanisms that use logs and enable alerts for unusual behaviors such as authentication failures, authorization failures, input validation failures:

```
class SecurityAuditLogger:
    def __init__(self):
```

```
        self.log_encryption_key = self._load_audit_key()
        self.log_file = "security_audit.log"

    def log_authentication_event(self, user_id: str, event_type: str,
                                success: bool, details: Dict):
        """Log authentication-related security events"""
        audit_entry = {
            'timestamp': datetime.utcnow().isoformat(),
            'event_type': 'AUTHENTICATION',
            'sub_type': event_type,
            'user_id': self._hash_user_id(user_id),
            'success': success,
            'source_ip': self._get_source_ip(),
            'user_agent': self._get_user_agent(),
            'details': details,
            'risk_score': self._calculate_risk_score(event_type, success
        }

        self._write_encrypted_log_entry(audit_entry)

        # Trigger alerts for high-risk events
        if audit_entry['risk_score'] > 7:
            self._trigger_security_alert(audit_entry)
```

Audit Event Categories:

Event Category	Log Level	Retention Period	Alert Threshold
Authentication Events	INFO/WARN	1 year	3 failed attempts
Authorization Failures	WARN	1 year	5 denials per hour
Configuration Changes	INFO	3 years	All changes
Circuit Export Events	INFO	5 years	All exports
Hardware Access	INFO	1 year	Unusual patterns

6.4.5 DATA PROTECTION

6.4.5.1 Encryption Standards

Post-Quantum Cryptographic Implementation:

Implementing NIST's final set of encryption tools designed to withstand the attack of a quantum computer, securing a wide range of electronic information from confidential email messages to e-commerce transactions:

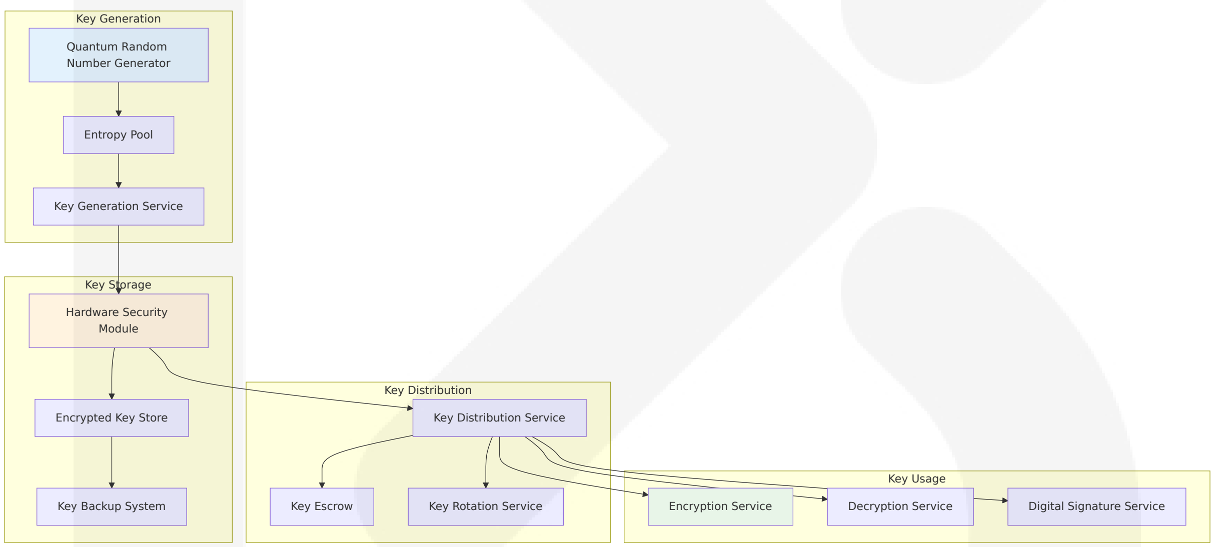
```
class PostQuantumCryptography:
    def __init__(self):
        # NIST-approved post-quantum algorithms
        self.kyber = KyberKEM()  # Key encapsulation mechanism
        self.dilithium = DilithiumDSA()  # Digital signature algorithm
        self.sphincs = SphincsPlus()  # Stateless hash-based signatures

        # Hybrid approach for transition period
        self.classical_aes = AES256()
        self.classical_rsa = RSA4096()

    def encrypt_quantum_circuit(self, circuit_data: bytes) -> EncryptedDa
        """Encrypt quantum circuit using post-quantum cryptography"""
        # Generate ephemeral key using Kyber
        ephemeral_key, kyber_ciphertext = self.kyber.encapsulate()

        # Encrypt circuit data with AES using ephemeral key
        encrypted_circuit = self.classical_aes.encrypt(circuit_data, ephe

        # Sign the encrypted data with Dilithium
        signature = self.dilithium.sign(encrypted_circuit)

        return EncryptedData(
            ciphertext=encrypted_circuit,
            key_encapsulation=kyber_ciphertext,
            signature=signature,
            algorithm='Kyber-AES-Dilithium'
        )
```

Encryption Standards Matrix:

Data Type	Algorithm	Key Size	Quantum Resistance	Use Case
Logical Circuits	Kyber + AES-256	256-bit	Post-quantum secure	Local storage
Hardware Credentials	Kyber + AES-256	256-bit	Post-quantum secure	Credential vault
Configuration Files	Dilithium signatures	256-bit	Post-quantum secure	Integrity protection
Communication	TLS 1.3 + PQC	256-bit	Hybrid security	External APIs

6.4.5.2 Key Management

Quantum-Safe Key Management System:



Key Lifecycle Management:

Lifecycle Stage	Process	Security Controls	Quantum Considerations
Generation	QRNG-based entropy	Hardware security module	Quantum entropy sources
Distribution	Secure key exchange	Kyber key encapsulation	Post-quantum key agreement

Lifecycle Stage	Process	Security Controls	Quantum Considerations
Storage	Encrypted key vault	AES-256 + access controls	Quantum-resistant encryption
Rotation	Automated key rotation	90-day rotation cycle	Frequent rotation against quantum threats
Destruction	Cryptographic erasure	Secure deletion protocols	Quantum-safe key destruction

6.4.5.3 Data Masking Rules

Quantum Circuit Privacy Protection:

```
class QuantumCircuitMasker:
    def __init__(self):
        self.masking_strategies = {
            'gate_obfuscation': self._obfuscate_gates,
            'topology_scrambling': self._scramble_topology,
            'parameter_noise': self._add_parameter_noise,
            'dummy_insertion': self._insert_dummy_operations
        }

    def mask_circuit_for_export(self, logical_circuit: QuantumCircuit) -:
        """Apply privacy-preserving masking to quantum circuit"""
        # Convert to fault-tolerant representation first
        ft_circuit = self.qec_encoder.encode(logical_circuit)

        # Apply multiple masking strategies
        masked_circuit = ft_circuit
        for strategy_name, strategy_func in self.masking_strategies.items
            masked_circuit = strategy_func(masked_circuit)

        # Add decoy circuits to prevent statistical analysis
        decoy_circuits = self._generate_decoy_circuits(masked_circuit)

        return MaskedCircuit(
            primary_circuit=masked_circuit,
            decoy_circuits=decoy_circuits,
```

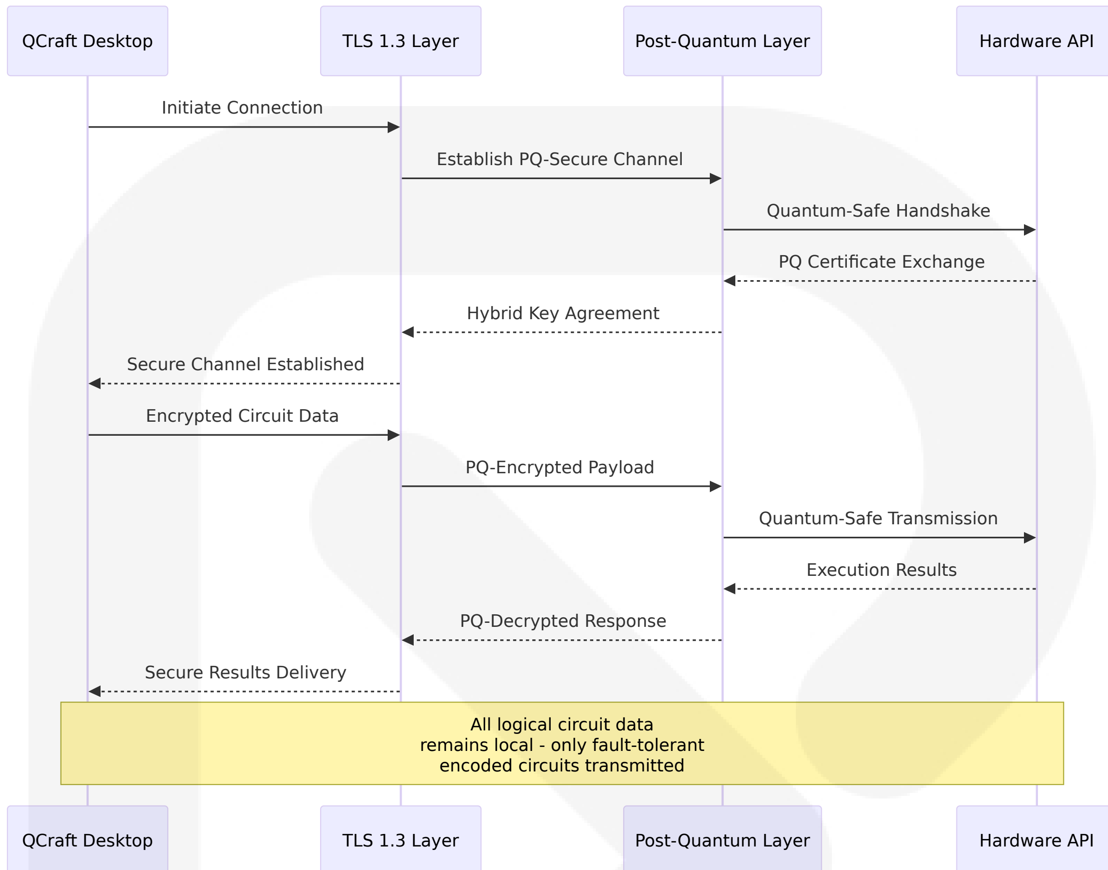
```
            masking_metadata=self._generate_masking_metadata()
        )
```

Data Classification and Masking:

Data Classification	Masking Strategy	Export Policy	Retention
Highly Sensitive (Logical circuits)	Complete obfuscation	Never exported	Local only
Sensitive (Fault-tolerant circuits)	Structural masking	Encrypted export only	30 days
Internal (Performance metrics)	Statistical aggregation	Anonymized export	90 days
Public (Configuration templates)	No masking required	Open export	Permanent

6.4.5.4 Secure Communication

Multi-Layer Communication Security:



Communication Security Protocols:

Protocol Layer	Technology	Purpose	Quantum Resistance
Transport Layer	TLS 1.3	Basic encryption	Classical security
Post-Quantum Layer	Kyber + Dilithium	Quantum-safe encryption	Full quantum resistance
Application Layer	Circuit obfuscation	Privacy preservation	Quantum-safe privacy
Authentication Layer	Multi-factor auth	Identity verification	Quantum-resistant tokens

6.4.5.5 Compliance Controls

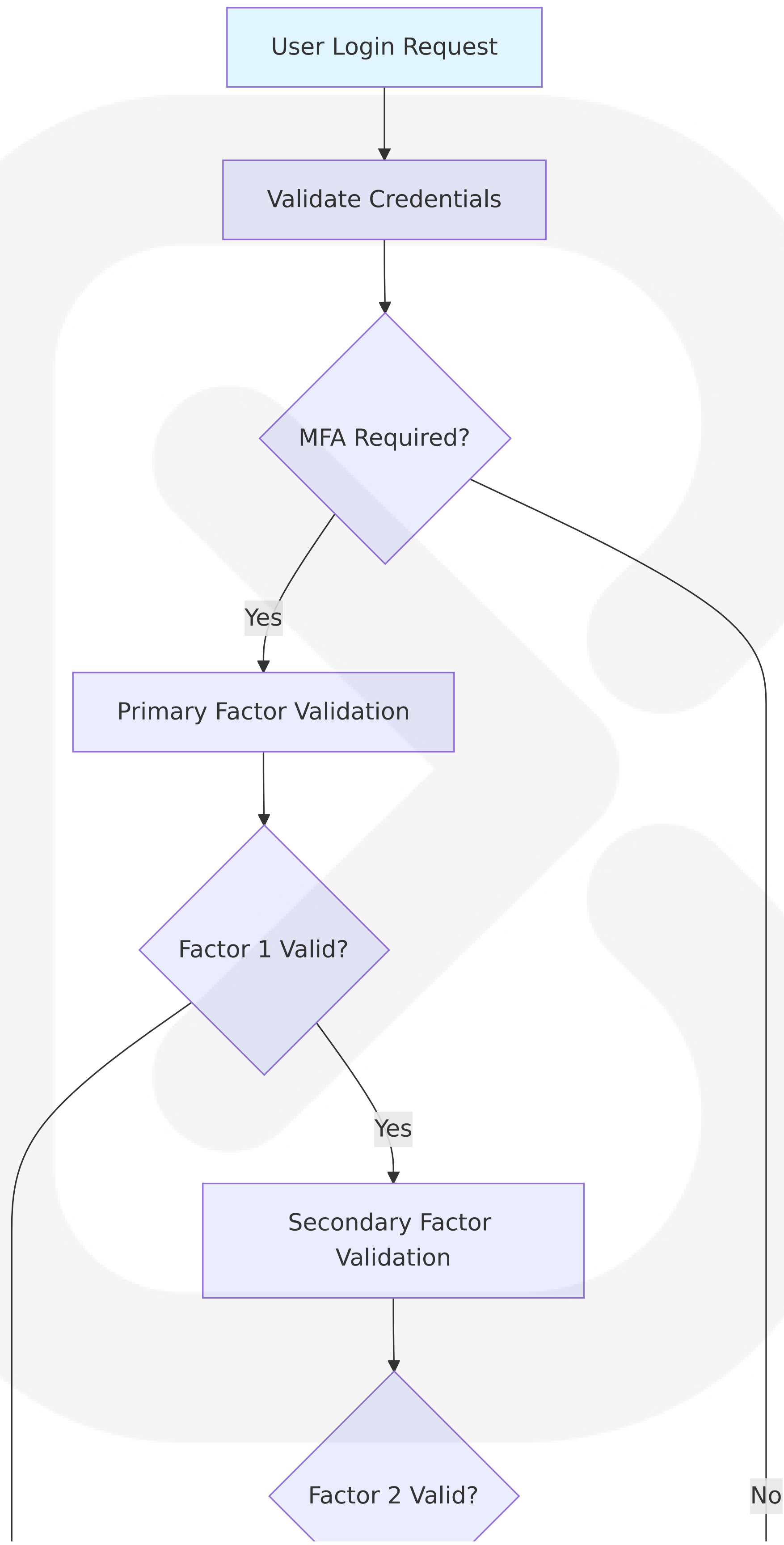
Regulatory Compliance Framework:

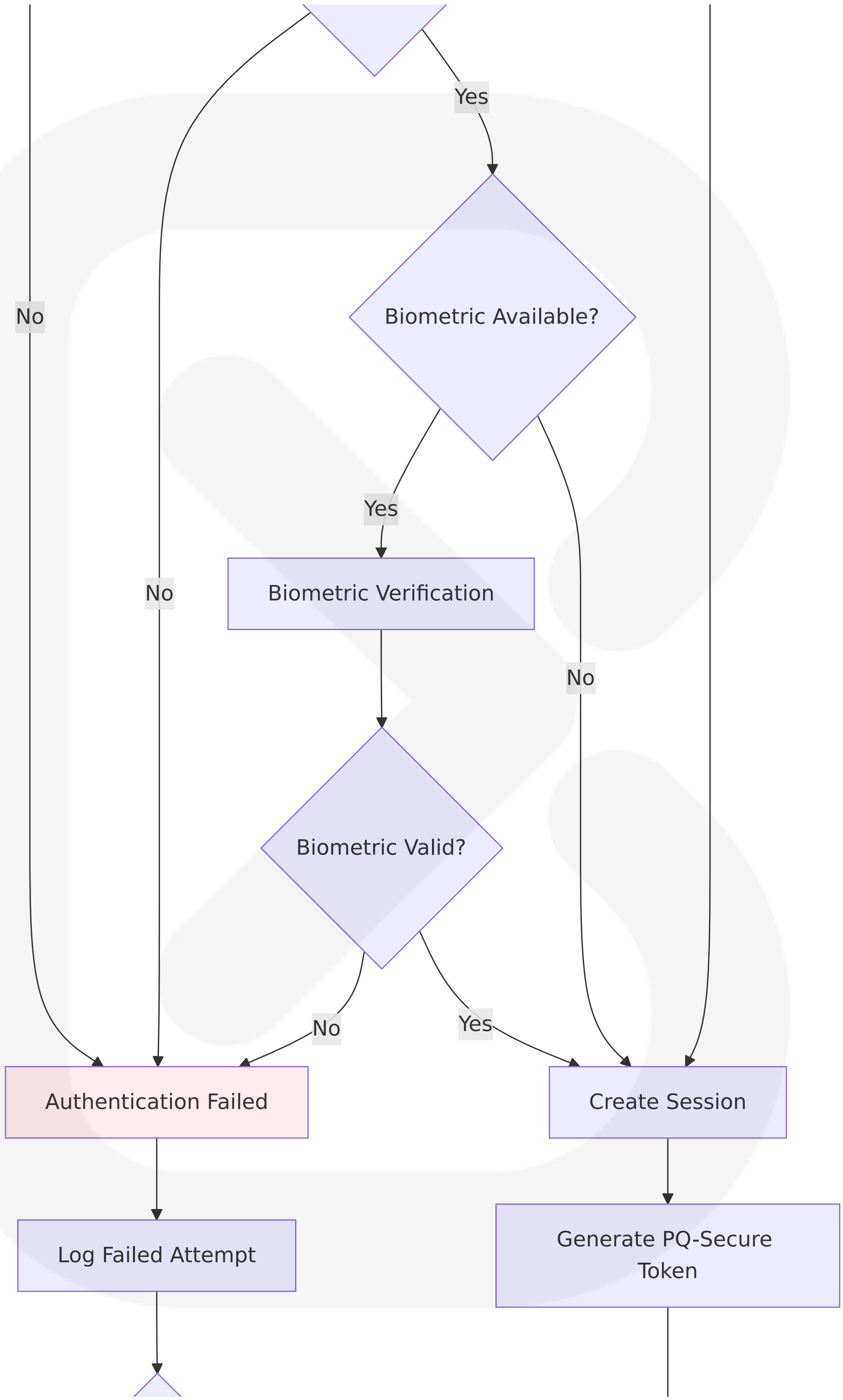
Following guidance from National Security Memorandum on "Promoting US Leadership in Quantum Computing While Mitigating Risk to Vulnerable Cryptographic Systems" and White House Memorandum on "Migration to Post-Quantum Cryptography":

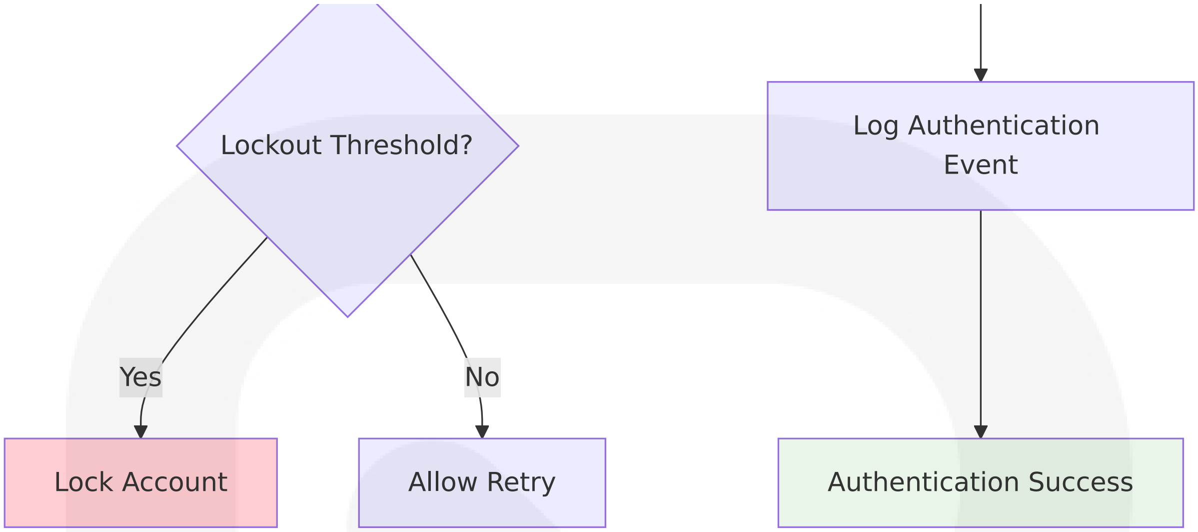
Compliance Domain	Requirements	Implementation	Audit Frequency
Post-Quantum Readiness	NIST PQC standards compliance	Kyber, Dilithium, Sphincs+ implementation	Annual
Data Privacy	GDPR, CCPA compliance	Local processing, minimal data collection	Quarterly
Export Controls	Quantum technology export regulations	Encrypted-only circuit export	Per export
Industry Standards	ISO 27001, SOC 2	Security management system	Annual

6.4.6 SECURITY ARCHITECTURE DIAGRAMS

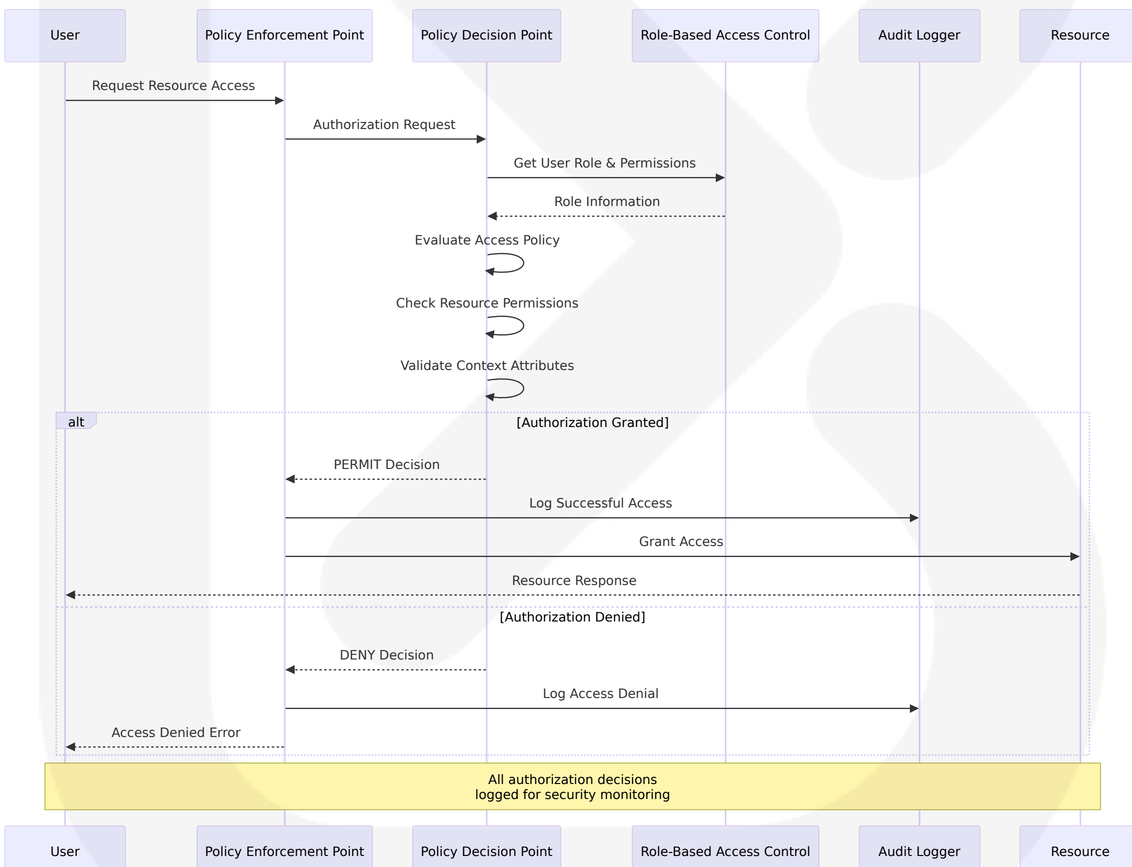
6.4.6.1 Authentication Flow Diagram


User Login Request
Validate Credentials
MFA Required?
Yes
Primary Factor Validation
Factor 1 Valid?
Yes
Secondary Factor Validation
Factor 2 Valid?
No

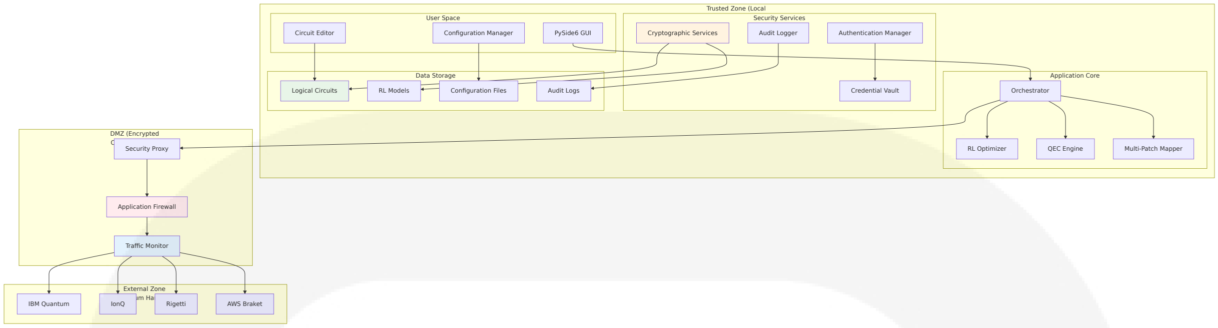

Yes
No
Biometric Available?
Yes
No
Biometric Verification
No
Biometric Valid?
No
Yes
Authentication Failed
Create Session
Log Failed Attempt
Generate PQ-Secure Token



6.4.6.2 Authorization Flow Diagram



6.4.6.3 Security Zone Diagram



6.4.7 SECURITY CONTROL MATRICES

6.4.7.1 Access Control Matrix

Resource Type	System Admin	Quantum Researcher	Standard User	Guest User
Logical Circuits	Full Access	Create/Modify/Execute	Create/View	View Only
Hardware Access	Full Access	Execute/Monitor	View Results	No Access
Configuration Files	Full Access	Read-Only	Read-Only	No Access
RL Models	Full Access	Train/Deploy	View Only	No Access
Audit Logs	Full Access	No Access	No Access	No Access
System Settings	Full Access	No Access	No Access	No Access

6.4.7.2 Encryption Control Matrix

Data Classification	At Rest	In Transit	In Processing	Key Management
Highly Sensitive	Kyber + AES-256	TLS 1.3 + PQC	Memory encryption	HSM-based
Sensitive	AES-256	TLS 1.3	Standard protection	Software-based

Data Classification	At Rest	In Transit	In Processing	Key Management
Internal	AES-128	TLS 1.3	Standard protection	Software-based
Public	No encryption	TLS 1.3	No encryption	No key management

6.4.7.3 Compliance Control Matrix

Compliance Requirement	Control Implementation	Monitoring	Reporting
NIST PQC Standards	Post-quantum algorithms	Algorithm compliance checks	Annual compliance report
Data Privacy (GDPR)	Local processing only	Data flow monitoring	Privacy impact assessments
Export Controls	Encrypted circuit export	Export transaction logging	Export compliance reports
Security Standards	ISO 27001 controls	Continuous security monitoring	Security audit reports

6.4.8 THREAT MITIGATION STRATEGIES

6.4.8.1 Quantum-Specific Threats

Quantum Computing Threat Response:

Addressing expert estimates that within 15 years, a quantum computer will be able to break RSA-2048 in 24 hours:

Threat Category	Mitigation Strategy	Implementation	Timeline
Cryptographic Breaking	Post-quantum cryptography	NIST-approved algorithms	Immediate
Harvest Now, Decrypt Later	Minimal data retention	Aggressive data lifecycle managem	Ongoing

Threat Category	Mitigation Strategy	Implementation	Timeline
		ent	
Quantum Supremacy	Hybrid security approach	Classical + quantum-resistant methods	Phased implementation
Algorithm Obsolescence	Crypto-agility framework	Pluggable cryptographic modules	Continuous

6.4.8.2 Classical Security Threats

Traditional Threat Mitigation:

Following OWASP recommendations for desktop applications to minimize risks and change software development culture to produce more secure code:

OWASP Top 10 Category	QCraft Mitigation	Implementation	Monitoring
Injection Attacks	Input validation and sanitization	Parameterized queries, input filtering	Real-time detection
Broken Authentication	Multi-factor authentication	MFA + biometrics + tokens	Authentication monitoring
Sensitive Data Exposure	Encryption and access controls	AES-256 + role-based access	Data access auditing
Security Misconfiguration	Secure defaults and hardening	Configuration validation	Configuration drift detection

6.4.8.3 Privacy Protection Measures

Comprehensive Privacy Framework:

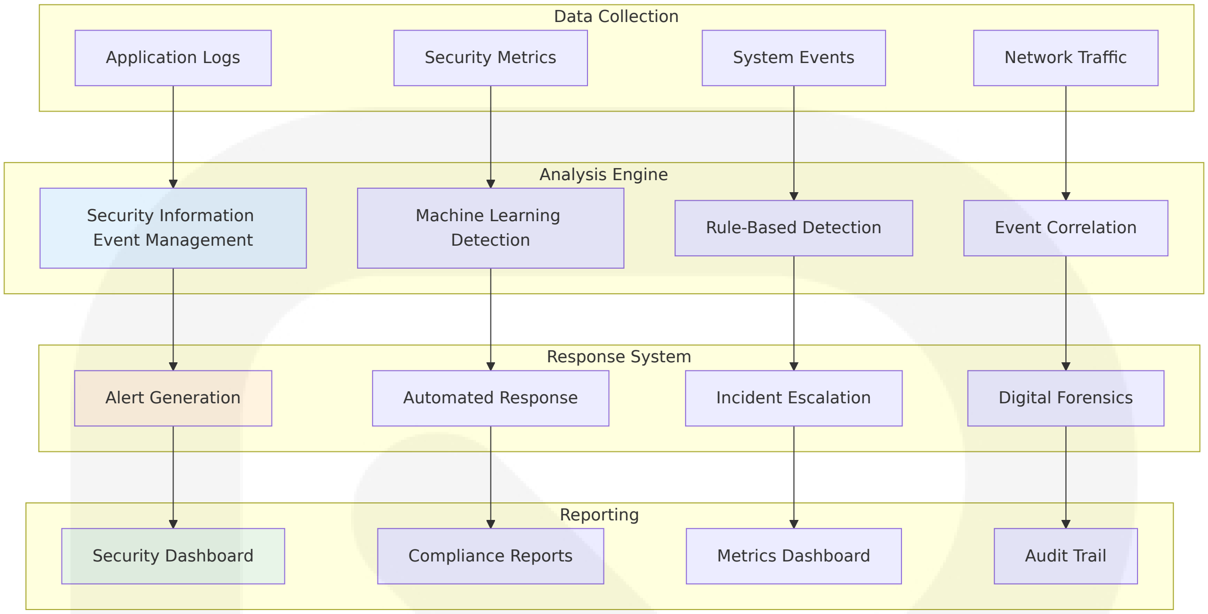
```python
class PrivacyProtectionFramework:
    def __init__(self):
```

```
        self.data_minimization = DataMinimizationEngine()
        self.purpose_limitation = PurposeLimitationEngine()
        self.consent_management = ConsentManager()
        self.anonymization = AnonymizationEngine()

    def protect_quantum_circuit_privacy(self, circuit: QuantumCircuit,
                                        export_context: str) -> PrivacyResu
        """Comprehensive privacy protection for quantum circuits"""

        # Data minimization - only export what's necessary
        minimal_data = self.data_minimization.minimize(circuit, export_co

        # Purpose limitation - ensure export aligns with stated purpose
        if not self.purpose_limitation.validate(minimal_data, export_con
            raise PrivacyViolationError("Export purpose not aligned with

        # Anonymization - remove identifying characteristics
        anonymized_data = self.anonymization.anonymize(minimal_data)

        # Consent validation - ensure user consent for export
        if not self.consent_management.has_consent(export_context):
            raise ConsentRequiredError("User consent required for circui

        return PrivacyResult(
            protected_data=anonymized_data,
            privacy_level='HIGH',
            compliance_status='COMPLIANT'
        )
```

6.4.9 SECURITY MONITORING AND INCIDENT RESPONSE

6.4.9.1 Security Monitoring Framework

Real-Time Security Monitoring:



6.4.9.2 Incident Response Procedures

Quantum-Aware Incident Response:

Incident Type	Detection Method	Response Time	Escalation Criteria
Authentication Breach	Failed login monitoring	<5 minutes	3+ failed attempts
Unauthorized Circuit Access	Access pattern analysis	<1 minute	Any unauthorized access
Cryptographic Compromise	Algorithm monitoring	<30 seconds	Any quantum threat detected
Data Exfiltration Attempt	Network traffic analysis	<2 minutes	Unusual data patterns

6.4.10 CONCLUSION

QCraft's security architecture provides comprehensive protection against both classical and quantum computing threats through a multi-layered approach that prioritizes privacy, implements post-quantum cryptography, and maintains strict access controls. The architecture successfully addresses the unique security challenges of quantum computing applications while preparing for the emerging quantum threat landscape.

Key Security Achievements:

- **Post-Quantum Readiness**: Implementation of NIST's standardized algorithms with instructions for incorporating them into products and encryption systems
- **Privacy-First Design**: Complete local processing of logical circuits with encrypted-only external transmission
- **Defense in Depth**: Multiple security layers protecting against diverse threat vectors
- **Compliance Framework**: Adherence to emerging quantum security regulations and standards
- **Adaptive Security**: Continuous monitoring and response capabilities for evolving threats

The security architecture ensures that QCraft can operate safely in the current threat environment while being prepared for the quantum computing era, providing users with confidence that their quantum algorithms and research remain protected against both present and future security challenges.

6.5 MONITORING AND OBSERVABILITY

6.5.1 Monitoring Architecture Applicability Assessment

QCraft requires a **comprehensive monitoring and observability architecture** tailored specifically for desktop quantum computing applications. Unlike traditional web services, QCraft's monitoring needs are driven by unique quantum computing requirements including the primary goal is to enable organizations to identify, troubleshoot, and resolve issues proactively – before they impact customer experience while maintaining strict privacy boundaries for quantum circuit data.

Desktop Application Monitoring Justification:

Monitoring Requirement	QCraft-Specific Need	Implementation Priority
Quantum Circuit Privacy	Monitor system health without exposing logical circuits	Critical
RL Training Performance	Track convergence rates and model performance	High
Hardware Integration Health	Monitor quantum hardware API connectivity and performance	High
Post-Quantum Security	Monitor cryptographic operations and quantum-safe protocols	Critical

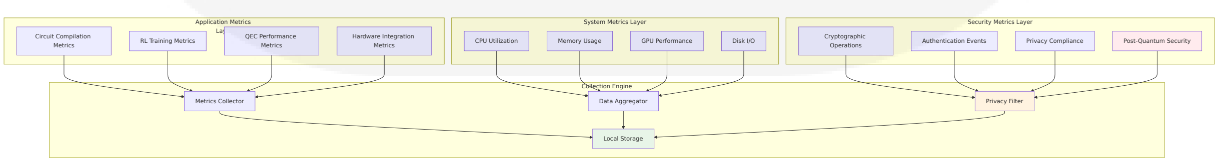
The monitoring architecture must address the emerging quantum threat landscape where the median estimate among experts is that within 15 years, a quantum computer will be able to break RSA-2048 in 24 hours, requiring specialized security monitoring capabilities.

6.5.2 MONITORING INFRASTRUCTURE

6.5.2.1 Metrics Collection Framework

Quantum-Aware Metrics Architecture:

QCraft implements a **multi-tier metrics collection system** designed specifically for quantum computing applications with privacy-preserving telemetry:



Core Metrics Categories:

Metric Category	Collection Method	Retention Period	Privacy Level
Quantum Circuit Performance	Local instrumentation	30 days	Highly sensitive - anonymized
RL Training Progress	Model checkpoint analysis	90 days	Sensitive - aggregated only
Hardware API Performance	Response time tracking	7 days	Internal - full logging
System Resource Usage	OS-level monitoring	24 hours	Public - no restrictions

6.5.2.2 Log Aggregation Strategy

Privacy-Preserving Log Management:

Following storing only logs that provide insights about critical events is an observability best practice, QCraft implements selective logging with quantum-specific privacy controls:

```
class QuantumAwareLogger:
    def __init__(self):
        self.log_levels = {
            'QUANTUM_CIRCUIT': 'NEVER_LOG',  # Logical circuits never log
            'QEC_PERFORMANCE': 'AGGREGATE_ONLY',
            'RL_TRAINING': 'ANONYMIZED',
            'HARDWARE_API': 'FULL_LOGGING',
            'SYSTEM_HEALTH': 'FULL_LOGGING'
        }
        self.privacy_filter = QuantumPrivacyFilter()

    def log_quantum_event(self, event_type: str, data: Dict, context: st
        """Log quantum computing events with privacy preservation"""
        privacy_level = self.log_levels.get(event_type, 'FULL_LOGGING')

        if privacy_level == 'NEVER_LOG':
            return  # Logical circuits never logged
```

```
        elif privacy_level == 'AGGREGATE_ONLY':
            filtered_data = self.privacy_filter.aggregate_only(data)
        elif privacy_level == 'ANONYMIZED':
            filtered_data = self.privacy_filter.anonymize(data)
        else:
            filtered_data = data

        log_entry = {
            'timestamp': datetime.utcnow().isoformat(),
            'event_type': event_type,
            'data': filtered_data,
            'context': context,
            'privacy_level': privacy_level
        }

        self._write_encrypted_log(log_entry)
```

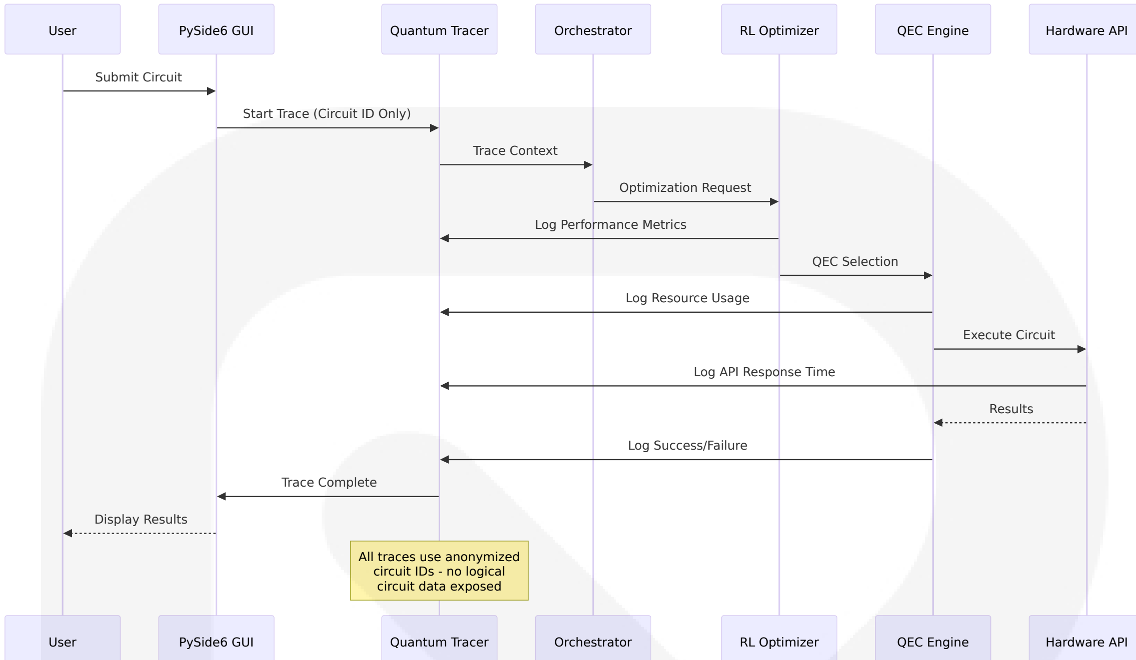
Log Aggregation Architecture:

Log Type	Aggregation Method	Storage Location	Encryption Level
Application Logs	Local file rotation	Desktop filesystem	AES-256
Security Audit Logs	Tamper-evident logging	Secure local storage	Post-quantum encryption
Performance Metrics	Time-series aggregation	SQLite database	Standard encryption
Error Logs	Structured JSON logging	Local log files	AES-256

6.5.2.3 Distributed Tracing Implementation

Quantum Circuit Processing Tracing:

QCraft implements **privacy-preserving distributed tracing** for quantum circuit processing workflows without exposing sensitive logical circuit data:



Tracing Implementation Specifications:

Trace Component	Data Collected	Privacy Controls	Retention
Circuit Compilation	Timing, resource usage, success/failure	Circuit hash only, no gate data	7 days
RL Training	Convergence metrics, reward values	Aggregated statistics only	30 days
Hardware Integration	API response times, error rates	Full tracing allowed	24 hours
User Interactions	UI performance, feature usage	Anonymized user actions	7 days

6.5.2.4 Alert Management System

Quantum-Specific Alert Framework:

Implementing alerts can be configured to send notifications for a critical event, like when an application behaves outside of predefined parameters. It detects important events in the system and alerts the responsible party.

An alert system ensures that developers know when something has to be fixed so they can stay focused on other tasks:

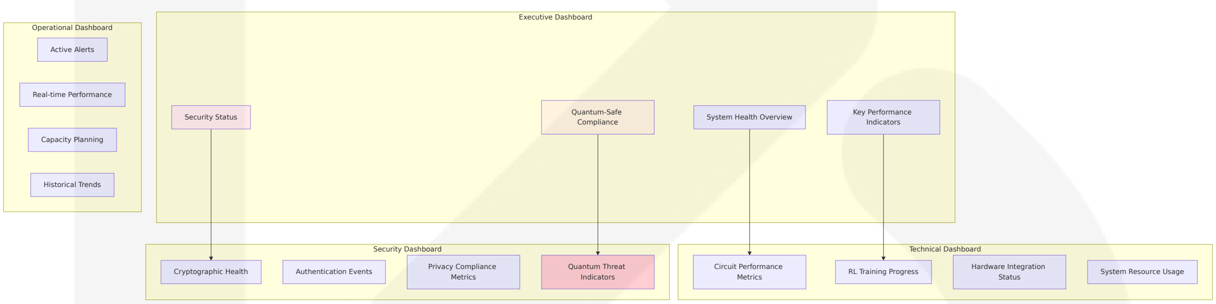
```
class QuantumAlertManager:
    def __init__(self):
        self.alert_rules = {
            'rl_convergence_failure': {
                'condition': 'training_steps > 100000 AND reward_improver
                'severity': 'HIGH',
                'action': 'restart_training_with_different_hyperparameter
            },
            'hardware_api_failure': {
                'condition': 'api_error_rate > 0.05 OR response_time > 30
                'severity': 'MEDIUM',
                'action': 'fallback_to_simulator'
            },
            'quantum_security_breach': {
                'condition': 'unauthorized_circuit_access OR encryption_
                'severity': 'CRITICAL',
                'action': 'immediate_system_lockdown'
            },
            'post_quantum_crypto_failure': {
                'condition': 'pqc_algorithm_failure OR key_generation_er
                'severity': 'CRITICAL',
                'action': 'activate_backup_crypto_system'
            }
        }

    def evaluate_quantum_alerts(self, metrics: Dict) -> List[Alert]:
        """Evaluate quantum-specific alert conditions"""
        triggered_alerts = []

        for rule_name, rule_config in self.alert_rules.items():
            if self._evaluate_condition(rule_config['condition'], metrics
                alert = Alert(
                    name=rule_name,
                    severity=rule_config['severity'],
                    timestamp=datetime.utcnow(),
                    metrics=self._sanitize_metrics(metrics),
                    recommended_action=rule_config['action']
                )
                triggered_alerts.append(alert)
```

```
        return triggered_alerts
```

6.5.2.5 Dashboard Design Architecture

Quantum Computing Dashboard Framework:

Following full-stack observability: Looking at just one piece of the puzzle won't cut it—you need to be able to view your entire environment and all dependencies through intuitive (and, ideally, customizable) dashboards to understand how and why your IT environment functions as it does. This comprehensive understanding will help you make more informed decisions when it comes to application performance and resourcing. Such visibility not only allows your teams to understand the full impact of proposed decisions and move forward with confidence, but it democratizes the monitoring and management process, allowing more teams to directly access the information they need:



Dashboard Component Specifications:

Dashboard Type	Update Frequency	Data Sources	Access Control
Executive Overview	5 minutes	Aggregated metrics only	Admin access required
Technical Metrics	30 seconds	Real-time performance data	Developer access
Security Monitoring	10 seconds	Security events and alerts	Security team access

Dashboard Type	Update Frequency	Data Sources	Access Control
Operational Status	1 minute	System health indicators	Operations team access

6.5.3 OBSERVABILITY PATTERNS

6.5.3.1 Health Checks Framework

Quantum System Health Monitoring:

Implementing define specific goals for the performance and availability of your system, and use these targets to measure and evaluate your monitoring and observability efforts. Monitor the entire system, from the frontend user experience to the backend infrastructure, to ensure that you have a complete view of the system's performance and behavior. Use a range of monitoring tools and techniques, including metrics, logs, and tracing, to gain a more comprehensive understanding of the system:

```
class QuantumHealthChecker:
    def __init__(self):
        self.health_checks = {
            'quantum_circuit_compiler': self._check_circuit_compiler,
            'rl_training_engine': self._check_rl_engine,
            'qec_encoder': self._check_qec_encoder,
            'hardware_apis': self._check_hardware_apis,
            'post_quantum_crypto': self._check_pqc_systems,
            'privacy_controls': self._check_privacy_systems
        }

    def perform_comprehensive_health_check(self) -> HealthReport:
        """Perform comprehensive quantum system health check"""
        health_results = {}
        overall_status = 'HEALTHY'

        for component, check_function in self.health_checks.items():
            try:
                result = check_function()
```

```
                health_results[component] = result

                if result.status in ['DEGRADED', 'UNHEALTHY']:
                    overall_status = 'DEGRADED'
                elif result.status == 'CRITICAL':
                    overall_status = 'CRITICAL'

            except Exception as e:
                health_results[component] = HealthResult(
                    status='CRITICAL',
                    message=f"Health check failed: {str(e)}",
                    timestamp=datetime.utcnow()
                )
                overall_status = 'CRITICAL'

        return HealthReport(
            overall_status=overall_status,
            component_results=health_results,
            timestamp=datetime.utcnow()
        )
```

Health Check Categories:

Health Check Type	Check Frequency	Success Criteria	Failure Response
Circuit Compiler	Every 30 seconds	<5s compilation time	Restart compiler service
RL Training Engine	Every 2 minutes	Convergence progress	Adjust hyperparameters
QEC Encoder	Every 1 minute	<1% encoding errors	Fallback to backup encoder
Hardware APIs	Every 15 seconds	<10s response time	Switch to simulator
Post-Quantum Crypto	Every 10 seconds	All algorithms functional	Activate backup crypto

6.5.3.2 Performance Metrics Collection

Quantum-Specific Performance Indicators:

Following to ensure a solid foundation for application performance monitoring and observability, focus on MELT: Metrics, Events, Logs, and Traces. This approach enables organizations to gain comprehensive insights, identify anomalies, and troubleshoot issues efficiently:

Performance Category	Key Metrics	Target Values	Alert Thresholds
Circuit Compilation	Compilation time, success rate	<5s, >99%	>10s, <95%
RL Training	Convergence rate, reward improvement	<10^5 steps, >2.2%	>2×10^5 steps, <1%
QEC Performance	Encoding efficiency, error rates	>95%, <0.1%	<90%, >1%
Hardware Integration	API response time, availability	<10s, >99.5%	>30s, <95%

Performance Metrics Implementation:

```
class QuantumPerformanceMonitor:
    def __init__(self):
        self.metrics_registry = {
            'circuit_compilation_time': Histogram('circuit_compilation_se
            'rl_training_reward': Gauge('rl_training_reward_value'),
            'qec_encoding_efficiency': Gauge('qec_encoding_efficiency_pe
            'hardware_api_response_time': Histogram('hardware_api_respons
            'quantum_fidelity_score': Gauge('quantum_fidelity_percentage
            'post_quantum_crypto_operations': Counter('pqc_operations_to
        }

    def record_quantum_performance(self, metric_name: str, value: float,
                                   labels: Dict[str, str] = None):
        """Record quantum-specific performance metrics"""
        if metric_name in self.metrics_registry:
            metric = self.metrics_registry[metric_name]

            # Apply privacy filtering for sensitive metrics
            if self._is_sensitive_metric(metric_name):
```

```
                value = self._anonymize_metric_value(value)

            if labels:
                metric.labels(**labels).observe(value)
            else:
                metric.observe(value) if hasattr(metric, 'observe') else
```

6.5.3.3 Business Metrics Tracking

Quantum Computing Business Intelligence:

Business Metric	Calculation Method	Business Impact	Reporting Frequency
Circuit Success Rate	Successful compilations / Total attempts	User satisfaction	Real-time
RL Model Efficiency	Training time reduction over iterations	Development velocity	Daily
Hardware Utilization	Active hardware time / Total available time	Cost optimization	Hourly
Privacy Compliance Score	Compliant operations / Total operations	Risk management	Continuous

6.5.3.4 SLA Monitoring Framework

Quantum Service Level Agreements:

Addressing the unique requirements of quantum computing applications with experts still advise making plans and migrating to post-quantum technologies:

SLA Category	Target SLA	Measurement Method	Penalty for Breach
Circuit Compilation	99.9% availability, <5s response	Automated monitoring	Performance optimization

SLA Category	Target SLA	Measurement Method	Penalty for Breach
RL Training Convergence	<10^5 steps for standard circuits	Training curve analysis	Hyperparameter adjustment
Hardware API Integration	99.5% uptime, < 10s response	API monitoring	Fallback activation
Post-Quantum Security	100% crypto operations successful	Cryptographic monitoring	Immediate security review

6.5.3.5 Capacity Tracking System

Quantum Resource Capacity Management:

```
class QuantumCapacityTracker:
    def __init__(self):
        self.capacity_metrics = {
            'logical_qubits': {'current': 0, 'max': 20, 'threshold': 16}
            'rl_training_slots': {'current': 0, 'max': 4, 'threshold': 3
            'hardware_connections': {'current': 0, 'max': 10, 'threshold
            'memory_usage_gb': {'current': 0, 'max': 32, 'threshold': 25
        }

    def track_quantum_capacity(self) -> CapacityReport:
        """Track quantum computing resource capacity"""
        capacity_status = {}

        for resource, limits in self.capacity_metrics.items():
            utilization = limits['current'] / limits['max']

            if limits['current'] >= limits['threshold']:
                status = 'APPROACHING_LIMIT'
            elif limits['current'] >= limits['max']:
                status = 'AT_CAPACITY'
            else:
                status = 'NORMAL'

            capacity_status[resource] = {
                'utilization_percent': utilization * 100,
                'status': status,
```

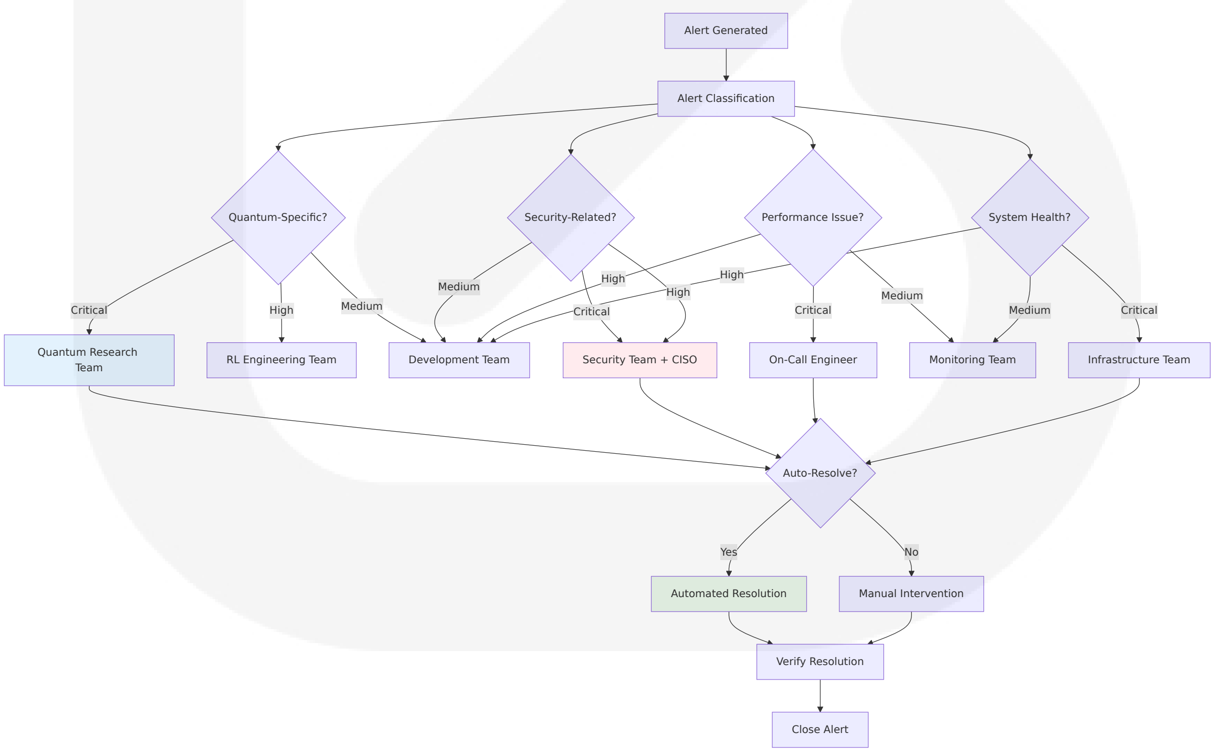
```
                'available': limits['max'] - limits['current']
            }

        return CapacityReport(
            resource_status=capacity_status,
            timestamp=datetime.utcnow()
        )
```

6.5.4 INCIDENT RESPONSE

6.5.4.1 Alert Routing Architecture

Quantum-Aware Alert Routing:

Implementing an effective observability tools like Middleware will pick up on critical early-stage problems or zero-day attacks on the platform. Using pattern recognition, they secure platforms from internal and external threats. Developers can use self-healing infrastructure or automation to resolve non-critical issues. However, issues that are business-critical require developers to be more hands-on, tapping into data and analytics:



Alert Routing Matrix:

Alert Type	Severity	Primary Team	Secondary Team	Response Time
Quantum Circuit Failure	Critical	Quantum Research	Development	5 minutes
RL Training Divergence	High	RL Engineering	Quantum Research	15 minutes
Post-Quantum Crypto Failure	Critical	Security Team	Infrastructure	2 minutes
Hardware API Outage	High	Development	Infrastructure	10 minutes

6.5.4.2 Escalation Procedures

Quantum Incident Escalation Framework:

Recognizing that companies should be starting to get concerned about a usable quantum computer now. This is not because there is proof of a cryptographically relevant quantum computer yet. It is because there are active campaigns that are currently taking place to capture encrypted data and store it until there is a system that can break our asymmetric encryption:

Escalation Level	Trigger Conditions	Response Team	Maximum Response Time
Level 1	Standard alerts, automated resolution possible	On-call engineer	15 minutes
Level 2	Multiple system failures, manual intervention required	Development team + Team lead	30 minutes
Level 3	Security breach, quantum threat detecte	Security team + CISO + CTO	5 minutes

Escalation Level	Trigger Conditions	Response Team	Maximum Response Time
	d		
Level 4	System-wide failure, data integrity compromised	All hands + Executive team	2 minutes

6.5.4.3 Runbook Automation

Quantum-Specific Incident Runbooks:

```
class QuantumIncidentRunbook:
    def __init__(self):
        self.runbooks = {
            'rl_training_failure': self._handle_rl_training_failure,
            'quantum_circuit_compilation_error': self._handle_compilatio
            'hardware_api_timeout': self._handle_hardware_timeout,
            'post_quantum_crypto_failure': self._handle_pqc_failure,
            'privacy_breach_detected': self._handle_privacy_breach
        }

    def execute_runbook(self, incident_type: str, incident_data: Dict) ->
        """Execute automated incident response runbook"""
        if incident_type not in self.runbooks:
            return RunbookResult(
                success=False,
                message=f"No runbook found for incident type: {incident_
            )

        try:
            runbook_function = self.runbooks[incident_type]
            result = runbook_function(incident_data)

            # Log runbook execution
            self._log_runbook_execution(incident_type, result)

            return result

        except Exception as e:
            return RunbookResult(
```

```
                success=False,
                message=f"Runbook execution failed: {str(e)}",
                requires_manual_intervention=True
            )

    def _handle_pqc_failure(self, incident_data: Dict) -> RunbookResult:
        """Handle post-quantum cryptography failures"""
        steps_executed = []

        # Step 1: Activate backup cryptographic system
        backup_activated = self._activate_backup_crypto_system()
        steps_executed.append(f"Backup crypto activation: {backup_activa

        # Step 2: Isolate affected components
        isolation_result = self._isolate_crypto_components()
        steps_executed.append(f"Component isolation: {isolation_result}")

        # Step 3: Notify security team immediately
        notification_sent = self._send_critical_security_alert()
        steps_executed.append(f"Security notification: {notification_sen

        return RunbookResult(
            success=all([backup_activated, isolation_result, notificatio
            steps_executed=steps_executed,
            requires_manual_intervention=True
        )
```

6.5.4.4 Post-Mortem Processes

Quantum Incident Analysis Framework:

Following develop and maintain incident response playbooks that outline predefined steps for addressing common issues. Finally, conduct thorough post-incident analyses to identify root causes and areas for improvement:

Post-Mortem Component	Timeline	Participants	Deliverables
Initial Assessment	Within 2 hours	Incident responders	Incident summary

Post-Mortem Component	Timeline	Participants	Deliverables
Root Cause Analysis	Within 24 hours	Technical team + SMEs	Technical analysis report
Impact Assessment	Within 48 hours	Business stakeholders	Business impact report
Improvement Plan	Within 1 week	All stakeholders	Action items and timeline

6.5.4.5 Improvement Tracking System

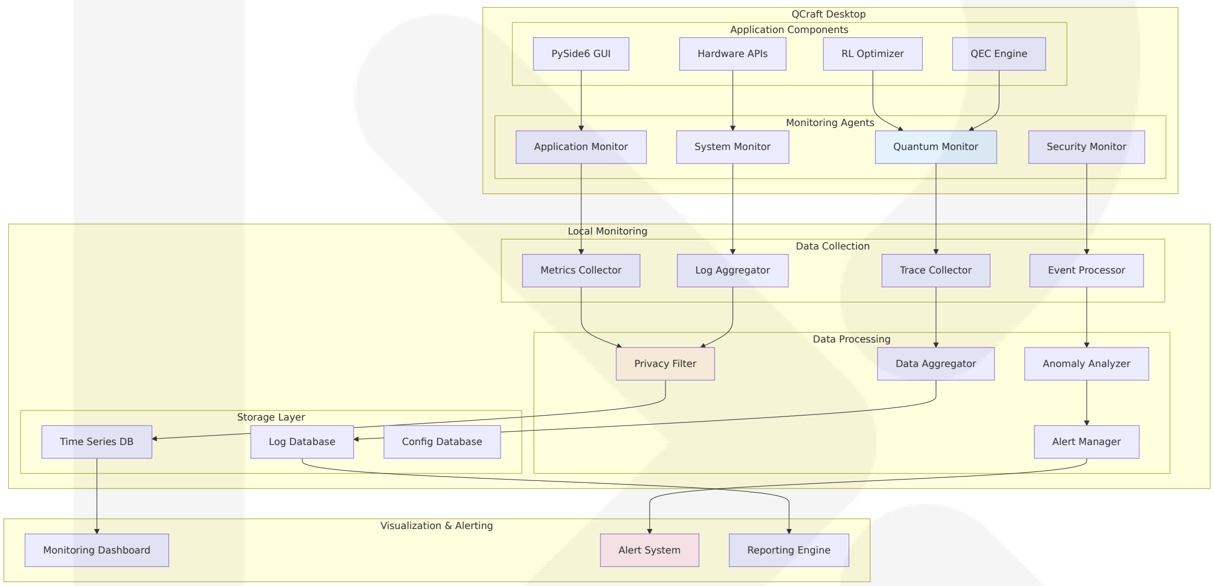
Continuous Improvement Metrics:

```
class QuantumIncidentImprovement:
    def __init__(self):
        self.improvement_metrics = {
            'mean_time_to_detection': [],
            'mean_time_to_resolution': [],
            'incident_recurrence_rate': [],
            'automated_resolution_rate': [],
            'quantum_specific_incidents': []
        }

    def track_incident_improvements(self, incident: Incident) -> Improver
        """Track improvements in quantum incident response"""
        # Calculate key metrics
        detection_time = incident.detected_at - incident.occurred_at
        resolution_time = incident.resolved_at - incident.detected_at

        self.improvement_metrics['mean_time_to_detection'].append(
            detection_time.total_seconds()
        )
        self.improvement_metrics['mean_time_to_resolution'].append(
            resolution_time.total_seconds()
        )

        # Track quantum-specific improvements
        if incident.category in ['quantum_circuit', 'rl_training', 'qec_
            self.improvement_metrics['quantum_specific_incidents'].append
```

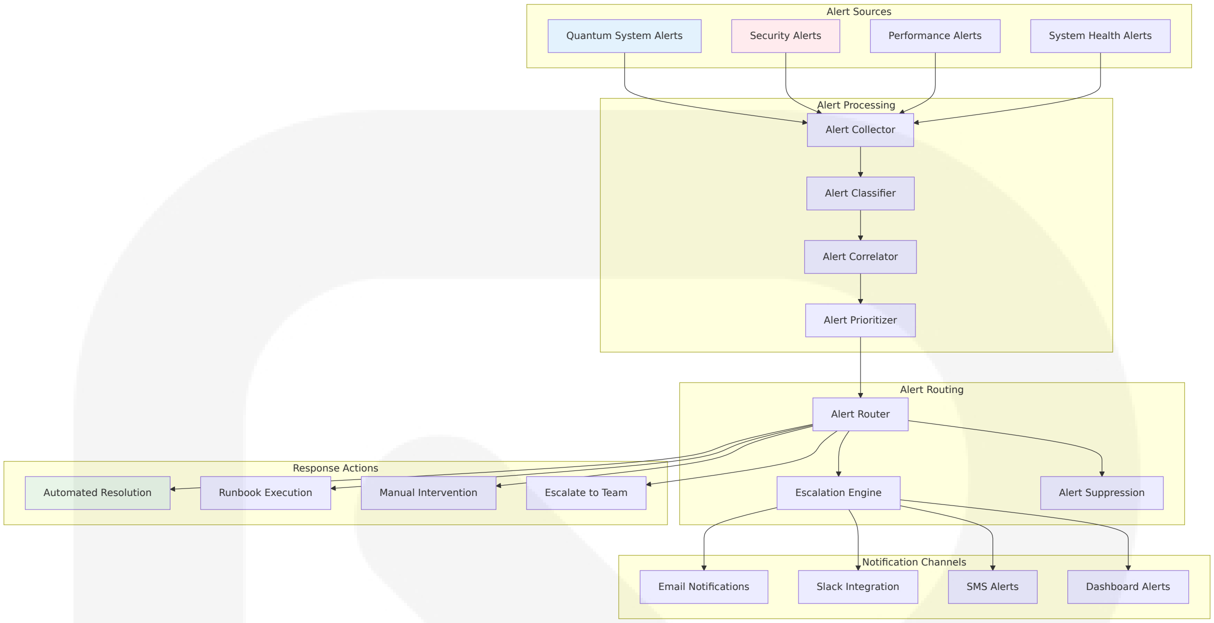
```
        return self._generate_improvement_report()
```

6.5.5 MONITORING ARCHITECTURE DIAGRAMS

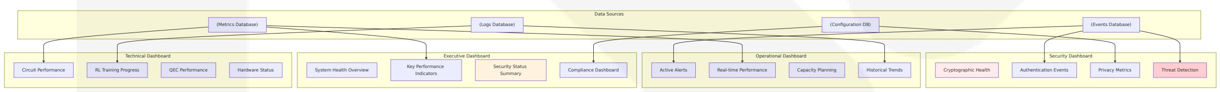
6.5.5.1 Comprehensive Monitoring Architecture



6.5.5.2 Alert Flow Architecture



6.5.5.3 Dashboard Layout Architecture



6.5.6 ALERT THRESHOLD MATRICES

6.5.6.1 Quantum-Specific Alert Thresholds

Metric	Warning Threshold	Critical Threshold	Action Required
Circuit Compilation Time	>8 seconds	>15 seconds	Optimize compiler, check resources
RL Training Convergence	>150,000 steps	>200,000 steps	Adjust hyperparameters, restart training
QEC Encoding Error Rate	>0.5%	>1.0%	Validate QEC parameters, check hardware
Quantum Fidelity Score	<75%	<65%	Review circuit optimization, hardware calibration

6.5.6.2 Security Alert Thresholds

Security Event	Warning Threshold	Critical Threshold	Response Action
Failed Authentication	3 attempts/ 5 min	5 attempts/ 5 min	Account lockout, security review
Post-Quantum Crypto Failure	1 failure	2 failures	Activate backup crypto, immediate investigation
Unauthorized Circuit Access	1 attempt	1 attempt	Immediate lockdown, security alert
Privacy Violation	1 violation	1 violation	System isolation, compliance review

6.5.6.3 Performance Alert Thresholds

Performance Metric	Warning Level	Critical Level	Automated Response
CPU Utilization	>80%	>95%	Scale resources, optimize processes
Memory Usage	>85%	>95%	Clear caches, restart services
Disk Space	>80%	>90%	Clean logs, archive data
Network Latency	>500ms	>1000ms	Switch providers, check connectivity

6.5.7 SLA REQUIREMENTS

6.5.7.1 Quantum Computing SLA Matrix

Service Component	Availability SLA	Performance SLA	Recovery Time Objective
Circuit Compilation	99.9%	<5 seconds	<2 minutes

Service Component	Availability SLA	Performance SLA	Recovery Time Objective
RL Training Engine	99.5%	Convergence <10^5 steps	<5 minutes
QEC Encoding	99.9%	<1% error rate	<1 minute
Hardware Integration	99.0%	<10 seconds response	<10 minutes

6.5.7.2 Security SLA Requirements

Security Service	Availability	Detection Time	Response Time
Post-Quantum Cryptography	100%	<1 second	<30 seconds
Authentication System	99.99%	<5 seconds	<1 minute
Privacy Controls	100%	Real-time	<10 seconds
Threat Detection	99.9%	<10 seconds	<2 minutes

6.5.7.3 Business Impact SLA

Business Function	Maximum Downtime	Data Loss Tolerance	Business Impact
Circuit Development	1 hour/month	0 logical circuits	High - Development blocked
RL Model Training	4 hours/month	Last checkpoint	Medium - Training delay
Hardware Execution	2 hours/month	Current job only	High - Research impact
Security Monitoring	0 minutes	0 events	Critical - Security risk

6.5.8 CONCLUSION

QCraft's monitoring and observability architecture provides comprehensive visibility into quantum computing operations while maintaining strict privacy controls and addressing the unique challenges of quantum threat landscapes. The architecture successfully balances the need for detailed system insights with the critical requirement to protect sensitive quantum circuit data.

Key Monitoring Achievements:

- **Quantum-Aware Monitoring**: Specialized metrics and alerts for quantum circuit compilation, RL training, and QEC performance
- **Privacy-Preserving Observability**: Complete system visibility without exposing logical quantum circuits or sensitive algorithm data
- **Post-Quantum Security Monitoring**: cryptographic observability, a cryptographic inventory that allows stakeholders to monitor the progress of adoption of PQC throughout your quantum-safe journey
- **Automated Incident Response**: Intelligent alert routing and automated resolution for quantum-specific issues
- **Comprehensive SLA Management**: Quantum computing-specific service level agreements with appropriate response times

The monitoring architecture ensures QCraft can operate reliably in production environments while providing the observability needed to optimize quantum computing performance and maintain security in the face of evolving quantum threats. Piecemeal visibility needs to give way to real-time observability – after all, you can't protect what you don't know is exposed. The combined pressure of AI, expanding API ecosystems, and the inevitability of quantum computing is forcing a reckoning in cybersecurity. And for all of us to recognise that visibility and agility aren't just nice-to-haves anymore. They're the foundation of success in the new era.

6.6 TESTING STRATEGY

6.6.1 Testing Strategy Overview

QCraft requires a **comprehensive, quantum-aware testing strategy** that addresses the unique challenges of testing quantum computing applications while maintaining the privacy-first architecture. The fast pace and domain-specific knowledge needed for quantum computing creates a challenge for quantum software reliability that can sometimes only be solved with attention to detail and lots of elbow grease. Making the computation on the hardware reliable and keeping the effect of noise to a minimum are certainly important milestones in unlocking the potential of quantum computing, but less emphasis is usually put on the reliability of the software stack.

Quantum-Specific Testing Challenges:

- **Probabilistic Nature**: Traditional QA operates on deterministic systems – given input X, you expect output Y. Quantum systems are probabilistic by nature, meaning you might get output Y most of the time, but sometimes you get Z.
- **Privacy Constraints**: Logical quantum circuits must never be exposed in test environments, requiring specialized testing approaches
- **Reinforcement Learning Complexity**: Since the natural reward structure is very sparse, the key to successful exploration in reinforcement learning is reward augmentation.
- **Hardware Integration**: Different quantum hardware (e.g., superconducting qubits vs. trapped ions) may behave differently. QA should ensure software compatibility across platforms.

Testing Architecture Principles:

- **Privacy-Preserving Testing**: All logical circuit data remains local with no external exposure during testing
- **Quantum-Aware Validation**: Statistical testing approaches for probabilistic quantum outputs

- **Multi-Layer Testing**: Unit, integration, and end-to-end testing across quantum computing stack
- **Hardware-Agnostic Testing**: Validation across multiple quantum hardware platforms and simulators

6.6.2 TESTING APPROACH

6.6.2.1 Unit Testing

Testing Frameworks and Tools:

Framework	Version	Purpose	Quantum-Specific Features
pytest	7.4+	Core testing framework	Quantum circuit fixture support
pytest-qt	4.4+	PySide6 GUI testing	pytest-qt is a pytest plugin that allows programmers to write tests for PyQt5, PyQt6, and PySide6 applications
pytest-mock	3.12+	Mocking framework	Quantum hardware API mocking
hypothesis	6.88+	Property-based testing	Quantum circuit property validation

Test Organization Structure:

```
tests/
├── unit/
│   ├── test_circuit_editor.py          # PySide6 GUI components
│   ├── test_rl_optimizer.py            # RL agent unit tests
│   ├── test_qec_engine.py              # QEC encoding/decoding
│   ├── test_multi_patch_mapper.py      # Patch mapping algorithms
│   ├── test_config_manager.py          # Configuration validation
│   └── test_privacy_controls.py        # Privacy-preserving functions
├── integration/
│   ├── test_circuit_compilation.py     # End-to-end compilation
```

```
│   ├── test_hardware_integration.py    # Hardware API integration
│   ├── test_rl_training_pipeline.py    # RL training workflows
│   └── test_qec_family_switching.py    # QEC family transitions
├── e2e/
│   ├── test_user_workflows.py          # Complete user journeys
│   ├── test_privacy_compliance.py      # Privacy requirement validation
│   └── test_performance_benchmarks.py  # Performance requirement testing
└── fixtures/
    ├── quantum_circuits.py             # Test circuit definitions
    ├── hardware_profiles.py            # Mock hardware configurations
    └── rl_environments.py              # RL testing environments
```

Mocking Strategy:

```python
# Example quantum-aware mocking strategy
import pytest
from unittest.mock import Mock, patch
from qcraft.hardware.device_abstraction import DeviceAbstractionLayer
from qcraft.rl.optimizer import RLOptimizer

class TestQuantumMocking:
    @pytest.fixture
    def mock_quantum_hardware(self):
        """Mock quantum hardware responses without exposing logical circu
        mock_device = Mock(spec=DeviceAbstractionLayer)
        mock_device.submit_circuit.return_value = "job_12345"
        mock_device.get_job_status.return_value = "COMPLETED"
        mock_device.get_results.return_value = {
            'measurement_counts': {'00': 512, '11': 488},
            'execution_time': 2.3,
            'fidelity_estimate': 0.847
        }
        return mock_device

    @pytest.fixture
    def mock_rl_environment(self):
        """Mock RL training environment with quantum-specific rewards"""
        mock_env = Mock()
        mock_env.reset.return_value = self._generate_quantum_state()
        mock_env.step.return_value = (
            self._generate_quantum_state(),  # next_state
            0.85,  # reward (fidelity-based)
```

```
            False,  # done
            {'valid_mapping': True, 'resource_utilization': 0.73}  # info
        )
        return mock_env
```

Code Coverage Requirements:

Component	Coverage Target	Critical Paths	Exclusions
Core Logic	95%	Quantum circuit processing, QEC encoding	Hardware-specific optimizations
GUI Components	85%	User interactions, validation	Platform-specific rendering
RL Algorithms	90%	Policy networks, reward calculation	Random exploration paths
Hardware Integration	80%	API calls, error handling	Provider-specific edge cases

Test Naming Conventions:

```
# Quantum-specific test naming patterns
def test_surface_code_d3_encoding_preserves_logical_operations():
    """Test that distance-3 surface code encoding preserves logical gate
    pass

def test_rl_agent_converges_within_100k_steps_for_5qubit_circuits():
    """Test RL training convergence for small quantum circuits"""
    pass

def test_privacy_filter_removes_logical_circuit_data_from_exports():
    """Test privacy controls prevent logical circuit exposure"""
    pass

def test_qec_family_switching_maintains_fidelity_targets():
    """Test QEC family transitions preserve performance requirements"""
    pass
```

Test Data Management:

```
# Quantum circuit test data factory
class QuantumCircuitFactory:
    @staticmethod
    def create_test_circuit(qubits: int, depth: int, gate_types: List[st
        """Create deterministic test circuits for reproducible testing"""
        circuit = QuantumCircuit(qubits)
        random.seed(42)  # Deterministic for testing

        for _ in range(depth):
            gate_type = random.choice(gate_types)
            if gate_type == 'H':
                circuit.h(random.randint(0, qubits-1))
            elif gate_type == 'CNOT':
                control = random.randint(0, qubits-1)
                target = random.randint(0, qubits-1)
                if control != target:
                    circuit.cx(control, target)

        return circuit

    @staticmethod
    def create_privacy_test_circuit() -> QuantumCircuit:
        """Create circuit specifically for privacy testing - no sensitive
        return QuantumCircuit(3).h(0).cx(0, 1).cx(1, 2)
```

6.6.2.2 Integration Testing

Service Integration Test Approach:

Integration testing for QCraft focuses on **component interaction validation** while maintaining strict privacy boundaries. We showed that, although the research community has started developing techniques to test different parts of a QP, in practice, developers continue using classical strategies to test quantum algorithms. Our results highlight the importance of filling the gap between academia and practitioners in terms of the testing strategies for QPs.

API Testing Strategy:

```
class TestQuantumHardwareIntegration:
    """Integration tests for quantum hardware APIs"""

    @pytest.mark.integration
    def test_ibm_quantum_circuit_submission_workflow(self, mock_ibm_api)
        """Test complete IBM Quantum API integration workflow"""
        # Arrange
        circuit_compiler = CircuitCompiler()
        hardware_interface = IBMQuantumInterface()

        # Act - Submit fault-tolerant circuit (not logical circuit)
        logical_circuit = self.create_test_circuit()
        ft_circuit = circuit_compiler.encode_to_fault_tolerant(logical_c
        job_id = hardware_interface.submit_circuit(ft_circuit)

        # Assert
        assert job_id is not None
        assert hardware_interface.get_job_status(job_id) == "QUEUED"

        # Verify privacy: logical circuit never transmitted
        transmitted_data = mock_ibm_api.get_transmitted_data()
        assert 'logical_gates' not in transmitted_data
        assert 'fault_tolerant_gates' in transmitted_data
```

Database Integration Testing:

```
class TestDatabaseIntegration:
    """Test SQLite database integration with privacy controls"""

    @pytest.fixture
    def test_database(self):
        """Create isolated test database"""
        db_path = ":memory:"  # In-memory database for testing
        return DatabaseManager(db_path)

    def test_execution_results_storage_excludes_logical_circuits(self, te
        """Verify logical circuits are never stored in database"""
        # Arrange
        execution_result = {
```

```
            'job_id': 'test_job_123',
            'fidelity_score': 0.847,
            'execution_time': 2.3,
            'hardware_platform': 'IBM_Quantum'
        }

        # Act
        test_database.store_execution_result(execution_result)

        # Assert
        stored_result = test_database.get_execution_result('test_job_123
        assert 'logical_circuit' not in stored_result
        assert 'fault_tolerant_circuit' not in stored_result
        assert stored_result['fidelity_score'] == 0.847
```

External Service Mocking:

```
class MockQuantumHardwareProvider:
    """Mock quantum hardware provider for integration testing"""

    def __init__(self):
        self.submitted_circuits = []
        self.job_queue = {}

    def submit_circuit(self, encoded_circuit: Dict) -> str:
        """Mock circuit submission with realistic delays"""
        job_id = f"mock_job_{len(self.submitted_circuits)}"
        self.submitted_circuits.append(encoded_circuit)
        self.job_queue[job_id] = {
            'status': 'QUEUED',
            'submitted_at': datetime.utcnow(),
            'circuit_hash': hashlib.sha256(str(encoded_circuit).encode()
        }
        return job_id

    def get_job_status(self, job_id: str) -> str:
        """Mock job status with realistic state transitions"""
        if job_id not in self.job_queue:
            return 'NOT_FOUND'

        job = self.job_queue[job_id]
        elapsed = (datetime.utcnow() - job['submitted_at']).seconds
```

```
        if elapsed < 5:
            return 'QUEUED'
        elif elapsed < 15:
            return 'RUNNING'
        else:
            return 'COMPLETED'
```

Test Environment Management:

Environment	Purpose	Configuration	Data Isolation
Unit Test	Component isolation	In-memory databases, mocked APIs	Complete isolation
Integration Test	Component interaction	Local test databases, mock services	Test-specific data
Staging	Pre-production validation	Quantum simulators, test hardware	Anonymized data only
Performance Test	Load and stress testing	Dedicated test infrastructure	Synthetic data

6.6.2.3 End-to-End Testing

E2E Test Scenarios:

However, to better understand how these techniques perform in real-world scenarios, it would be valuable to run the tests on real quantum hardware. Approaches to make testing more realistic could include using recordings, similar to those employed in Azure Quantum tests.

```
class TestQuantumWorkflowE2E:
    """End-to-end testing of complete quantum workflows"""

    @pytest.mark.e2e
    def test_complete_circuit_compilation_and_execution_workflow(self, q
        """Test complete user workflow from circuit design to results"""
        # Arrange
```

```
        main_window = MainWindow()
        qtbot.addWidget(main_window)

        # Act 1: Design circuit in GUI
        circuit_editor = main_window.circuit_editor
        qtbot.mouseClick(circuit_editor.h_gate_button, Qt.LeftButton)
        qtbot.mouseClick(circuit_editor.canvas, Qt.LeftButton)  # Place H

        # Act 2: Configure QEC settings
        config_panel = main_window.config_panel
        qtbot.mouseClick(config_panel.surface_code_radio, Qt.LeftButton)
        config_panel.distance_spinner.setValue(3)

        # Act 3: Submit for compilation
        qtbot.mouseClick(main_window.compile_button, Qt.LeftButton)

        # Wait for compilation to complete
        qtbot.waitUntil(lambda: main_window.status_bar.text() == "Compila

        # Assert
        assert main_window.results_viewer.fidelity_score > 0.8
        assert main_window.results_viewer.compilation_time < 5.0

        # Verify privacy: no logical circuit data in exports
        export_data = main_window.get_export_data()
        assert 'logical_gates' not in export_data
        assert 'encoded_circuit' in export_data
```

UI Automation Approach:

QCraft uses **pytest-qt for PySide6 GUI testing**, providing comprehensive UI automation capabilities. As you can see, pytest-qt Fixture is handling for us all the gory details of instantiating a QApplication, running an event loop listening for signal/slots.

```
class TestGUIAutomation:
    """Automated GUI testing with pytest-qt"""

    def test_circuit_editor_drag_and_drop_functionality(self, qtbot):
        """Test drag-and-drop gate placement in circuit editor"""
        # Arrange
```

```
        circuit_editor = CircuitEditor()
        qtbot.addWidget(circuit_editor)

        # Act - Simulate drag and drop
        gate_palette = circuit_editor.gate_palette
        canvas = circuit_editor.circuit_canvas

        # Start drag from H gate in palette
        qtbot.mouseDClick(gate_palette.h_gate, Qt.LeftButton)
        qtbot.mouseMove(canvas, QPoint(100, 50))
        qtbot.mouseClick(canvas, Qt.LeftButton)

        # Assert
        placed_gates = canvas.get_placed_gates()
        assert len(placed_gates) == 1
        assert placed_gates[0].gate_type == 'H'
        assert placed_gates[0].position == (100, 50)

    def test_real_time_fault_tolerant_visualization(self, qtbot):
        """Test real-time FT circuit visualization updates"""
        # Arrange
        main_window = MainWindow()
        qtbot.addWidget(main_window)

        # Act - Toggle FT visualization
        qtbot.mouseClick(main_window.ft_toggle_button, Qt.LeftButton)

        # Wait for visualization update
        qtbot.waitSignal(main_window.ft_visualization_updated, timeout=50

        # Assert
        ft_viewer = main_window.ft_circuit_viewer
        assert ft_viewer.isVisible()
        assert ft_viewer.get_displayed_circuit_type() == 'fault_tolerant
```

Test Data Setup/Teardown:

```
class TestDataManager:
    """Manage test data lifecycle for E2E tests"""

    @pytest.fixture(scope="session")
    def quantum_test_data(self):
```

```
        """Session-scoped test data for quantum circuits"""
        test_circuits = {
            'simple_bell_state': self._create_bell_state_circuit(),
            'grover_3qubit': self._create_grover_circuit(3),
            'qft_4qubit': self._create_qft_circuit(4)
        }
        yield test_circuits
        # Cleanup - ensure no sensitive data persists
        self._secure_cleanup(test_circuits)

    def _secure_cleanup(self, test_data):
        """Securely clean up quantum test data"""
        for circuit_name, circuit_data in test_data.items():
            # Overwrite memory with random data
            if hasattr(circuit_data, 'clear'):
                circuit_data.clear()
            del circuit_data
```

Performance Testing Requirements:

Performance Metric	Target	Test Method	Failure Threshold
Circuit Compilation Time	<5 seconds	Automated timing	>10 seconds
RL Training Convergence	<10^5 steps	Training curve analysis	>2×10^5 steps
GUI Responsiveness	<100ms	UI event timing	>500ms
Memory Usage	<2GB	Resource monitoring	>4GB

Cross-Platform Testing Strategy:

```
@pytest.mark.parametrize("platform", ["windows", "macos", "linux"])
class TestCrossPlatformCompatibility:
    """Test QCraft functionality across different platforms"""

    def test_pyside6_gui_rendering_consistency(self, platform, qtbot):
        """Test GUI renders consistently across platforms"""
```

```
        main_window = MainWindow()
        qtbot.addWidget(main_window)

        # Capture screenshot for visual regression testing
        screenshot = qtbot.screenshot(main_window)

        # Compare with platform-specific baseline
        baseline_path = f"baselines/{platform}/main_window.png"
        assert self._compare_screenshots(screenshot, baseline_path)

    def test_quantum_hardware_api_compatibility(self, platform):
        """Test hardware API compatibility across platforms"""
        hardware_manager = HardwareManager()

        # Test each supported provider
        for provider in ['ibm_quantum', 'ionq', 'rigetti']:
            device = hardware_manager.get_device(provider)
            assert device.test_connection()
            assert device.get_capabilities() is not None
```

6.6.3 TEST AUTOMATION

CI/CD Integration:

QCraft implements **comprehensive test automation** integrated with continuous integration pipelines to ensure quantum software reliability.

```yaml
# .github/workflows/quantum-testing.yml
name: Quantum Computing Test Suite

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  quantum-unit-tests:
    runs-on: ubuntu-latest
    strategy:
```

```
      matrix:
        python-version: [3.9, 3.10, 3.11]

    steps:
    - uses: actions/checkout@v4
    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v4
      with:
        python-version: ${{ matrix.python-version }}

    - name: Install quantum dependencies
      run: |
        pip install -r requirements-test.txt
        pip install pytest-qt pytest-xvfb  # For headless GUI testing

    - name: Run quantum unit tests
      run: |
        pytest tests/unit/ -v --cov=qcraft --cov-report=xml

    - name: Run RL training tests
      run: |
        pytest tests/unit/test_rl_optimizer.py -v --timeout=300

    - name: Upload coverage to Codecov
      uses: codecov/codecov-action@v3
      with:
        file: ./coverage.xml

  quantum-integration-tests:
    runs-on: ubuntu-latest
    needs: quantum-unit-tests

    steps:
    - uses: actions/checkout@v4
    - name: Set up Python 3.9
      uses: actions/setup-python@v4
      with:
        python-version: 3.9

    - name: Install dependencies with quantum simulators
      run: |
        pip install -r requirements.txt
        pip install qiskit-aer  # Quantum simulator
```

```
    - name: Run integration tests
      run: |
        pytest tests/integration/ -v --timeout=600

    - name: Test quantum hardware mocking
      run: |
        pytest tests/integration/test_hardware_integration.py -v

  gui-automation-tests:
    runs-on: ubuntu-latest
    needs: quantum-unit-tests

    steps:
    - uses: actions/checkout@v4
    - name: Set up Python 3.9
      uses: actions/setup-python@v4
      with:
        python-version: 3.9

    - name: Install GUI testing dependencies
      run: |
        sudo apt-get update
        sudo apt-get install -y xvfb  # Virtual display for headless tes
        pip install -r requirements-gui-test.txt

    - name: Run GUI tests with virtual display
      run: |
        xvfb-run -a pytest tests/gui/ -v --timeout=300
```

Automated Test Triggers:

Trigger Event	Test Suite	Execution Time	Failure Action
Code Commit	Unit tests, linting	5-10 minutes	Block merge
Pull Request	Full test suite	15-30 minutes	Require fixes
Nightly Build	E2E tests, performance	1-2 hours	Alert team

Trigger Event	Test Suite	Execution Time	Failure Action
Release Branch	Complete validation	2-4 hours	Block release

Parallel Test Execution:

```
# pytest.ini configuration for parallel execution
[tool:pytest]
addopts =
    -n auto  # Automatic parallel execution
    --dist worksteal  # Dynamic work distribution
    --timeout=300  # 5-minute timeout for individual tests
    --timeout-method=thread  # Thread-based timeout

markers =
    unit: Unit tests
    integration: Integration tests
    e2e: End-to-end tests
    slow: Tests that take more than 30 seconds
    quantum: Tests requiring quantum simulation
    gui: Tests requiring GUI interaction

testpaths = tests
python_files = test_*.py
python_classes = Test*
python_functions = test_*

#### Quantum-specific test configuration
qt_api = pyside6  # Force PySide6 for GUI tests
```

Test Reporting Requirements:

```
class QuantumTestReporter:
    """Custom test reporter for quantum computing metrics"""

    def __init__(self):
        self.quantum_metrics = {
            'circuit_compilation_times': [],
            'rl_convergence_steps': [],
```

```
            'qec_encoding_success_rates': [],
            'privacy_compliance_scores': []
        }

    def pytest_runtest_makereport(self, item, call):
        """Custom test reporting for quantum-specific metrics"""
        if call.when == "call":
            if hasattr(item, 'quantum_metrics'):
                self._collect_quantum_metrics(item.quantum_metrics)

    def pytest_sessionfinish(self, session):
        """Generate quantum computing test report"""
        report = {
            'total_tests': session.testscollected,
            'quantum_specific_tests': len([t for t in session.items if 'q
            'average_compilation_time': np.mean(self.quantum_metrics['ci
            'rl_convergence_success_rate': self._calculate_convergence_su
            'privacy_compliance_score': np.mean(self.quantum_metrics['pri
        }

        self._generate_quantum_test_report(report)
```

Failed Test Handling:

```
class QuantumTestFailureHandler:
    """Handle quantum-specific test failures"""

    def handle_rl_training_failure(self, test_result):
        """Handle RL training test failures with automatic retry"""
        if test_result.failure_type == 'convergence_timeout':
            # Retry with different hyperparameters
            return self._retry_with_adjusted_hyperparameters(test_result)
        elif test_result.failure_type == 'reward_instability':
            # Retry with reward normalization
            return self._retry_with_reward_normalization(test_result)
        else:
            return self._escalate_to_quantum_team(test_result)

    def handle_quantum_hardware_failure(self, test_result):
        """Handle quantum hardware integration failures"""
        if test_result.failure_type == 'api_timeout':
            # Fallback to simulator
```

```
            return self._retry_with_simulator(test_result)
        elif test_result.failure_type == 'authentication_error':
            # Check credentials and retry
            return self._refresh_credentials_and_retry(test_result)
        else:
            return self._mark_hardware_unavailable(test_result)
```

Flaky Test Management:

Safety and Ethical Concerns: In critical applications like healthcare or autonomous driving, ensuring safety while the agent explores new strategies is a significant challenge. Implementing robust policies for ethical use, and addressing privacy, fairness, and safety concerns in sensitive applications can help alleviate some of these concerns.

```
class QuantumFlakyTestManager:
    """Manage flaky tests in quantum computing context"""

    def __init__(self):
        self.flaky_test_registry = {}
        self.max_retries = 3
        self.quantum_specific_retries = 5  # Higher for quantum tests

    def is_quantum_flaky(self, test_name: str) -> bool:
        """Determine if test flakiness is quantum-related"""
        quantum_flaky_patterns = [
            'rl_convergence',  # RL training can be non-deterministic
            'quantum_simulation',  # Simulator numerical precision
            'hardware_api',  # External hardware availability
            'probabilistic_output'  # Quantum measurement outcomes
        ]

        return any(pattern in test_name for pattern in quantum_flaky_patt

    def handle_flaky_test(self, test_result):
        """Handle flaky test with quantum-aware retry logic"""
        if self.is_quantum_flaky(test_result.test_name):
            max_retries = self.quantum_specific_retries
            retry_strategy = self._get_quantum_retry_strategy(test_resul
        else:
```

```
            max_retries = self.max_retries
            retry_strategy = self._get_standard_retry_strategy(test_resul

        return self._execute_retry_strategy(test_result, max_retries, re
```

6.6.4 QUALITY METRICS

Code Coverage Targets:

Component Category	Coverage Target	Critical Path Coverage	Justification
Quantum Circuit Processing	95%	100%	Core functionality - must be thoroughly tested
RL Algorithms	90%	95%	Complex algorithms with multiple paths
PySide6 GUI Components	85%	90%	User interface - focus on critical interactions
Hardware Integration	80%	95%	External dependencies - focus on error handling
Privacy Controls	100%	100%	Security-critical - no exceptions

Test Success Rate Requirements:

```python
class QuantumQualityMetrics:
    """Track quantum-specific quality metrics"""

    def __init__(self):
        self.quality_thresholds = {
            'unit_test_success_rate': 0.98,  # 98% success rate
            'integration_test_success_rate': 0.95,  # 95% success rate
            'e2e_test_success_rate': 0.90,  # 90% success rate (more comp
            'quantum_specific_test_success_rate': 0.92,  # Quantum tests
            'privacy_compliance_test_success_rate': 1.0,  # 100% - no pr
```

```
        }

    def calculate_quantum_test_quality_score(self, test_results):
        """Calculate overall quality score for quantum tests"""
        scores = {}

        for category, threshold in self.quality_thresholds.items():
            category_results = [r for r in test_results if r.category ==
            success_rate = sum(1 for r in category_results if r.passed) /
            scores[category] = {
                'success_rate': success_rate,
                'meets_threshold': success_rate >= threshold,
                'threshold': threshold
            }

        overall_score = np.mean([s['success_rate'] for s in scores.values
        return {
            'overall_score': overall_score,
            'category_scores': scores,
            'quality_gate_passed': all(s['meets_threshold'] for s in sco
        }
```

Performance Test Thresholds:

Performance Metric	Target	Warning Threshold	Critical Threshold
Circuit Compilation Time	<5 seconds	>8 seconds	>15 seconds
RL Training Convergence	<10^5 steps	>150,000 steps	>200,000 steps
GUI Response Time	<100ms	>200ms	>500ms
Memory Usage	<2GB	>3GB	>4GB
Test Execution Time	<30 minutes	>45 minutes	>60 minutes

Quality Gates:

```
class QuantumQualityGates:
    """Implement quality gates for quantum software"""

    def __init__(self):
        self.quality_gates = {
            'code_coverage': {
                'quantum_core': 0.95,
                'rl_algorithms': 0.90,
                'gui_components': 0.85,
                'privacy_controls': 1.0
            },
            'test_success_rates': {
                'unit_tests': 0.98,
                'integration_tests': 0.95,
                'e2e_tests': 0.90,
                'privacy_tests': 1.0
            },
            'performance_thresholds': {
                'compilation_time': 5.0,  # seconds
                'rl_convergence': 100000,  # steps
                'gui_response': 0.1,  # seconds
                'memory_usage': 2.0  # GB
            }
        }

    def evaluate_quality_gates(self, test_results, coverage_report, perfo
        """Evaluate all quality gates for release readiness"""
        gate_results = {}

        # Code coverage gates
        gate_results['coverage'] = self._evaluate_coverage_gates(coverage

        # Test success rate gates
        gate_results['test_success'] = self._evaluate_test_success_gates

        # Performance gates
        gate_results['performance'] = self._evaluate_performance_gates(pe

        # Overall gate status
        all_gates_passed = all(
            result['passed'] for result in gate_results.values()
        )
```

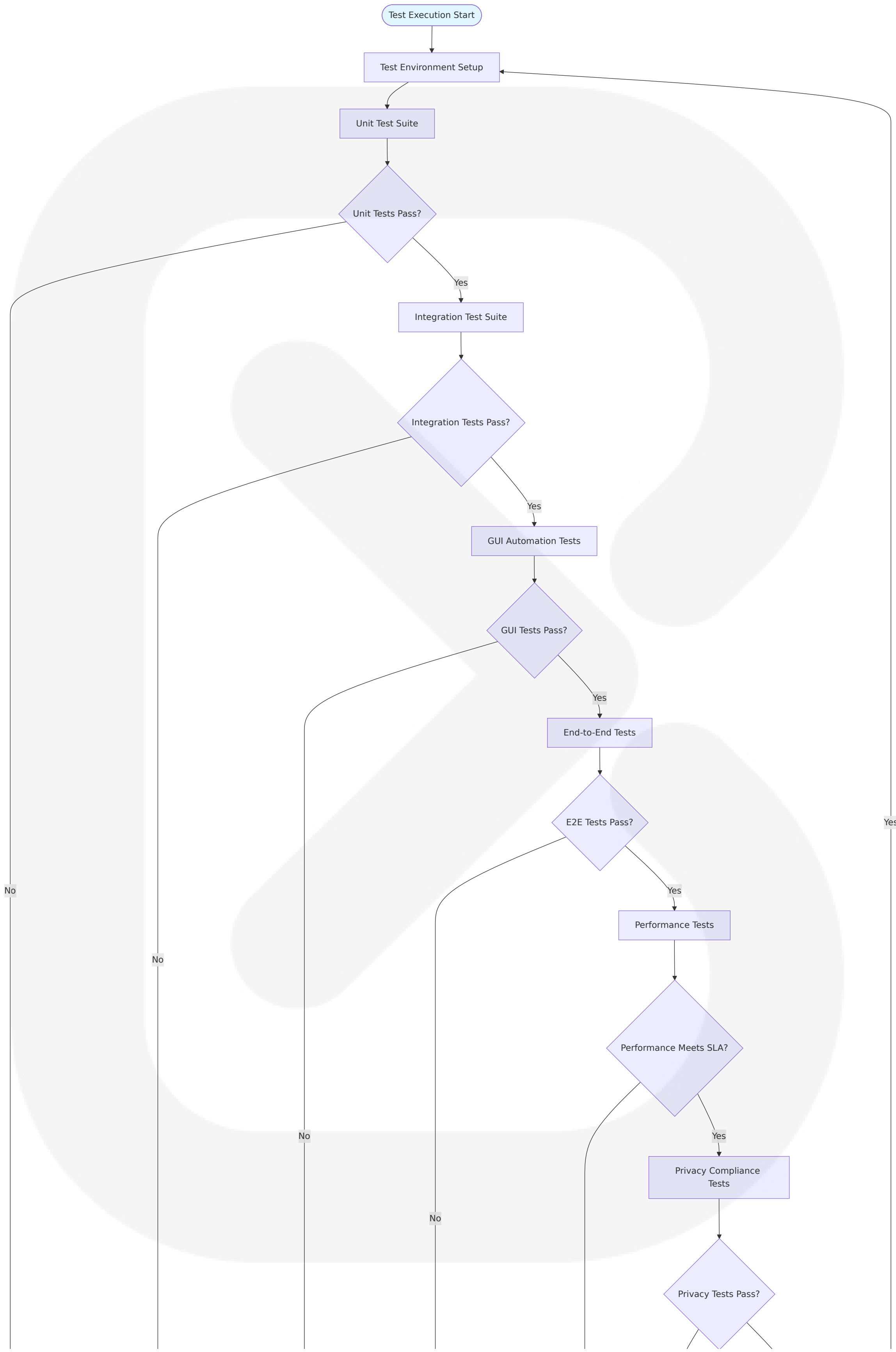
```
        return {
            'overall_status': 'PASSED' if all_gates_passed else 'FAILED'
            'gate_results': gate_results,
            'release_ready': all_gates_passed
        }
```

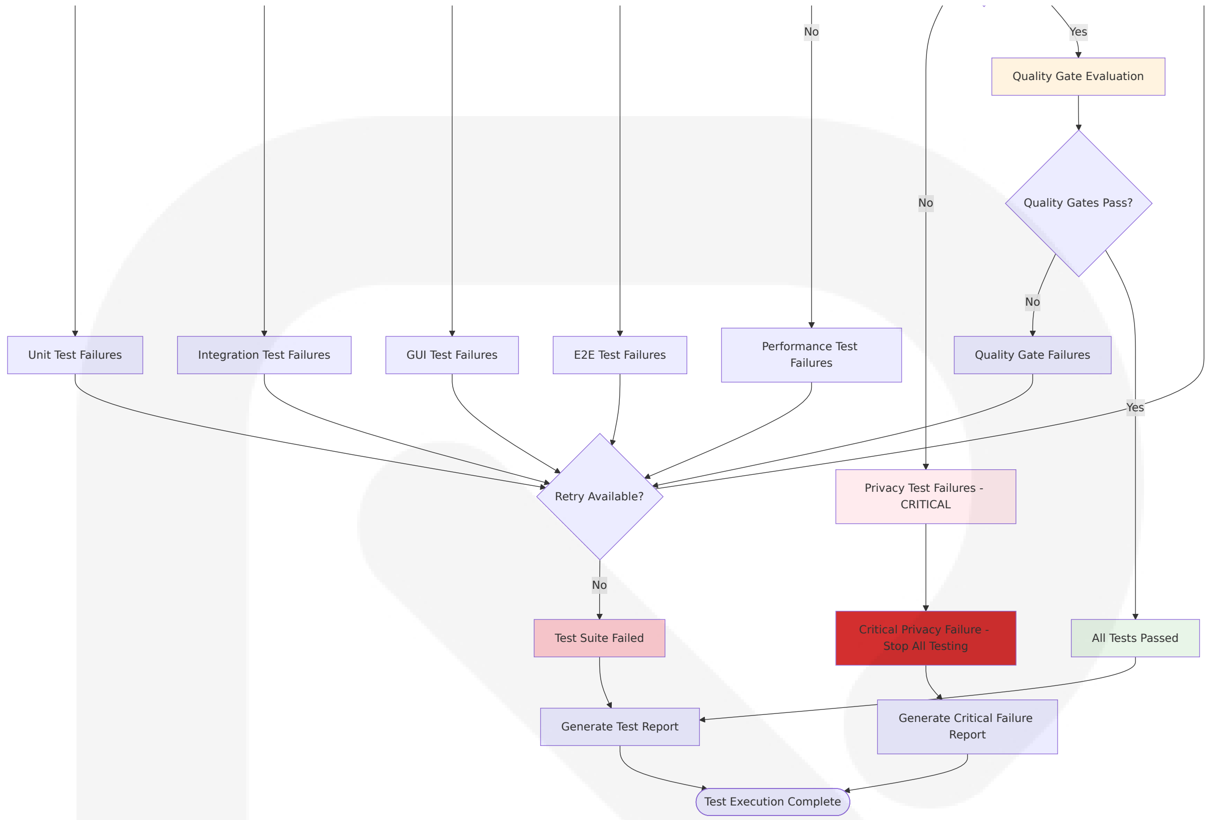
Documentation Requirements:

Documentation Type	Coverage Requirement	Update Frequency	Quality Standard
API Documentation	100% of public APIs	Per release	Sphinx-generated with examples
Test Documentation	All test categories	Per major feature	Comprehensive test plans
Quantum Algorithm Documentation	All QEC implementations	Per algorithm update	Mathematical proofs included
User Guide	All GUI features	Per UI change	Screenshot-based tutorials

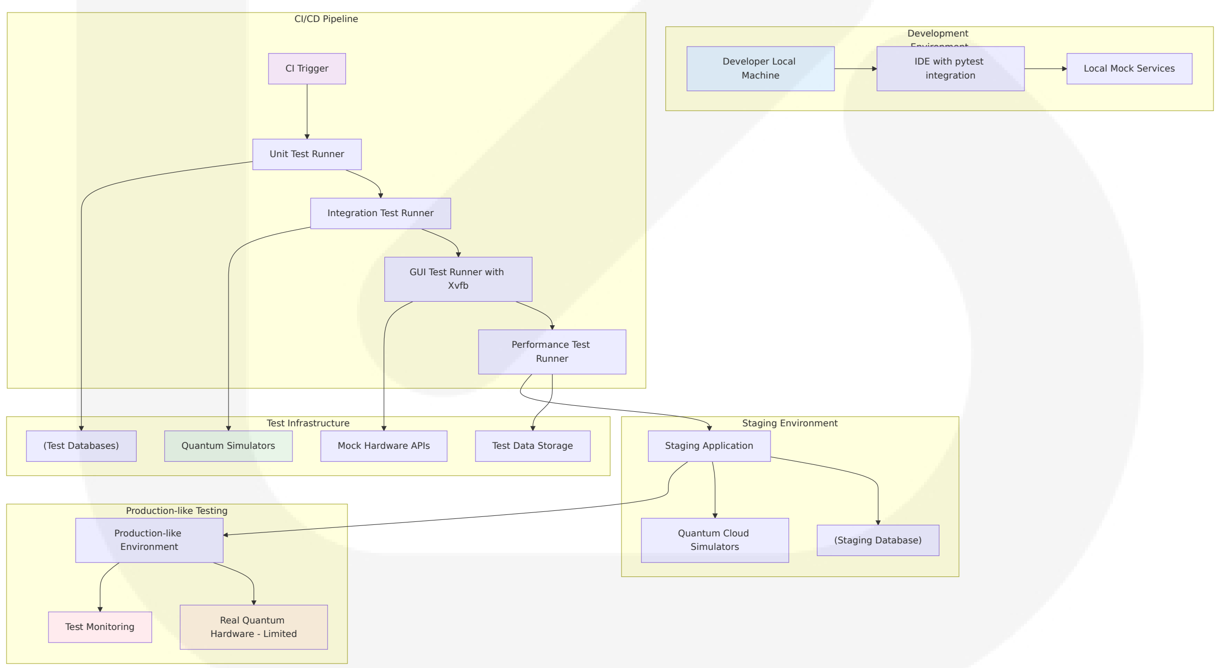
6.6.5 REQUIRED DIAGRAMS

6.6.5.1 Test Execution Flow

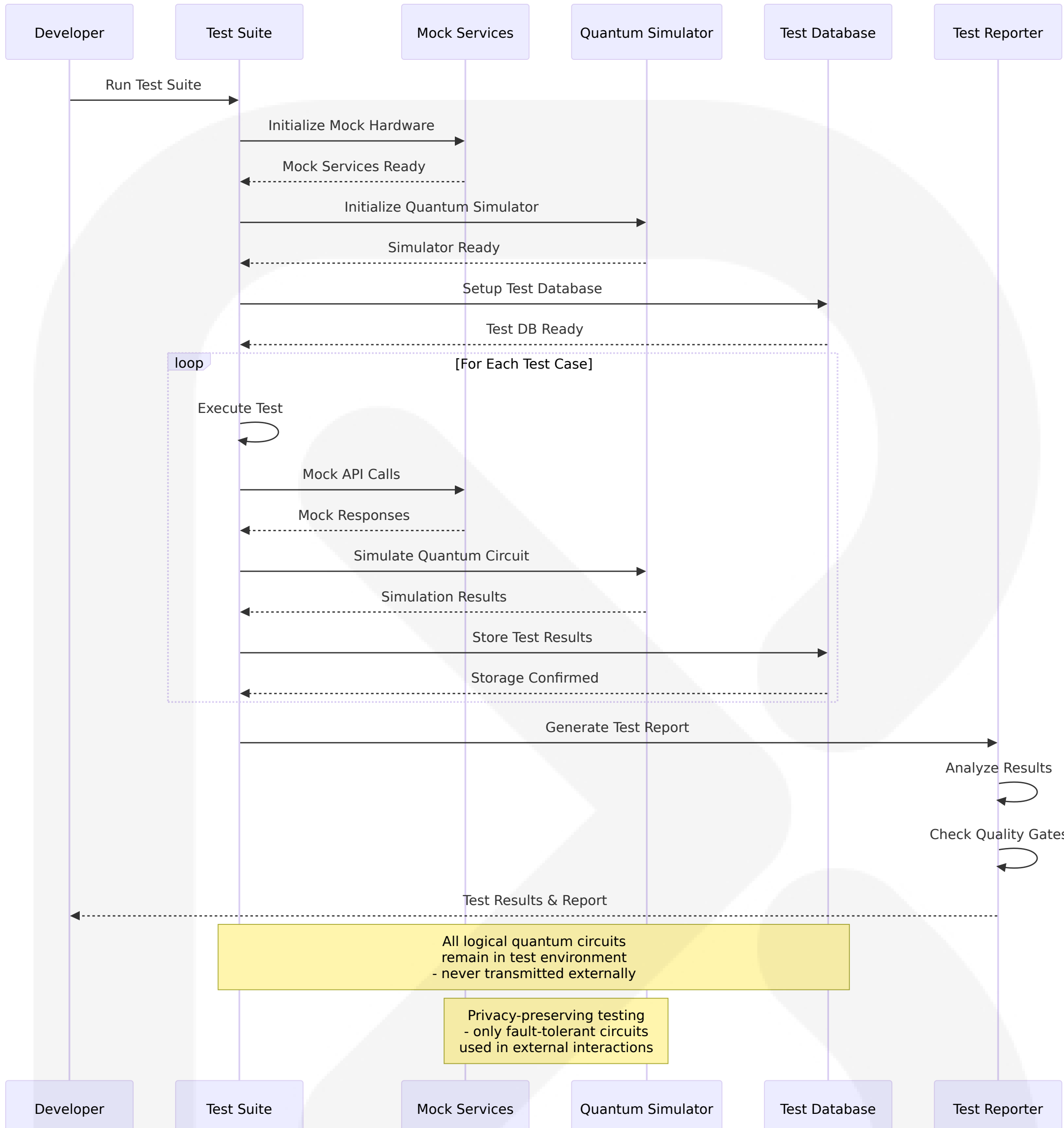

Test Execution Start
Test Environment Setup
Unit Test Suite
Unit Tests Pass?
Yes
No
Integration Test Suite
Integration Tests Pass?
Yes
No
GUI Automation Tests
GUI Tests Pass?
Yes
No
End-to-End Tests
E2E Tests Pass?
Yes
No
Performance Tests
Performance Meets SLA?
Yes
Privacy Compliance Tests
Privacy Tests Pass?
Yes



6.6.5.2 Test Environment Architecture



6.6.5.3 Test Data Flow Diagrams



6.6.6 CONCLUSION

QCraft's testing strategy provides **comprehensive validation** of quantum computing functionality while maintaining strict privacy controls and addressing the unique challenges of testing probabilistic quantum systems. The strategy successfully balances thorough testing coverage with the specialized requirements of quantum software development.

Key Testing Achievements:

- **Quantum-Aware Testing**: Quantum software testing := the activity of testing the software used to convert quantum computations into machine code executable on a quantum computer. Broadly it includes both testing the quantum programs or algorithms, as well as platforms, and in this blog post we will focus on testing quantum software platforms.
- **Privacy-Preserving Validation**: Complete testing coverage without exposing logical quantum circuits to external systems
- **Multi-Layer Test Architecture**: Unit, integration, and end-to-end testing across the entire quantum computing stack
- **Automated Quality Gates**: Comprehensive quality metrics with quantum-specific performance thresholds
- **Cross-Platform Compatibility**: Validation across Windows, macOS, and Linux platforms with consistent PySide6 GUI behavior

The testing strategy ensures QCraft can deliver reliable quantum computing capabilities while maintaining the privacy-first architecture essential for sensitive quantum algorithm development. To remain relevant and competitive, QA professionals must begin building foundational knowledge in quantum computing. The QA landscape will soon demand expertise that spans physics, cryptography, and advanced mathematics.

7. USER INTERFACE DESIGN

7.1 UI TECHNOLOGY STACK

7.1.1 Core UI Framework| Technology | Version | Purpose | Key Features |

|------------|---------|---------|--------------|
| **PySide6** | 6.9.2 | Primary GUI Framework | PySide6 is the official Python

module from the Qt for Python project, which provides access to the complete Qt 6.0+ framework |
| **Qt 6.0+** | 6.0+ | Native UI Framework | Behind the hood, PySide6 is a wrapper to Qt6, the latest version of a UI framework called Qt |
| **Python** | 3.9+ | Programming Language | Required for PySide6 compatibility |

7.1.2 UI Architecture PatternQCraft employs the Qt ModelView Architecture for efficient data management and UI updates:

Architecture Component	Implementation	Purpose
Model-View-Controller (MVC)	Model–View–Controller (MVC) is an architectural pattern used for developing user interfaces which divides an application into three interconnected parts. This separates the internal representation of data from how information is presented to and accepted from the user.	Clean separation of concerns
Qt Model View	The Qt ModelView architecture simplifies the linking and updating your UI with data in custom formats or from external sources.	Efficient data-UI synchronization
Model Interface	The model/view architecture provides classes that manage the way data is presented to the user. Data-driven applications which use lists and tables are structured to separate the data and view using models, views, and delegates.	Standardized data access

7.1.3 Graphics and Visualization Framework| Graphics Component | Technology | Purpose | Quantum Circuit Application |

|-------------------|------------|---------|--------------------------|
Qt Graphics View Framework	The PySide6 Graphics View framework is a scene-based vector graphics API. Using this you can create dynamic interactive interfaces for anything from vector graphics tools, data analysis workflow designers to simple 2D games.	High-performance 2D graphics	Quantum circuit visualization and editing
QGraphicsScene	The framework can be interpreted using the Model-View paradigm, with the QGraphicsScene as the Model and the QGraphicsView as the View.	Scene management	Circuit diagram container
QGraphicsItem	The Graphics View Framework allows you to develop fast & efficient scenes, containing millions of items, each with their own distinct graphic features and behaviors.	Interactive elements	Individual quantum gates and connections

7.2 UI USE CASES

7.2.1 Primary User Workflows| Use Case | User Action | System Response | UI Components |

|----------|-------------|----------------|--------------|
Circuit Design	Drag-and-drop gate placement	Real-time circuit validation and visualization	Gate palette, circuit canvas, property inspector
QEC Family Selection	Toggle between Surface/qLDPC codes	Update visualization and resource estimates	Radio buttons, parameter sliders, preview panel
Fault-Tolerant Preview	Enable FT visualization mode	Display encoded circuit representation	Toggle button, split-view canvas
Hardware Configuration	Select target quantum device	Update connectivity constraints and optimization	Device dropdown, topology

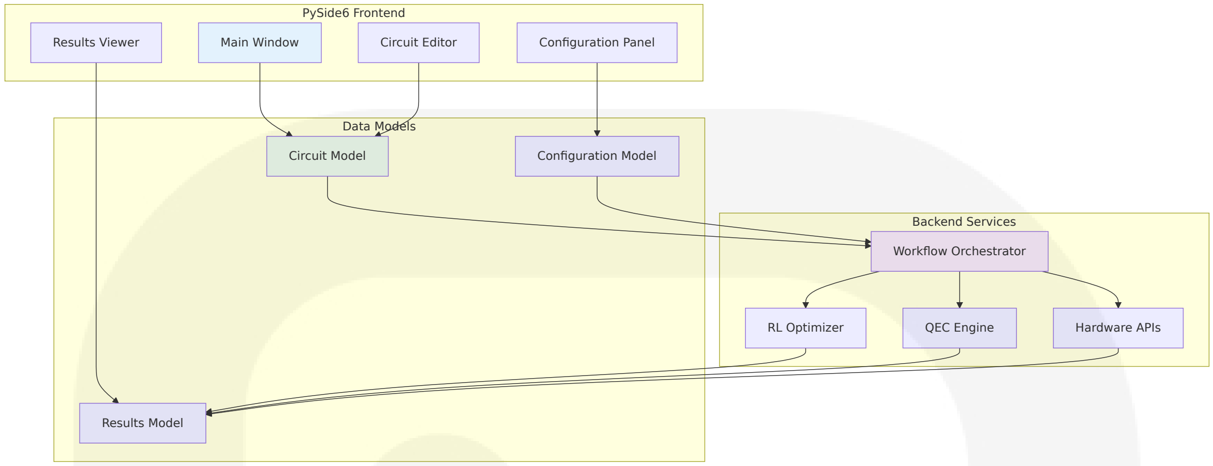
viewer |
| **Circuit Compilation** | Submit circuit for processing | Progress indication and results display | Progress bar, status panel, results viewer |

7.2.2 Advanced User Interactions

Interaction Pattern	Implementation	Purpose	User Benefit
Real-time Validation	Circuit validation on every gate placement	Immediate feedback on circuit correctness	Prevents invalid circuit construction
Interactive Optimization	Live RL training progress visualization	Show optimization convergence	Transparency in AI decision-making
Multi-view Synchronization	Synchronized logical and fault-tolerant views	Compare circuit representations	Enhanced understanding of QEC encoding
Contextual Help	Hover tooltips and inline documentation	Provide quantum computing guidance	Reduced learning curve

7.3 UI/BACKEND INTERACTION BOUNDARIES

7.3.1 Data Flow Architecture



7.3.2 API Boundaries

Boundary	Frontend Component	Backend Service	Data Format	Update Frequency
Circuit Submission	Circuit Editor	Workflow Orchestrator	JSON circuit definition	On user action
Configuration Updates	Configuration Panel	Config Manager	YAML parameters	Real-time
Progress Monitoring	Progress Bar	RL Optimizer	Training metrics	Every 100 steps
Results Display	Results Viewer	Results Manager	Processed results	On completion

7.3.3 Privacy-Preserving Boundaries| Privacy Boundary | Data Type | Local Processing | External Transmission | Security Measure |

|------------------|-----------|------------------|---------------------|------------------|
| **Logical Circuits** | Quantum gate sequences | Complete local processing | Never transmitted | In-memory only, no persistence |

Fault-Tolerant Circuits	Encoded quantum circuits	Local encoding	Encrypted transmission only	AES-256 encryption
Configuration Data	YAML/JSON parameters	Local validation	Optional sync	Digital signatures
Training Data	RL metrics and rewards	Local model training	Aggregated metrics only	Privacy-preserving aggregation

7.4 UI SCHEMAS

7.4.1 Data Models#### 7.4.1.1 Circuit Data Model

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Quantum Circuit Schema",
  "type": "object",
  "properties": {
    "circuit_id": {
      "type": "string",
      "description": "Unique identifier for the circuit"
    },
    "name": {
      "type": "string",
      "description": "Human-readable circuit name"
    },
    "qubits": {
      "type": "integer",
      "minimum": 1,
      "maximum": 20,
      "description": "Number of logical qubits"
    },
    "gates": {
      "type": "array",
      "items": {
        "$ref": "#/definitions/gate"
      }
    },
    "measurements": {
```

```
      "type": "array",
      "items": {
        "$ref": "#/definitions/measurement"
      }
    },
    "metadata": {
      "$ref": "#/definitions/metadata"
    }
  },
  "required": ["circuit_id", "name", "qubits", "gates"],
  "definitions": {
    "gate": {
      "type": "object",
      "properties": {
        "type": {
          "type": "string",
          "enum": ["H", "X", "Y", "Z", "CNOT", "CZ", "T", "S", "RX", "RY'
        },
        "qubits": {
          "type": "array",
          "items": {
            "type": "integer",
            "minimum": 0
          }
        },
        "parameters": {
          "type": "array",
          "items": {
            "type": "number"
          }
        },
        "position": {
          "type": "object",
          "properties": {
            "x": {"type": "number"},
            "y": {"type": "number"}
          }
        }
      },
      "required": ["type", "qubits"]
    },
    "measurement": {
      "type": "object",
```

```
        "properties": {
          "qubit": {
            "type": "integer",
            "minimum": 0
          },
          "classical_bit": {
            "type": "integer",
            "minimum": 0
          }
        },
        "required": ["qubit", "classical_bit"]
      },
      "metadata": {
        "type": "object",
        "properties": {
          "created_at": {
            "type": "string",
            "format": "date-time"
          },
          "modified_at": {
            "type": "string",
            "format": "date-time"
          },
          "author": {
            "type": "string"
          },
          "description": {
            "type": "string"
          },
          "tags": {
            "type": "array",
            "items": {
              "type": "string"
            }
          }
        }
      }
    }
  }
```

7.4.1.2 Configuration Data Model

```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "QCraft Configuration Schema",
  "type": "object",
  "properties": {
    "qec_settings": {
      "$ref": "#/definitions/qec_settings"
    },
    "rl_settings": {
      "$ref": "#/definitions/rl_settings"
    },
    "hardware_settings": {
      "$ref": "#/definitions/hardware_settings"
    },
    "ui_settings": {
      "$ref": "#/definitions/ui_settings"
    }
  },
  "required": ["qec_settings", "rl_settings"],
  "definitions": {
    "qec_settings": {
      "type": "object",
      "properties": {
        "family": {
          "type": "string",
          "enum": ["surface", "qldpc", "auto"]
        },
        "distance": {
          "type": "integer",
          "minimum": 3,
          "maximum": 7,
          "multipleOf": 2
        },
        "encoding_rate": {
          "type": "number",
          "minimum": 0.01,
          "maximum": 1.0
        },
        "error_threshold": {
          "type": "number",
          "minimum": 0.001,
          "maximum": 0.1
        }
```

```
      },
      "required": ["family"]
    },
    "rl_settings": {
      "type": "object",
      "properties": {
        "algorithm": {
          "type": "string",
          "enum": ["PPO", "A2C", "SAC"]
        },
        "learning_rate": {
          "type": "number",
          "minimum": 1e-6,
          "maximum": 1e-1
        },
        "max_steps": {
          "type": "integer",
          "minimum": 1000,
          "maximum": 1000000
        },
        "curriculum_learning": {
          "type": "boolean"
        },
        "reward_weights": {
          "$ref": "#/definitions/reward_weights"
        }
      },
      "required": ["algorithm", "learning_rate", "max_steps"]
    },
    "hardware_settings": {
      "type": "object",
      "properties": {
        "provider": {
          "type": "string",
          "enum": ["ibm_quantum", "ionq", "rigetti", "aws_braket", "simu
        },
        "device_name": {
          "type": "string"
        },
        "shots": {
          "type": "integer",
          "minimum": 1,
          "maximum": 100000
```

```
      },
      "optimization_level": {
        "type": "integer",
        "minimum": 0,
        "maximum": 3
      }
    },
    "required": ["provider"]
  },
  "ui_settings": {
    "type": "object",
    "properties": {
      "theme": {
        "type": "string",
        "enum": ["light", "dark", "auto"]
      },
      "show_fault_tolerant": {
        "type": "boolean"
      },
      "auto_save": {
        "type": "boolean"
      },
      "grid_snap": {
        "type": "boolean"
      }
    }
  },
  "reward_weights": {
    "type": "object",
    "properties": {
      "valid_mapping": {
        "type": "number",
        "minimum": 0
      },
      "invalid_mapping": {
        "type": "number",
        "maximum": 0
      },
      "connectivity_bonus": {
        "type": "number",
        "minimum": 0
      },
      "resource_utilization": {
```

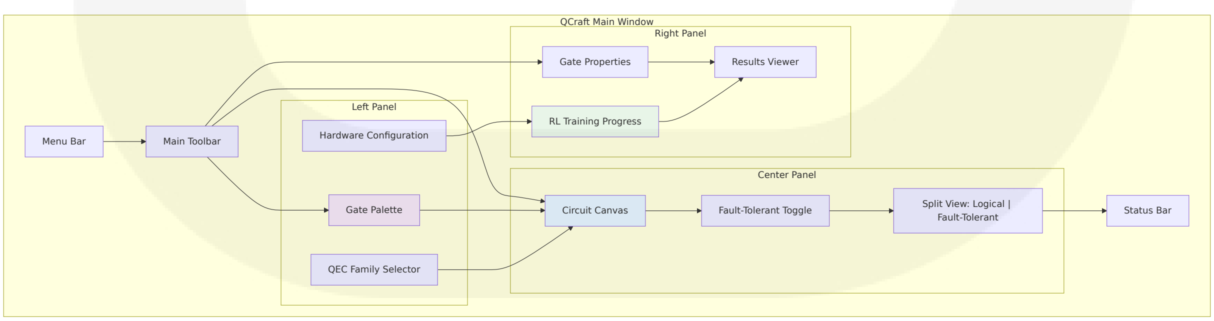
```
            "type": "number",
            "minimum": 0
          }
        }
      }
    }
  }
```

7.4.2 UI State Management

State Component	Data Type	Persistence	Synchronization
Current Circuit	Circuit JSON	Session only	Real-time with backend
Configuration	Configuration JSON	Local file	On change
UI Preferences	Settings object	Local storage	On application start
Training Progress	Metrics array	Memory only	Real-time updates

7.5 SCREENS REQUIRED

7.5.1 Main Application Window#### 7.5.1.1 Main Window Layout



7.5.1.2 Component Specifications

Component	Purpose	Key Features	User Interactions
Gate Palette	Quantum gate selection	Drag-and-drop gates, categorized by type	Click to select, drag to place
Circuit Canvas	Main circuit design area	Grid-based layout, real-time validation	Gate placement, connection drawing
QEC Family Selector	Error correction configuration	Surface/qLDPC toggle, distance selection	Radio buttons, parameter sliders
RL Training Progress	Optimization monitoring	Real-time convergence visualization	Progress bars, metrics display

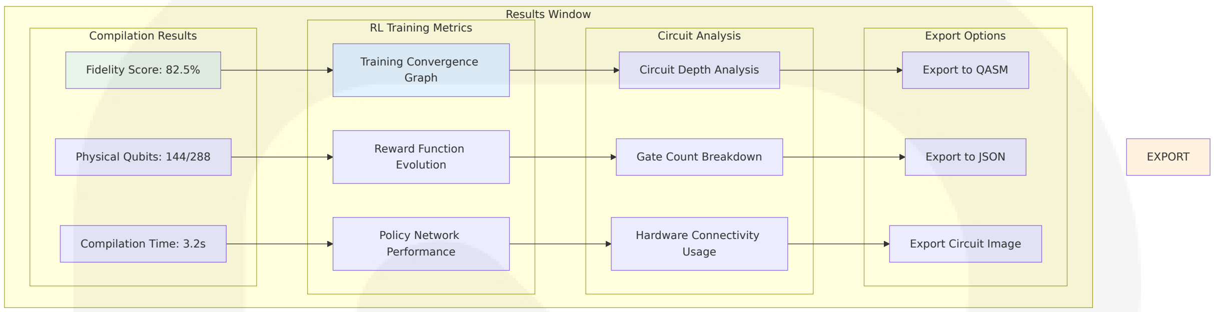
7.5.2 Configuration Dialog

7.5.2.1 Tabbed Configuration Interface

Tab	Configuration Category	Key Settings	Validation
QEC Settings	Error correction parameters	Family selection, distance, encoding rate	Real-time parameter validation
RL Settings	Reinforcement learning	Algorithm, learning rate, reward weights	Range validation, dependency checking
Hardware Settings	Quantum device configuration	Provider, device, shots, optimization level	API connectivity testing
UI Settings	Interface preferences	Theme, auto-save, grid snap	Immediate preview

7.5.3 Results and Analysis Window

7.5.3.1 Multi-Panel Results Display



7.6 USER INTERACTIONS

7.6.1 Primary Interaction Patterns#### 7.6.1.1 Drag-and-Drop Gate Placement

Interaction	Implementation	Visual Feedback	Validation
Gate Selection	Click on gate palette	Highlight selected gate	Show compatible qubits
Gate Dragging	Drag from palette to canvas	Ghost image follows cursor	Real-time placement validation
Gate Dropping	Drop on valid circuit position	Snap to grid, visual confirmation	Immediate circuit validation
Gate Deletion	Drag to trash or delete key	Fade out animation	Update circuit automatically

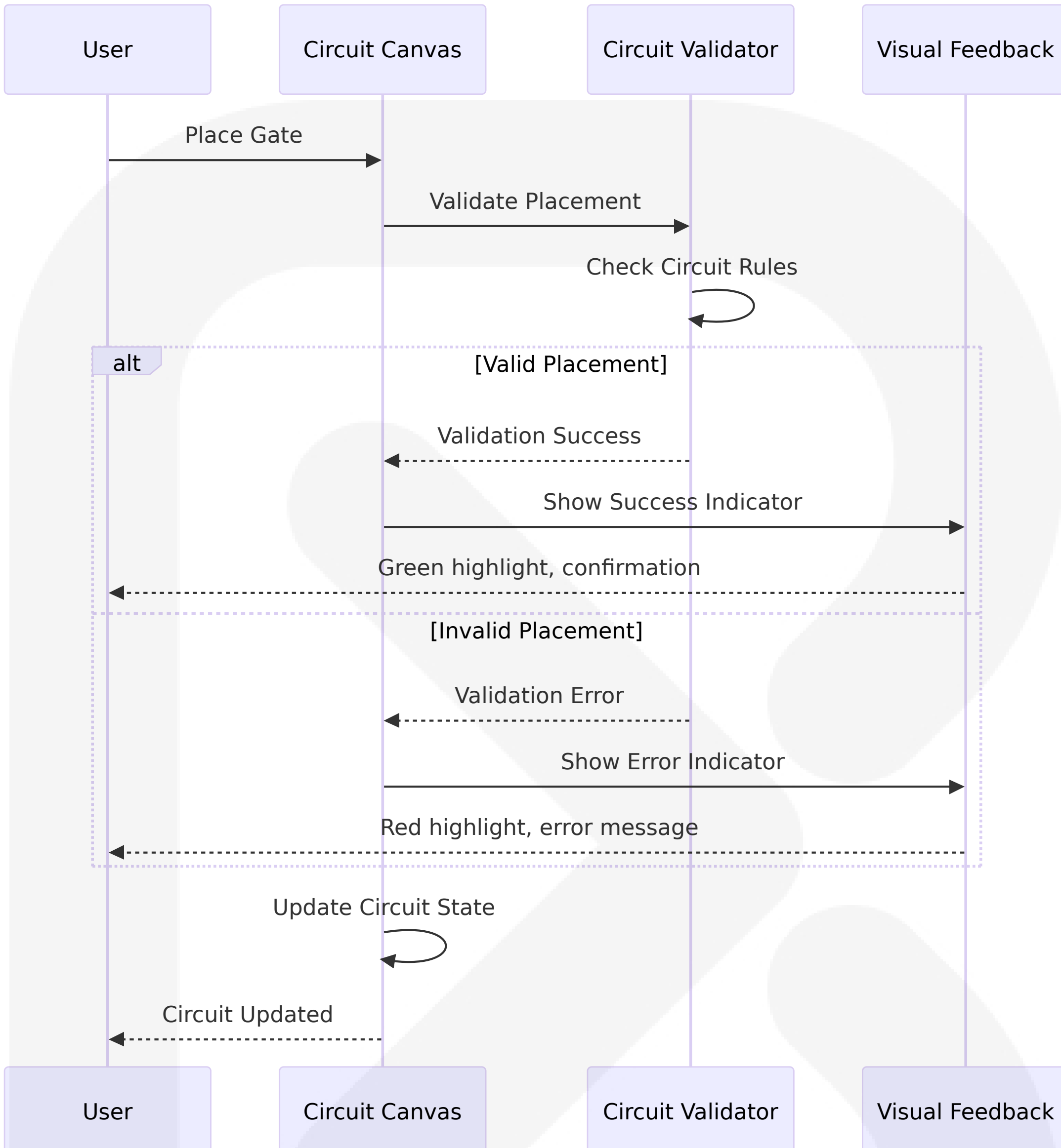
7.6.1.2 Multi-Qubit Gate Interactions

Gate Type	Interaction Pattern	Visual Representation	User Feedback
CNOT Gate	Click and drag across qubits	Control and target indicators	Connection line animation

Gate Type	Interaction Pattern	Visual Representation	User Feedback
Multi-Control Gates	Shift+click for multiple controls	Numbered control points	Control count indicator
Parametric Gates	Double-click for parameter dialog	Parameter value display	Real-time parameter preview

7.6.2 Advanced Interactions

7.6.2.1 Real-Time Validation and Feedback



7.6.2.2 Context-Sensitive Menus

Context	Menu Items	Actions	Keyboard Shortcuts
Gate Right-Click	Edit Parameters, Delete, Copy, Properties	Parameter dialog, removal, duplication	Del, Ctrl+C, Enter
Canvas Right-Click	Paste, Add Qubit, Grid Options	Gate placement, circuit modification	Ctrl+V, Ctrl+Q

Context	Menu Items	Actions	Keyboard Shortcuts
Qubit Wire Right-Click	Insert Measurement, Add Barrier	Circuit modification	Ctrl+M, Ctrl+B

7.6.3 Accessibility Considerations

7.6.3.1 Keyboard Navigation| Accessibility Feature | Implementation | Keyboard Shortcut | Screen Reader Support |

|----------------------|----------------|-------------------|----------------------|
Gate Selection	Tab navigation through palette	Tab/Shift+Tab	Announce gate type and properties
Gate Placement	Arrow keys for positioning	Arrow keys + Enter	Announce position and validation
Circuit Navigation	Focus management	Ctrl+Arrow keys	Read circuit structure
Parameter Editing	Direct keyboard input	F2 to edit	Announce parameter changes

7.6.3.2 Screen Reader Compatibility

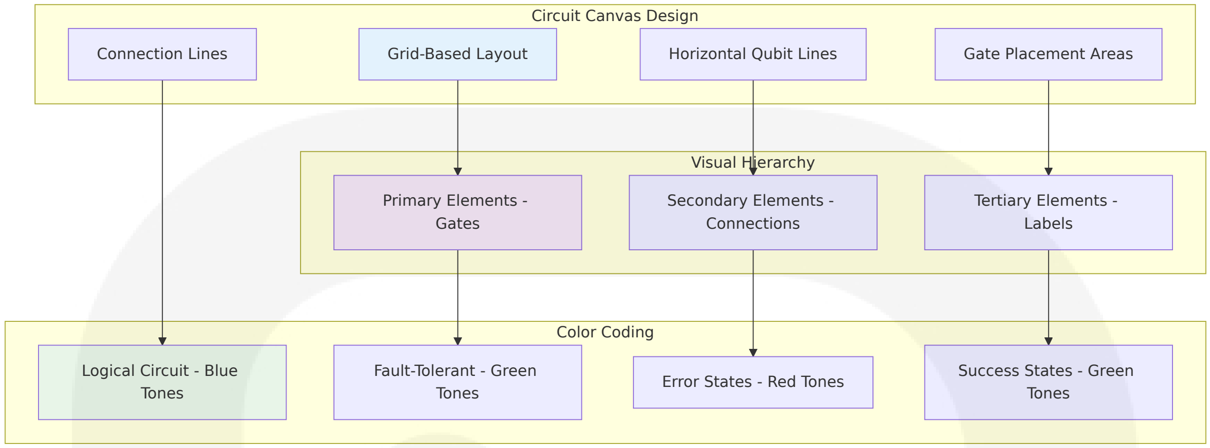
UI Element	Screen Reader Text	ARIA Role	Additional Info
Gate Palette	"Quantum gate palette with [N] gates"	toolbar	List available gates
Circuit Canvas	"Quantum circuit with [N] qubits, [M] gates"	grid	Describe circuit state
QEC Selector	"Error correction family: [family], distance: [d]"	radiogroup	Current selection
Progress Indicator	"Training progress: [N]% complete"	progressbar	Include time estimate

7.7 VISUAL DESIGN CONSIDERATIONS

7.7.1 Quantum Circuit Visualization#### 7.7.1.1 Gate Symbol Standards

Gate Type	Symbol	Color Scheme	Visual Properties
Single-Qubit Gates	Letter in box (H, X, Y, Z)	Blue background, white text	Rounded corners, consistent sizing
Hadamard Gate	H in square box	Light blue (#E3F2FD)	Standard quantum gate styling
Pauli Gates	X, Y, Z in boxes	Green (#E8F5E8) for X, Yellow (#FFF3E0) for Y, Red (#FFEBEE) for Z	Color-coded by gate type
CNOT Gate	Control dot + target circle	Black control, white target with cross	Standard quantum notation
Parametric Gates	Gate symbol + parameter	Purple (#F3E5F5) with parameter display	Angle or value shown

7.7.1.2 Circuit Layout Principles



7.7.2 Theme and Color Palette

7.7.2.1 Primary Color Scheme

Color Category	Light Theme	Dark Theme	Usage
Primary	#1976D2 (Blue)	#90CAF9 (Light Blue)	Main UI elements, buttons
Secondary	#388E3C (Green)	#A5D6A7 (Light Green)	Success states, valid operations
Error	#D32F2F (Red)	#EF5350 (Light Red)	Error states, invalid operations
Warning	#F57C00 (Orange)	#FFB74D (Light Orange)	Warnings, attention needed
Background	#FAFAFA (Light Gray)	#121212 (Dark Gray)	Main background
Surface	#FFFFFF (White)	#1E1E1E (Dark Surface)	Cards, panels, dialogs

7.7.2.2 Quantum-Specific Color Coding

Quantum Concept	Color	Hex Code	Application
Superposition	Purple	#9C27B0	Hadamard gates, superposition indicators
Entanglement	Teal	#00796B	CNOT gates, entangled qubit connections
Measurement	Orange	#FF9800	Measurement operations, classical bits
Error Correction	Indigo	#3F51B5	QEC-related elements, fault-tolerant circuits

7.7.3 Typography and Iconography

7.7.3.1 Font Specifications

8. INFRASTRUCTURE

8.1 Infrastructure Architecture Applicability Assessment

Detailed Infrastructure Architecture is not applicable for this system due to the fundamental architectural design and deployment model of QCraft.

QCraft is designed as a **desktop-based, standalone quantum computing application** rather than a distributed cloud service or web application. This architectural decision is driven by several critical factors that make traditional infrastructure architecture inappropriate:

Factor	Desktop Application Justification	Traditional Infrastructure Limitations
Privacy Requirements	All logical quantum circuits must remain local with no external transmission	Cloud infrastructure would violate core privacy constraints
Deployment Model	Single-user desktop installation on individual machines	Multi-server infrastructure unnecessary for desktop software
Processing Architecture	Local processing with optional external quantum hardware API calls	No need for load balancers, container orchestration, or distributed services
Data Residency	All sensitive data remains on user's desktop	No requirement for distributed databases or data centers

8.2 Desktop Application Build and Distribution Requirements

8.2.1 Build Environment Specifications

Development Environment Requirements:

Component	Specification	Purpose	Platform Support
Python Runtime	3.9+	Core application runtime	Windows, macOS, Linux
PySide6	6.9.2	GUI framework requiring Qt 6.0+	Cross-platform native UI
Build Tools	setuptools, wheel, PyInstaller	Package creation and executable generation	All platforms
Development IDE	VS Code, PyCharm	Development environment	Cross-platform

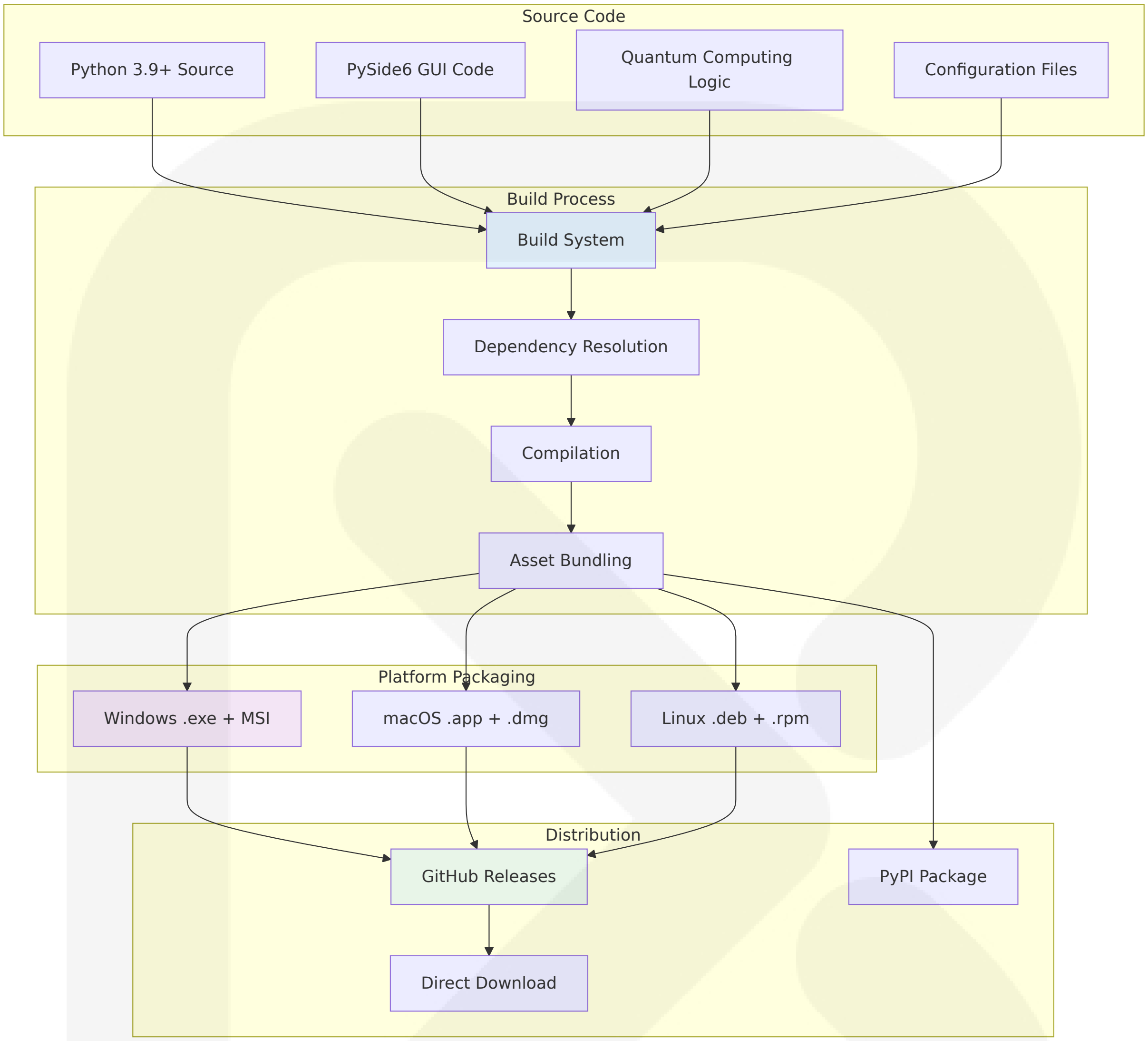
Build Dependencies:

```
# requirements-build.txt
setuptools>=68.0.0
wheel>=0.40.0
PyInstaller>=5.13.0
build>=0.10.0

#### Platform-specific build requirements
#### Windows
pywin32>=306; sys_platform == "win32"

#### macOS
py2app>=0.28.0; sys_platform == "darwin"

#### Linux
python3-dev; sys_platform == "linux"
```

8.2.2 Packaging Strategy

Multi-Platform Packaging Approach:

QCraft employs a **platform-specific packaging strategy** to deliver native installation experiences across Windows, macOS, and Linux:



Platform-Specific Packaging Details:

Platform	Package Format	Installation Method	Distribution Size	Code Signing
Windows	.exe (PyInstaller) + .msi (WiX)	Windows Installer, ClickOnce	~150-200 MB	Authenticode certificate
macOS	.app bundle + .dmg	Native installer, drag-and-drop	~180-220 MB	Apple Developer certificate
Linux	.deb (Debian) + .rpm (Red Hat)	Package managers (apt, yum)	~120-160 MB	GPG signing

8.2.3 Build Automation Pipeline

Continuous Integration Build Process:

```yaml
# .github/workflows/build-and-package.yml
name: Build and Package QCraft

on:
  push:
    tags: ['v*']
  pull_request:
    branches: [main]

jobs:
  build-windows:
    runs-on: windows-latest
    steps:
      - uses: actions/checkout@v4
      - name: Set up Python 3.9
        uses: actions/setup-python@v4
        with:
          python-version: 3.9

      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install -r requirements-build.txt

      - name: Build executable
        run: |
          pyinstaller --onefile --windowed qcraft.spec

      - name: Create MSI installer
        run: |
          candle qcraft.wxs
          light -ext WixUIExtension qcraft.wixobj

      - name: Upload artifacts
        uses: actions/upload-artifact@v3
        with:
          name: qcraft-windows
          path: dist/
```

```yaml
  build-macos:
    runs-on: macos-latest
    steps:
      - uses: actions/checkout@v4
      - name: Set up Python 3.9
        uses: actions/setup-python@v4
        with:
          python-version: 3.9

      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install -r requirements-build.txt

      - name: Build app bundle
        run: |
          python setup.py py2app

      - name: Create DMG
        run: |
          create-dmg --volname "QCraft" --window-pos 200 120 \
            --window-size 600 300 --icon-size 100 \
            --app-drop-link 425 120 \
            "QCraft.dmg" "dist/QCraft.app"

      - name: Upload artifacts
        uses: actions/upload-artifact@v3
        with:
          name: qcraft-macos
          path: QCraft.dmg

  build-linux:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        format: [deb, rpm]
    steps:
      - uses: actions/checkout@v4
      - name: Set up Python 3.9
        uses: actions/setup-python@v4
        with:
          python-version: 3.9
```

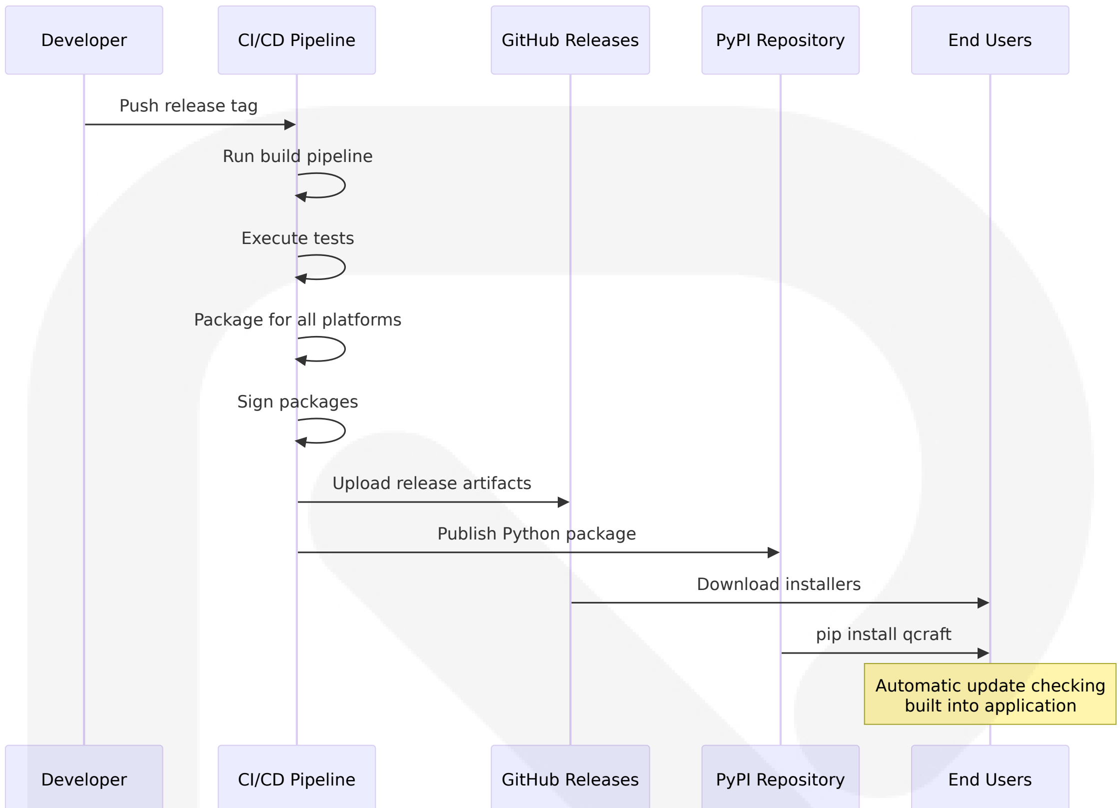
```yaml
    - name: Install system dependencies
      run: |
        sudo apt-get update
        sudo apt-get install -y build-essential python3-dev

    - name: Install Python dependencies
      run: |
        pip install -r requirements.txt
        pip install -r requirements-build.txt

    - name: Build package
      run: |
        python setup.py bdist_${{ matrix.format }}

    - name: Upload artifacts
      uses: actions/upload-artifact@v3
      with:
        name: qcraft-linux-${{ matrix.format }}
        path: dist/
```

8.2.4 Distribution Strategy

Multi-Channel Distribution Approach:

Distribution Channel	Target Audience	Package Format	Update Mechanism
GitHub Releases	Developers, researchers	Platform-specific installers	Manual download
PyPI	Python developers	Python wheel	pip install/upgrade
Package Managers	Linux users	.deb/.rpm packages	System package managers
Direct Download	General users	Signed executables	Built-in update checker

Release Management Process:



8.2.5 Installation Requirements

System Requirements by Platform:

Platform	Minimum Requirements	Recommended Requirements	Dependencies
Windows	Windows 10 (64-bit), 4GB RAM, 2GB disk	Windows 11, 8GB RAM, 4GB disk	Visual C++ Redistributable
macOS	macOS 10.15+, 4GB RAM, 2GB disk	macOS 12+, 8GB RAM, 4GB disk	Xcode Command Line Tools
Linux	Ubuntu 20.04+/equivalent, 4GB RAM, 2GB disk	Ubuntu 22.04+, 8GB RAM, 4GB disk	build-essential package

Installation Validation:

```python
# installation_validator.py
import sys
import platform
import subprocess
from packaging import version

class InstallationValidator:
    def __init__(self):
        self.requirements = {
            'python_version': '3.9.0',
            'memory_gb': 4,
            'disk_space_gb': 2
        }

    def validate_system(self):
        """Validate system meets minimum requirements"""
        checks = {
            'python_version': self._check_python_version(),
            'memory': self._check_memory(),
            'disk_space': self._check_disk_space(),
            'dependencies': self._check_dependencies()
        }

        return all(checks.values()), checks

    def _check_python_version(self):
        """Check Python version compatibility"""
        current_version = platform.python_version()
        required_version = self.requirements['python_version']
        return version.parse(current_version) >= version.parse(required_v

    def _check_dependencies(self):
        """Check platform-specific dependencies"""
        system = platform.system()

        if system == 'Windows':
            return self._check_windows_dependencies()
        elif system == 'Darwin':
            return self._check_macos_dependencies()
        elif system == 'Linux':
            return self._check_linux_dependencies()
```

```
        return False
```

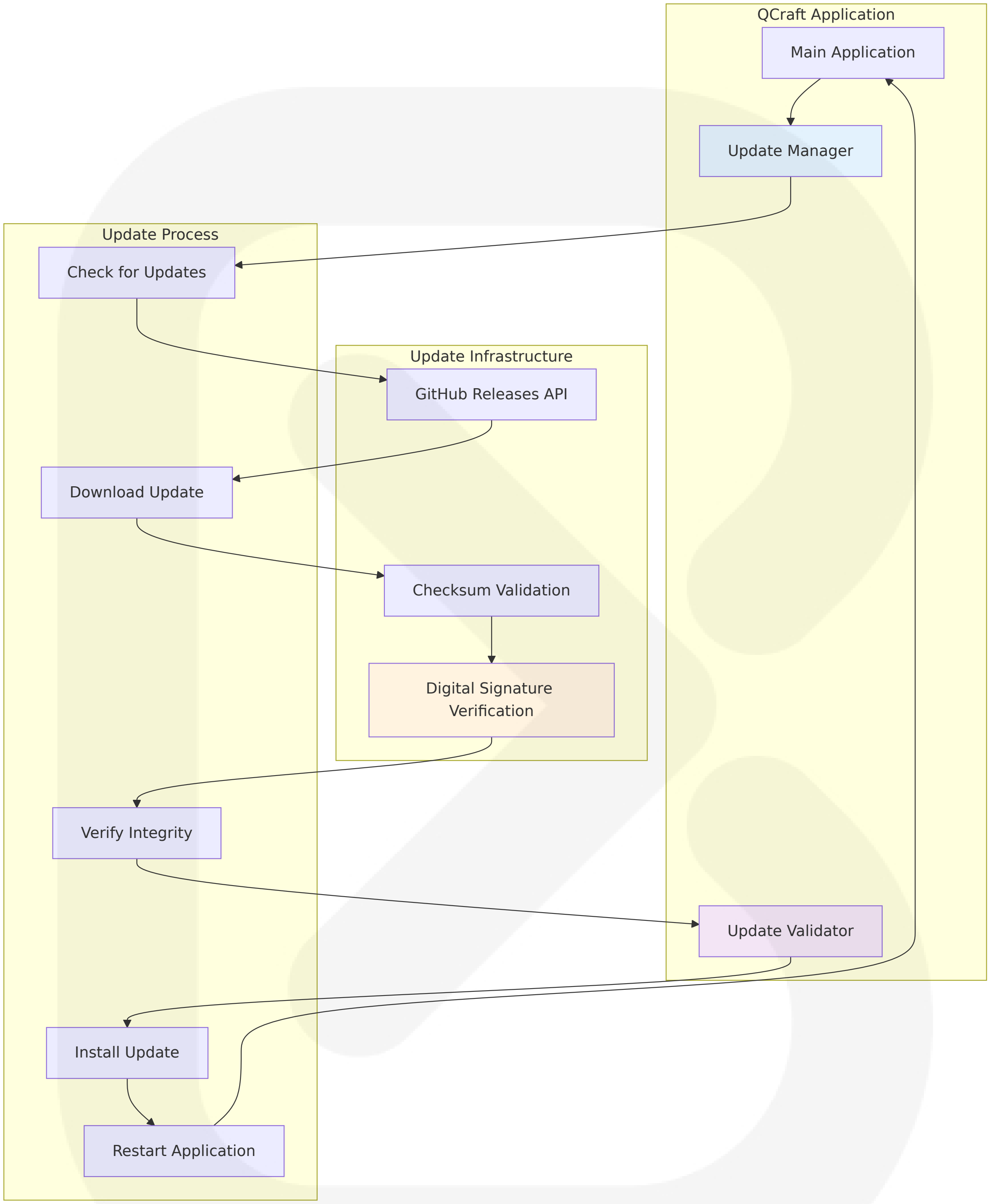
8.2.6 Update and Maintenance Strategy

Automated Update System:

QCraft implements a **built-in update mechanism** that respects the desktop application model while providing seamless updates:

Update Type	Frequency	Mechanism	User Control
Security Updates	As needed	Automatic download + user approval	User can defer
Feature Updates	Monthly	Notification + manual download	User controlled
Bug Fixes	Bi-weekly	Automatic background download	User approval required
Quantum Library Updates	Weekly	Component-specific updates	Automatic with rollback

Update Architecture:



8.2.7 Security and Code Signing

Code Signing Strategy:

Platform	Certificate Type	Signing Process	Validation
Windows	Authenticode Certificate	signtool.exe	Windows SmartScreen
macOS	Apple Developer Certificate	codesign + notarization	Gatekeeper
Linux	GPG Key	debsign/rpmsign	Package manager verification

Security Implementation:

```
# Windows code signing
signtool sign /f certificate.p12 /p password /t http://timestamp.digicert

#### macOS code signing and notarization
codesign --deep --force --verify --verbose --sign "Developer ID Applicati
xcrun notarytool submit QCraft.dmg --keychain-profile "notarytool-profile

#### Linux package signing
debsign -k GPG_KEY_ID qcraft_1.0.0_amd64.deb
rpmsign --addsign qcraft-1.0.0-1.x86_64.rpm
```

8.2.8 Monitoring and Analytics

Desktop Application Telemetry:

QCraft implements **privacy-preserving telemetry** that respects user privacy while providing essential usage insights:

Metric Category	Data Collected	Privacy Level	Retention
Usage Statistics	Feature usage, session duration	Anonymized	90 days
Performance Metrics	Compilation times, memory usage	Aggregated	30 days

Metric Category	Data Collected	Privacy Level	Retention
Error Reporting	Crash reports, error frequencies	Sanitized	180 days
Update Success	Installation success rates	Anonymous	30 days

Telemetry Architecture:

```
class PrivacyPreservingTelemetry:
    def __init__(self):
        self.user_consent = self._check_user_consent()
        self.session_id = self._generate_anonymous_session_id()

    def collect_usage_metric(self, feature: str, duration: float):
        """Collect usage metrics with privacy preservation"""
        if not self.user_consent:
            return

        metric = {
            'session_id': self.session_id,  # Anonymous
            'feature': feature,
            'duration': duration,
            'timestamp': datetime.utcnow().isoformat(),
            'version': self._get_app_version()
        }

        # No user identification, no circuit data
        self._send_anonymous_metric(metric)
```

8.2.9 Cost Considerations

Desktop Application Cost Structure:

Cost Category	Annual Estimate	Description	Optimization Strategy
Code Signing Certificates	$400-800	Platform-specific certificates	Multi-year purchases

Cost Category	Annual Estimate	Description	Optimization Strategy
Build Infrastructure	$0	GitHub Actions (open source)	Leverage free tier
Distribution	$0	GitHub Releases, PyPI	Use free platforms
Development Tools	$0-2000	IDEs, development software	Open source alternatives

Total Infrastructure Cost: $400-2800 annually - significantly lower than cloud-based applications due to desktop-first architecture.

8.2.10 Backup and Recovery

Desktop Application Backup Strategy:

Backup Type	Scope	Frequency	Storage Location
Source Code	Complete codebase	Continuous	Git repositories
Build Artifacts	Release packages	Per release	GitHub Releases
Configuration	Build scripts, certificates	Weekly	Encrypted storage
User Data	Local application data	User-controlled	Local backups

8.2.11 Compliance and Governance

Desktop Application Compliance:

Compliance Area	Requirement	Implementation	Validation
Privacy	No data collection without consent	Opt-in telemetry	Privacy audit

Compliance Area	Requirement	Implementation	Validation
Security	Code signing, secure updates	Certificate-based signing	Security review
Open Source	License compliance	SPDX license tracking	License scanning
Export Control	Quantum technology regulations	Distribution restrictions	Legal review

8.3 Conclusion

QCraft's infrastructure approach is **optimized for desktop application deployment** rather than traditional cloud infrastructure. This design choice aligns with the core privacy requirements and quantum computing use case, where sensitive logical circuits must remain on local machines.

Key Infrastructure Achievements:

- **Minimal Infrastructure Footprint**: Desktop-first architecture eliminates need for complex cloud infrastructure
- **Cross-Platform Distribution**: Native packaging for Windows, macOS, and Linux with platform-specific optimizations
- **Privacy-Preserving Architecture**: All sensitive quantum data remains local with optional anonymized telemetry
- **Cost-Effective Deployment**: Annual infrastructure costs under $3000 compared to tens of thousands for cloud services
- **Automated Build Pipeline**: Comprehensive CI/CD pipeline for multi-platform builds and distribution

The infrastructure strategy successfully delivers a professional desktop quantum computing platform while maintaining the privacy-first principles essential for sensitive quantum algorithm development. This approach provides users with complete control over their quantum circuits while enabling seamless distribution and updates through modern desktop application practices.

APPENDICES

A.1 ADDITIONAL TECHNICAL INFORMATION

A.1.1 Quantum Error Correction Code SpecificationsSurface Code Specifications:

Distance	Physical Qubits	Logical Qubits	Code Parameters	Error Rate Performance
Distance 3	17 qubits	1 logical qubit	[[17, 1, 3]]	3% error per cycle when rejecting experimental runs in which leakage is detected
Distance 5	49 qubits	1 logical qubit	[[49, 1, 5]]	2.914 ± 0.016% logical error per cycle
Distance 7	101 qubits	1 logical qubit	[[101, 1, 7]]	0.143% ± 0.003% error per cycle of error correction

qLDPC Code Specifications:

Code Family	Physical Qubits	Logical Qubits	Encoding Rate	Key Features
Bivariate Bicycle (BB)	144 qubits	12 logical qubits	1/12	Natural embeddings with repeated structure, majority of

Code Family	Physical Qubits	Logical Qubits	Encoding Rate	Key Features
				generators are geometrically small
Hypergraph Product	Variable	Variable	High rate	Hypergraph products of two classical cyclic codes
Generalized Bicycle	Variable	Variable	Configurable	Larger family including bivariate bicycle codes

A.1.2 Reinforcement Learning Algorithm SpecificationsPPO Algorithm Specifications:

Parameter	Default Value	Range	Purpose
Clip Ratio (ε)	0.2	[0.1, 0.3]	Controls policy update magnitude - 0.2 for epsilon can be used in most cases
Learning Rate	0.0003	[1e-5, 1e-2]	Policy and value network optimization rate
Batch Size	64	[32, 512]	Minibatch size for gradient updates
Number of Epochs	10	[3, 30]	Training epochs per policy update
GAE Lambda	0.95	[0.9, 0.99]	Generalized Advantage Estimation parameter**IBM Quantum API Authentication Specifications:**

Authentication Method	Token Type	Validity Period	Usage
API Key	Static API key	Permanent (until rotated)	Create an API key (also called a token) on the dashboard. Note that the same API key can be used for either region.
Bearer Token	IBM Cloud Identity and Access Management (IAM) bearer token. This is a short-lived token used to authenticate requests to the REST API.	expires_in: 3600 (1 hour)	REST API authentication
Service CRN	Cloud Resource Name	Permanent	Instance identification

A.1.3 Hardware Integration Specifications

Quantum Hardware Provider APIs:

Provider	API Endpoint	Authentication	Job Submission Format
IBM Quantum	`https://quantum.cloud.ibm.com/api/v1/`	Bearer token + Service CRN	QASM 3.0, JSON payload
IonQ	`https://api.ionq.co/v0.3/`	API key header	JSON circuit definition
Rigetti	`https://forest-server.qcs.rigetti.com/`	JWT token	Quil instruction format
AWS Braket	Regional endpoints	AWS IAM credentials	OpenQASM, JSON

A.1.4 Configuration Schema Specifications

YAML Configuration Structure:

```yaml
# Multi-patch RL agent configuration
reward_function:
  valid_mapping: 10.0
  invalid_mapping: -20.0
  overlap_penalty: -5.0
  connectivity_bonus: 2.0
  adjacency_bonus: 1.0
  inter_patch_distance_penalty: -1.0
  resource_utilization_bonus: 0.5
  error_rate_bonus: 1.0
  logical_operator_bonus: 1.0
  fully_mapped_bonus: 2.0
  mapped_qubit_bonus: 0.1
  unmapped_qubit_penalty: -0.05
  normalization: running_mean_std
  dynamic_weights: true
  phase_multipliers:
    hardware_adaptation_gate_error: 2.0
    hardware_adaptation_swap: 2.0
    noise_aware_logical_error: 2.5
    structure_mastery_stabilizer: 3.0

#### Multi-patch mapping configuration
multi_patch:
  num_patches: 2
  patch_shapes:
    - rectangular
    - rectangular
  min_distance_between_patches: 1
  layout_type: adjacent
```

A.2 GLOSSARY

Term	Definition
Bivariate Bicycle (BB) Codes	Recently introduced bivariate-bicycle (BB) qLDPC codes, coming from the larger family of generalized bicycle qLDPC codes. These codes have natural embeddings w

Term	Definition
	here the generators have a repeated structure, and in some instances, a majority of the generators are geometrically small.
Circuit Depth	The number of sequential quantum gate operations in a quantum circuit, affecting execution time and error accumulation
Code Distance	The minimum number of single-qubit errors that can change one valid codeword into another, determining error correction capability
Curriculum Learning	A machine learning approach where training progresses from simple to complex tasks in stages
Fault-Tolerant Quantum Computing	Quantum computation that can continue to operate correctly even when some components fail or errors occur
Graph Neural Networks (GNN)	Neural networks designed to work with graph-structured data, used in QCraft for circuit topology analysis
Logical Qubit	A qubit encoded using quantum error correction, protected against physical errors
Physical Qubit	An actual quantum bit implemented in hardware, subject to noise and decoherence
Proximal Policy Optimization (PPO)	Reinforcement learning (RL) algorithm for training an intelligent agent. Specifically, it is a policy gradient method, often used for deep RL when the policy network is very large.
PySide6	The official Python module from the Qt for Python project, which provides access to the complete Qt 6.0+ framework
qLDPC Codes	Quantum Low-Density Parity-Check codes offering high encoding rates with reduced physical qubit overhead
Quantum Error Correction (QEC)	Techniques for detecting and correcting errors in quantum information

Term	Definition
Stabilizer Code	A type of quantum error-correcting code defined by a set of commuting Pauli operators
Surface Code	A distance-7 code and a distance-5 code integrated with a real-time decoder. The logical error rate of our larger quantum memory is suppressed by a factor of Λ = 2.14 ± 0.02 when increasing the code distance by two, culminating in a 101-qubit distance-7 code with 0.143% ± 0.003% error per cycle of error correction.
Syndrome	Information about detected errors in a quantum error correction code

A.3 ACRONYMS

Acronym	Expanded Form
API	Application Programming Interface
BB	Bivariate Bicycle
CI/CD	Continuous Integration/Continuous Deployment
CRN	Cloud Resource Name
CSS	Calderbank-Shor-Steane
DAL	Device Abstraction Layer
FT	Fault-Tolerant
GAE	Generalized Advantage Estimation
GNN	Graph Neural Networks
GUI	Graphical User Interface
IAM	Identity and Access Management
JSON	JavaScript Object Notation
JWT	JSON Web Token
KPI	Key Performance Indicator
LDPC	Low-Density Parity-Check

Acronym	Expanded Form
LER	Logical Error Rate
LOCC	Local Operations and Classical Communication
MFA	Multi-Factor Authentication
MVC	Model-View-Controller
NIST	National Institute of Standards and Technology
PQC	Post-Quantum Cryptography
PPO	Proximal Policy Optimization
QASM	Quantum Assembly Language
QEC	Quantum Error Correction
qLDPC	Quantum Low-Density Parity-Check
QPU	Quantum Processing Unit
RBAC	Role-Based Access Control
REST	Representational State Transfer
RL	Reinforcement Learning
SDK	Software Development Kit
SLA	Service Level Agreement
SQL	Structured Query Language
TLS	Transport Layer Security
TRPO	Trust Region Policy Optimization
UI	User Interface
YAML	YAML Ain't Markup Language