perplexity

make it as modular architecture

I'll restructure the comprehensive quantum cryptography education platform into a clean modular architecture. Here's how to organize it for better maintainability, scalability, and development workflow:

Proposed Modular Architecture

```
quantum_cryptography_simulator/
├── main.py                          # Main application entry point
├── config/
│   ├── __init__.py
│   ├── settings.py                  # Application configuration
│   └── constants.py                 # Quantum constants and parameters
├── core/
│   ├── __init__.py
│   ├── session_manager.py          # Session state management
│   ├── quantum_simulator.py        # Core quantum operations
│   └── achievement_system.py       # Gamification engine
├── modules/
│   ├── __init__.py
│   ├── polarization/
│   │   ├── __init__.py
│   │   ├── main.py                 # Main polarization module
│   │   ├── quantum_ops.py          # Quantum operations
│   │   ├── visualizations.py       # Bloch sphere & plots
│   │   └── tutorial.py             # Tutorial content
│   ├── bb84/
│   │   ├── __init__.py
│   │   ├── main.py                 # Main BB84 module
│   │   ├── protocol.py             # BB84 protocol implementation
│   │   ├── circuit_builder.py      # Quantum circuit construction
│   │   └── analysis.py             # Results analysis
│   ├── channel_errors/
│   │   ├── __init__.py
│   │   ├── main.py                 # Main error module
│   │   ├── noise_models.py         # Quantum noise simulation
│   │   └── error_analysis.py       # Error rate calculations
│   ├── eavesdropping/
│   │   ├── __init__.py
│   │   ├── main.py                 # Main eavesdropping module
│   │   ├── attack_simulations.py   # Attack scenarios
│   │   └── detection.py            # Detection algorithms
│   ├── error_correction/
│   │   ├── __init__.py
│   │   ├── main.py                 # Main correction module
│   │   ├── algorithms.py           # Correction algorithms
│   │   └── cascade.py              # CASCADE protocol
│   └── security_analysis/
```

```
│       ├── __init__.py
│       ├── main.py                 # Main security module
│       ├── key_rates.py            # Key rate calculations
│       └── bounds.py               # Security bounds analysis
├── utils/
│   ├── __init__.py
│   ├── quantum_utils.py            # Quantum computing utilities
│   ├── plotting.py                 # Common plotting functions
│   ├── data_processing.py          # Data manipulation utilities
│   └── validators.py               # Input validation
├── components/
│   ├── __init__.py
│   ├── ui_components.py            # Reusable UI components
│   ├── forms.py                    # Form components
│   └── layouts.py                  # Layout templates
├── assets/
│   ├── styles.py                   # Custom CSS styles
│   └── images/                     # Static images
├── tests/
│   ├── __init__.py
│   ├── test_quantum_ops.py         # Quantum operation tests
│   ├── test_protocols.py           # Protocol tests
│   └── test_ui.py                  # UI component tests
├── requirements.txt
├── README.md
└── setup.py
```

Implementation Guide

1. Core Architecture Files

`main.py` **- Application Entry Point**

```
"""
Main application entry point for Quantum Cryptography Simulator
"""
import streamlit as st
from core.session_manager import SessionManager
from core.achievement_system import AchievementSystem
from components.layouts import MainLayout
from config.settings import APP_CONFIG
from modules import (
    polarization,
    bb84,
    channel_errors,
    eavesdropping,
    error_correction,
    security_analysis
)

def main():
    """Main application function"""

    # Initialize Streamlit configuration
```

```python
    st.set_page_config(**APP_CONFIG['streamlit'])

    # Initialize session management
    session_manager = SessionManager()
    session_manager.initialize()

    # Initialize achievement system
    achievement_system = AchievementSystem()

    # Create main layout
    layout = MainLayout()
    layout.render_header()

    # Module routing
    module_map = {
        "Quantum States & Polarization": polarization.main,
        "BB84 Protocol Basics": bb84.main,
        "Channel Errors & Noise": channel_errors.main,
        "Eavesdropping Detection": eavesdropping.main,
        "Error Correction": error_correction.main,
        "Security Analysis": security_analysis.main
    }

    # Render sidebar and get selected module
    selected_module = layout.render_sidebar(list(module_map.keys()))

    # Execute selected module
    try:
        module_function = module_map[selected_module]
        module_function()

        # Update achievements
        achievement_system.check_module_completion(selected_module)

    except Exception as e:
        st.error(f"Module error: {str(e)}")
        st.info("Please refresh or contact support if the issue persists.")

    # Render gamification sidebar
    achievement_system.render_sidebar()

    # Render footer
    layout.render_footer()

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

`core/session_manager.py` **- Centralized Session State**

```python
"""
Centralized session state management for all modules
"""
import streamlit as st
from datetime import datetime
from typing import Dict, Any, List
```

```
from config.constants import DEFAULT_SESSION_VALUES

class SessionManager:
    """Manages application-wide session state"""

    def __init__(self):
        self.session_keys = [
            'learning_progress',
            'module_progress',
            'achievements',
            'experiment_history',
            'current_keys',
            'user_preferences'
        ]

    def initialize(self) -> None:
        """Initialize all session state variables"""
        for key in self.session_keys:
            if key not in st.session_state:
                st.session_state[key] = DEFAULT_SESSION_VALUES.get(key, {})

    def get_module_progress(self, module_name: str) -> Dict[str, Any]:
        """Get progress for specific module"""
        return st.session_state.module_progress.get(module_name, {})

    def update_module_progress(self, module_name: str, progress_data: Dict[str, Any]) ->
        """Update progress for specific module"""
        if 'module_progress' not in st.session_state:
            st.session_state.module_progress = {}

        st.session_state.module_progress[module_name] = progress_data

    def add_experiment_record(self, module_name: str, experiment_data: Dict[str, Any]) ->
        """Add experiment to history"""
        experiment_record = {
            'module': module_name,
            'timestamp': datetime.now().isoformat(),
            **experiment_data
        }

        if 'experiment_history' not in st.session_state:
            st.session_state.experiment_history = []

        st.session_state.experiment_history.append(experiment_record)

    def get_user_preference(self, key: str, default: Any = None) -> Any:
        """Get user preference with fallback"""
        return st.session_state.get('user_preferences', {}).get(key, default)

    def set_user_preference(self, key: str, value: Any) -> None:
        """Set user preference"""
        if 'user_preferences' not in st.session_state:
            st.session_state.user_preferences = {}

        st.session_state.user_preferences[key] = value
```

```
    def reset_module_progress(self, module_name: str) -> None:
        """Reset progress for specific module"""
        if module_name in st.session_state.get('module_progress', {}):
            del st.session_state.module_progress[module_name]

    def export_session_data(self) -> Dict[str, Any]:
        """Export session data for backup/analysis"""
        return {
            key: st.session_state.get(key, {})
            for key in self.session_keys
            if key in st.session_state
        }
```

`config/settings.py` **- Configuration Management**

```
"""
Application configuration settings
"""

APP_CONFIG = {
    'streamlit': {
        'page_title': "Interactive QKD Simulator",
        'page_icon': "▯",
        'layout': "wide",
        'initial_sidebar_state': "expanded"
    },
    'quantum': {
        'default_shots': 1000,
        'max_qubits': 10,
        'simulator_backend': 'aer_simulator'
    },
    'ui': {
        'animation_speed': 0.1,
        'color_scheme': {
            'primary': '#667eea',
            'secondary': '#764ba2',
            'success': '#28a745',
            'warning': '#ffc107',
            'error': '#dc3545'
        }
    },
    'security': {
        'max_qber': 0.11,
        'default_security_parameter': 1e-8,
        'key_rate_threshold': 0.1
    },
    'achievements': {
        'enable_notifications': True,
        'save_progress': True,
        'leaderboard_enabled': False
    }
}

MODULES_CONFIG = {
    'polarization': {
```

```
        'max_angle': 180,
        'default_shots': 1000,
        'animation_enabled': True
    },
    'bb84': {
        'min_bits': 4,
        'max_bits': 50,
        'default_bits': 12
    },
    'eavesdropping': {
        'attack_types': ['intercept_resend', 'beam_splitting', 'photon_number_splitting']
        'detection_threshold': 0.05
    }
}
```

2. Module Structure Example

`modules/polarization/main.py` **- Module Entry Point**

```
"""
Main polarization learning module with modular architecture
"""
import streamlit as st
from typing import Dict, Any

from .quantum_ops import PolarizationSimulator
from .visualizations import BlochSphereVisualization, ProbabilityPlots
from .tutorial import PolarizationTutorial
from core.session_manager import SessionManager
from utils.validators import validate_angle, validate_shots
from components.ui_components import ParameterControls, ResultsDisplay

class PolarizationModule:
    """Main polarization module class"""

    def __init__(self):
        self.session_manager = SessionManager()
        self.simulator = PolarizationSimulator()
        self.visualizer = BlochSphereVisualization()
        self.tutorial = PolarizationTutorial()

    def render_interface(self) -> None:
        """Render the main polarization interface"""

        st.header("□ Quantum Polarization Laboratory")

        # Get module progress
        progress = self.session_manager.get_module_progress('polarization')

        # Tutorial mode check
        if not progress.get('tutorial_completed', False):
            self.tutorial.render()
            return

        # Main interface tabs
```

```
        tabs = st.tabs([
            "▯ Interactive Lab",
            "▯ Analysis",
            "▯ Progress"
        ])

        with tabs[^0]:
            self.render_lab_interface()

        with tabs[^1]:
            self.render_analysis_interface()

        with tabs[^2]:
            self.render_progress_interface()

    def render_lab_interface(self) -> None:
        """Render the interactive laboratory interface"""

        col1, col2 = st.columns([2, 1])

        with col1:
            # Parameter controls
            controls = ParameterControls()
            params = controls.render_polarization_controls()

            # Validation
            if not validate_angle(params['theta']):
                st.error("Invalid angle parameter")
                return

            # Simulation button
            if st.button("▯ Run Simulation", type="primary"):
                self.run_simulation(params)

        with col2:
            # Tutorial hints
            self.tutorial.render_contextual_hints(params)

            # Quick presets
            self.render_quick_presets()

    def render_visualization_section(self, params: Dict[str, Any]) -> None:
        """Render visualization section"""

        viz_tabs = st.tabs(["▯ Bloch Sphere", "▯ Probabilities"])

        with viz_tabs[^0]:
            fig = self.visualizer.create_bloch_sphere(params['theta'], params['phi'])
            st.plotly_chart(fig, use_container_width=True)

        with viz_tabs[^1]:
            prob_plots = ProbabilityPlots()
            fig = prob_plots.create_probability_bars(params['theta'], params['basis'])
            st.plotly_chart(fig, use_container_width=True)

    def run_simulation(self, params: Dict[str, Any]) -> None:
```

```python
        """Execute quantum simulation with given parameters"""

        try:
            with st.spinner("Running quantum simulation..."):
                results = self.simulator.run_experiment(params)

            # Display results
            results_display = ResultsDisplay()
            results_display.show_polarization_results(results)

            # Update session with experiment data
            self.session_manager.add_experiment_record('polarization', {
                'parameters': params,
                'results': results,
                'success': True
            })

            # Render visualizations
            self.render_visualization_section(params)

        except Exception as e:
            st.error(f"Simulation failed: {str(e)}")

    def render_quick_presets(self) -> None:
        """Render quick preset buttons"""

        st.subheader("⚡ Quick Presets")

        presets = [
            ("□ |0⟩", 0, "Rectilinear"),
            ("□ |1⟩", 90, "Rectilinear"),
            ("□ |+⟩", 45, "Diagonal"),
            ("□ |-⟩", 135, "Diagonal")
        ]

        cols = st.columns(2)
        for i, (label, angle, basis) in enumerate(presets):
            col = cols[i % 2]
            if col.button(label, key=f"preset_{i}"):
                # Update session state with preset values
                st.session_state['theta'] = angle
                st.session_state['basis'] = basis
                st.rerun()

    def render_analysis_interface(self) -> None:
        """Render analysis and data exploration interface"""

        # Get experiment history for this module
        history = [
            exp for exp in st.session_state.get('experiment_history', [])
            if exp.get('module') == 'polarization'
        ]

        if not history:
            st.info("□ Run some experiments to see analysis here!")
            return
```

```python
        # Analysis options
        analysis_type = st.selectbox(
            "Analysis Type",
            ["Learning Progress", "Accuracy Trends", "Parameter Exploration"]
        )

        if analysis_type == "Learning Progress":
            self.render_learning_progress(history)
        elif analysis_type == "Accuracy Trends":
            self.render_accuracy_trends(history)
        else:
            self.render_parameter_exploration(history)

    def render_progress_interface(self) -> None:
        """Render progress and achievement interface"""

        progress = self.session_manager.get_module_progress('polarization')

        # Progress metrics
        col1, col2, col3 = st.columns(3)

        with col1:
            st.metric("Experiments Run", progress.get('experiments_count', 0))

        with col2:
            st.metric("Concepts Mastered", len(progress.get('concepts_mastered', [])))

        with col3:
            avg_accuracy = progress.get('average_accuracy', 0)
            st.metric("Average Accuracy", f"{avg_accuracy:.1%}")

def main():
    """Main function for polarization module"""
    module = PolarizationModule()
    module.render_interface()
```

`modules/polarization/quantum_ops.py` **- Quantum Operations**

```python
"""
Quantum operations for polarization experiments
"""
import numpy as np
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from typing import Dict, Any, Tuple
from config.settings import APP_CONFIG

class PolarizationSimulator:
    """Handles quantum simulations for polarization experiments"""

    def __init__(self):
        self.backend = AerSimulator()
        self.default_shots = APP_CONFIG['quantum']['default_shots']
```

```
    def create_polarization_circuit(self, theta: float, phi: float) -> QuantumCircuit:
        """Create quantum circuit for polarization state"""

        qc = QuantumCircuit(1, 1)

        # State preparation
        if theta != 0:
            qc.ry(2 * np.pi * theta / 180, 0)

        if phi != 0:
            qc.rz(phi, 0)

        qc.measure(0, 0)

        return qc

    def run_experiment(self, params: Dict[str, Any]) -> Dict[str, Any]:
        """Run complete polarization experiment"""

        # Extract parameters
        theta = params.get('theta', 0)
        phi = params.get('phi', 0)
        basis = params.get('basis', 'Rectilinear')
        shots = params.get('shots', self.default_shots)

        # Create circuit
        qc = self.create_polarization_circuit(theta, phi)

        # Add basis rotation for measurement
        if basis.lower() == 'diagonal':
            qc.h(0)
        elif basis.lower() == 'circular':
            qc.sdg(0)
            qc.h(0)

        # Execute simulation
        job = self.backend.run(transpile(qc, self.backend), shots=shots)
        result = job.result()
        counts = result.get_counts()

        # Process results
        total_shots = sum(counts.values())
        prob_0 = counts.get('0', 0) / total_shots
        prob_1 = counts.get('1', 0) / total_shots

        # Calculate theoretical values
        theoretical = self.calculate_theoretical_probabilities(theta, phi, basis)

        return {
            'experimental': {
                'counts': counts,
                'probabilities': {'0': prob_0, '1': prob_1},
                'total_shots': total_shots
            },
            'theoretical': theoretical,
            'parameters': params,
```

```
            'circuit': qc,
            'accuracy': 1 - abs(prob_0 - theoretical['prob_0'])
        }

    def calculate_theoretical_probabilities(self, theta: float, phi: float, basis: str) -
        """Calculate theoretical measurement probabilities"""

        theta_rad = np.pi * theta / 180

        if basis.lower() == 'rectilinear':
            prob_0 = np.cos(theta_rad / 2) ** 2
        elif basis.lower() == 'diagonal':
            prob_0 = 0.5 * (1 + np.sin(theta_rad))
        else:  # circular
            prob_0 = 0.5 * (1 + np.cos(theta_rad))

        return {
            'prob_0': prob_0,
            'prob_1': 1 - prob_0
        }
```

3. Utility and Component Modules

`utils/quantum_utils.py` **- Shared Quantum Utilities**

```
"""
Shared quantum computing utilities across all modules
"""
import numpy as np
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from typing import List, Dict, Any, Tuple
from config.constants import QUANTUM_CONSTANTS

class QuantumStateHelper:
    """Helper class for quantum state operations"""

    @staticmethod
    def bloch_coordinates(theta: float, phi: float) -> Tuple[float, float, float]:
        """Convert spherical to Cartesian coordinates for Bloch sphere"""
        theta_rad = np.pi * theta / 180
        phi_rad = np.pi * phi / 180

        x = np.sin(theta_rad) * np.cos(phi_rad)
        y = np.sin(theta_rad) * np.sin(phi_rad)
        z = np.cos(theta_rad)

        return x, y, z

    @staticmethod
    def binary_entropy(x: float) -> float:
        """Calculate binary entropy function"""
        if x <= 0 or x >= 1:
            return 0
        return -x * np.log2(x) - (1 - x) * np.log2(1 - x)
```

```python
    @staticmethod
    def fidelity(state1: np.ndarray, state2: np.ndarray) -> float:
        """Calculate fidelity between two quantum states"""
        return abs(np.vdot(state1, state2)) ** 2

class CircuitBuilder:
    """Advanced quantum circuit building utilities"""

    @staticmethod
    def create_bell_state(state_type: str = 'phi_plus') -> QuantumCircuit:
        """Create Bell state circuits"""

        qc = QuantumCircuit(2, 2)

        # Create |Φ⁺⟩ = (|00⟩ + |11⟩)/√2
        qc.h(0)
        qc.cx(0, 1)

        # Modify for other Bell states
        if state_type == 'phi_minus':
            qc.z(0)
        elif state_type == 'psi_plus':
            qc.x(1)
        elif state_type == 'psi_minus':
            qc.x(1)
            qc.z(0)

        qc.measure_all()
        return qc

    @staticmethod
    def add_noise_model(qc: QuantumCircuit, noise_params: Dict[str, float]) -> QuantumCi
        """Add noise model to quantum circuit"""
        # Implementation for adding various noise types
        # This is a placeholder for full noise model implementation
        return qc

def validate_quantum_params(**kwargs) -> Dict[str, bool]:
    """Validate quantum parameters"""
    validation_results = {}

    for param, value in kwargs.items():
        if param == 'theta':
            validation_results[param] = 0 <= value <= 180
        elif param == 'phi':
            validation_results[param] = 0 <= value <= 360
        elif param == 'shots':
            validation_results[param] = 1 <= value <= 100000
        elif param == 'qber':
            validation_results[param] = 0 <= value <= 0.5
        else:
            validation_results[param] = True

    return validation_results
```

`components/ui_components.py` **- Reusable UI Components**

```python
"""
Reusable UI components for consistent interface design
"""
import streamlit as st
import plotly.graph_objects as go
from typing import Dict, Any, List, Optional
from config.settings import APP_CONFIG

class ParameterControls:
    """Reusable parameter control widgets"""

    def render_polarization_controls(self) -> Dict[str, Any]:
        """Render polarization parameter controls"""

        col1, col2 = st.columns(2)

        with col1:
            theta = st.slider(
                "Polarization Angle (θ)",
                min_value=0, max_value=180,
                value=st.session_state.get('theta', 45),
                step=5,
                help="Angle from |0⟩ state on Bloch sphere"
            )

        with col2:
            phi = st.slider(
                "Azimuthal Angle (φ)",
                min_value=0, max_value=360,
                value=st.session_state.get('phi', 0),
                step=5,
                help="Phase angle on Bloch sphere"
            )

        basis = st.selectbox(
            "Measurement Basis",
            ["Rectilinear", "Diagonal", "Circular"],
            index=0,
            help="Basis for quantum measurement"
        )

        shots = st.select_slider(
            "Number of Measurements",
            options=[100, 500, 1000, 2000, 5000],
            value=1000
        )

        return {
            'theta': theta,
            'phi': phi,
            'basis': basis,
            'shots': shots
        }

    def render_bb84_controls(self) -> Dict[str, Any]:
        """Render BB84 protocol controls"""
```

```
        col1, col2 = st.columns(2)

        with col1:
            n_bits = st.slider(
                "Number of Qubits",
                min_value=4, max_value=50,
                value=12,
                help="Number of qubits to send in protocol"
            )

        with col2:
            noise_level = st.slider(
                "Channel Noise (%)",
                min_value=0, max_value=25,
                value=0,
                help="Percentage of noise in quantum channel"
            ) / 100

        return {
            'n_bits': n_bits,
            'noise_level': noise_level
        }

class ResultsDisplay:
    """Standardized results display components"""

    def show_polarization_results(self, results: Dict[str, Any]) -> None:
        """Display polarization experiment results"""

        st.subheader("□ Experiment Results")

        exp_results = results['experimental']
        theo_results = results['theoretical']

        # Metrics display
        col1, col2, col3 = st.columns(3)

        with col1:
            st.metric(
                "Measured |0⟩",
                f"{exp_results['probabilities']['0']:.2%}",
                delta=f"Theory: {theo_results['prob_0']:.2%}"
            )

        with col2:
            st.metric(
                "Measured |1⟩",
                f"{exp_results['probabilities']['1']:.2%}",
                delta=f"Theory: {theo_results['prob_1']:.2%}"
            )

        with col3:
            st.metric(
                "Accuracy",
                f"{results['accuracy']:.1%}",
```

```
                delta="vs Theory"
            )

        # Counts table
        counts_df = pd.DataFrame([
            {"Outcome": "|0⟩", "Count": exp_results['counts'].get('0', 0)},
            {"Outcome": "|1⟩", "Count": exp_results['counts'].get('1', 0)}
        ])

        st.dataframe(counts_df, use_container_width=True)

    def show_bb84_results(self, results: Dict[str, Any]) -> None:
        """Display BB84 protocol results"""

        st.subheader("� BB84 Protocol Results")

        # Implementation for BB84 specific results
        pass

class ModuleHeader:
    """Standardized module header component"""

    @staticmethod
    def render(title: str, description: str, icon: str, gradient_colors: List[str] = None
        """Render module header with consistent styling"""

        if gradient_colors is None:
            gradient_colors = ['#4a90e2', '#357abd']

        st.markdown(f"""
        <div style="
            background: linear-gradient(135deg, {gradient_colors[^0]} 0%, {gradient_colo
            padding: 2rem; border-radius: 15px; color: white; text-align: center; margin-
            box-shadow: 0 6px 12px rgba(0, 0, 0, 0.15);
        ">
            <h1 style="margin: 0; font-size: 2.2em; font-weight: 700;">
                {icon} {title}
            </h1>
            <p style="margin: 1rem 0 0 0; font-size: 1.1em; opacity: 0.95;">
                {description}
            </p>
        </div>
        """, unsafe_allow_html=True)
```

4. Testing Structure

`tests/test_quantum_ops.py` **- Quantum Operations Testing**

```
"""
Unit tests for quantum operations
"""
import unittest
import numpy as np
from modules.polarization.quantum_ops import PolarizationSimulator
from utils.quantum_utils import QuantumStateHelper
```

```python
class TestPolarizationSimulator(unittest.TestCase):

    def setUp(self):
        self.simulator = PolarizationSimulator()

    def test_circuit_creation(self):
        """Test quantum circuit creation"""
        qc = self.simulator.create_polarization_circuit(45, 0)

        # Verify circuit has correct number of qubits and gates
        self.assertEqual(qc.num_qubits, 1)
        self.assertEqual(qc.num_clbits, 1)

        # Verify circuit operations
        self.assertGreater(len(qc.data), 0)  # Should have at least measurement

    def test_theoretical_calculations(self):
        """Test theoretical probability calculations"""
        # Test |0⟩ state in rectilinear basis
        result = self.simulator.calculate_theoretical_probabilities(0, 0, 'rectilinear')
        self.assertAlmostEqual(result['prob_0'], 1.0, places=3)
        self.assertAlmostEqual(result['prob_1'], 0.0, places=3)

        # Test |+⟩ state in rectilinear basis
        result = self.simulator.calculate_theoretical_probabilities(90, 0, 'rectilinear')
        self.assertAlmostEqual(result['prob_0'], 0.5, places=3)
        self.assertAlmostEqual(result['prob_1'], 0.5, places=3)

class TestQuantumUtils(unittest.TestCase):

    def test_bloch_coordinates(self):
        """Test Bloch sphere coordinate conversion"""
        x, y, z = QuantumStateHelper.bloch_coordinates(0, 0)
        self.assertAlmostEqual(z, 1.0, places=3)  # |0⟩ state at north pole

        x, y, z = QuantumStateHelper.bloch_coordinates(180, 0)
        self.assertAlmostEqual(z, -1.0, places=3)  # |1⟩ state at south pole

    def test_binary_entropy(self):
        """Test binary entropy calculation"""
        # Test perfect cases
        self.assertEqual(QuantumStateHelper.binary_entropy(0), 0)
        self.assertEqual(QuantumStateHelper.binary_entropy(1), 0)

        # Test maximum entropy
        self.assertAlmostEqual(QuantumStateHelper.binary_entropy(0.5), 1.0, places=3)

if __name__ == '__main__':
    unittest.main()
```

5. Development and Deployment Scripts

`setup.py` - Package Setup

```
"""
Setup script for Quantum Cryptography Simulator
"""
from setuptools import setup, find_packages

with open("README.md", "r", encoding="utf-8") as fh:
    long_description = fh.read()

with open("requirements.txt", "r", encoding="utf-8") as fh:
    requirements = [line.strip() for line in fh if line.strip() and not line.startswith('

setup(
    name="quantum-cryptography-simulator",
    version="2.0.0",
    author="Quantum Education Team",
    description="Interactive Quantum Key Distribution Simulator for Education",
    long_description=long_description,
    long_description_content_type="text/markdown",
    packages=find_packages(),
    classifiers=[
        "Development Status :: 4 - Beta",
        "Intended Audience :: Education",
        "License :: OSI Approved :: MIT License",
        "Operating System :: OS Independent",
        "Programming Language :: Python :: 3",
        "Programming Language :: Python :: 3.8",
        "Programming Language :: Python :: 3.9",
        "Programming Language :: Python :: 3.10",
    ],
    python_requires=">=3.8",
    install_requires=requirements,
    extras_require={
        "dev": ["pytest", "black", "flake8", "mypy"],
        "docs": ["sphinx", "sphinx-rtd-theme"],
    },
    entry_points={
        "console_scripts": [
            "qkd-simulator=main:main",
        ],
    },
)
```

🎯 Benefits of This Modular Architecture

1. Maintainability

- **Separation of Concerns**: Each module handles specific functionality
- **Clear Dependencies**: Well-defined interfaces between components
- **Easy Debugging**: Issues isolated to specific modules

2. Scalability

- **Add New Modules**: Easy to add new quantum protocols
- **Extend Features**: Enhance individual modules without affecting others
- **Performance**: Load only required components

3. Development Workflow

- **Team Development**: Multiple developers can work on different modules
- **Testing**: Comprehensive unit testing for each component
- **Version Control**: Clean Git history with module-specific commits

4. Code Reusability

- **Shared Components**: UI components and utilities used across modules
- **Consistent Design**: Standardized interface patterns
- **Configuration Management**: Centralized settings and constants

5. Educational Benefits

- **Progressive Learning**: Students can focus on individual concepts
- **Modular Understanding**: Clear separation between different QKD aspects
- **Extensibility**: Instructors can customize or add new content easily

This modular architecture transforms the monolithic quantum cryptography simulator into a professional, maintainable, and scalable educational platform that follows software engineering best practices while preserving all the rich educational content and functionality.

⁂