Roadmap/Execution Plan for Milestone-01

Step 1: Requirements Specification

Objective: Define and document the requirements for the implementation, analysis, and documentation of specified algorithms.

Tasks:

1. **Gathering the Requirements:** Understanding the specific requirements for each encryption algorithms and framework (McEliece with QCMDPC, Niederreiter with Goppa Code, Alekhnovichs with Goppa Code, HyMes with Goppa Code, Classic McEliece with Goppa Code, HQC).

2. **Defining the Scope:** Clearly outline the scope of implementation, analysis, and documentation.

3. **Documenting the Functional Requirements**: Specify the functionalities each algorithm must perform, including encryption, decryption, key generation, etc.

4. **Documenting the Non-Functional Requirements:** Performance metrics, security features, compliance with ANSI C/C++ standards, and interoperability.

5. **Creating the SRS Document:** Compile all gathered information into a comprehensive **Software Requirements Specification** document.

Step 2: Software Design

Objective: Design the software architecture and system components for implementing the algorithms.

Tasks:

1. **Architectural Design:** Define the architectures for the cryptographic library, including module interactions and data flow.

2. **Module Design:** Specify detailed designs for each algorithm and framework (McEliece, Niederreiter, Alekhnovichs, HyMes, Classic McEliece, HQC)

3. **Interface Design:** Define the interfaces for communication between modules and with external systems.

4. **Documentation:** Compile the **Software Design Document (SDD)** detailing the architecture, design patterns, modules, and interfaces.

Step 3: Implementation

Objective: Develop the ANSI C/C++ code for each algorithm as per the **SDD** and Document the **SDP** (**Software Development Plan**).

Tasks:

1. Set Up Development Environment: Configure the development environment with necessary tools, compilers, and libraries for ANSI C/C++.

2. Code Development:

- Implementation of (McEliece, Niederreiter, Alekhnovichs, and HyMes) framework with Goppa code.
- Implementation of Classic McEliece, HQC, BIKE.

3. Code Review and Refactoring: Perform regular code reviews to ensure code quality and adherence to standards. Refactor code as necessary.

4. Version Control: Use a version control system (e.g., Git) to manage code changes and collaboration.

Step 4: Testing (STP/Test Case Documents)

Objective: Test the implementations to ensure correctness, performance, and security.

Tasks:

1. Test Plan Development: Develop a **Software Test Plan (STP)** that outlines test cases, testing strategies, and performance benchmarks.

2. Unit Testing: Write and execute unit tests for each module and function within the algorithms.

3. Integration Testing: Test the integration of different modules to ensure they work together as expected.

4. Performance Testing: Measure the performance of each algorithm and optimize for efficiency.

5. Security Testing: Conduct security tests to ensure the algorithms meet the necessary cryptographic security standards.

6. Bug Fixing: Identify and fix bugs identified during testing.

Step 5: Acceptance Testing and Documentation (Final Documents)

Objective: Finalize testing and prepare the final documentation for the project.

Tasks:

1. **Acceptance Testing:** Perform acceptance testing to validate that all requirements have been met.

2. **Prepare Documentation:**

   - Detailed analysis reports for each algorithm.

   - User manual for the implemented library.

   - Technical documentation covering design, implementation, and testing.

3. **Review and Feedback**: Share the final documents with stakeholders for review and gather feedback.

4. **Finalize Deliverables:** Incorporate feedback, finalize the deliverables, and prepare for project handoff.

Roadmap/Execution Plan for Milestone-02

Step 1: Requirements Specification (SRS)

Objective: Define and document the requirements for the implementation, analysis, and documentation of the specified cryptographic and quantum algorithms.

Tasks:

1. **Gather Requirements:** Identify specific requirements for all 3 code-based signature scheme(Curtois-Finiasz-Sendrier, LDGM and Random) and quantum implementations (QSIM/Qiskit) of Classic McEliece and HQC algorithms.

2. **Define Scope:** Clarify the extent of implementation and the specific functionalities to be covered for both classical and quantum algorithms.

3. **Functional Requirements:** Document encryption, decryption, signature creation, and verification processes for each algorithm.

4. **Non-Functional Requirements:** Specify performance expectations, security considerations, compliance with ANSI C/C++ standards, and quantum simulation constraints.

5. **Create SRS Document:** Compile all requirements into a comprehensive Software Requirements Specification document.

Step 2: Software Design (SDD)

Objective: Design the architecture and detailed components for implementing the algorithms.

Tasks:

1. **Architectural Design:** Define the overall architecture for both classical (ANSI C/C++) and quantum (QSIM/Qiskit) implementations.

2. **Module Design:**

- **Classical Algorithms:** Design modules for code-based signatures (Curtois-Finiasz-Sendrier, LDGM, Random).

- **Quantum Algorithms:** Design frameworks for simulating Classic McEliece, HQC.

3. **Interface Design:** Define interfaces for algorithm and framework.

4. **Documentation:** Prepare the Software Design Document (SDD) detailing architecture, module designs, and interfaces.

Step 3: Implementation (Development)

Objective: Develop the ANSI C/C++ and QSIM/Qiskit code for the algorithms and framework.

Tasks:

1. **Setup Development Environment:** Configure environments for ANSI C/C++ and QSIM/Qiskit.

2. **Code Development:**

-Implementation of code-based signature schemes (Curtois-Finiasz-Sendrier, LDGM, Random).

-Begin QSIM/Qiskit implementations of Classic McEliece, HQC.

3. **Code Review and Refactoring:** Regularly review code to ensure quality and optimize as needed.

4. **Version Control:** Maintain version control using Git for collaborative development and tracking changes.

Step 4: Testing and Validation

Objective: Test the implementation for accuracy, security, and performance.

Tasks:

1. **Test Plan Development:** Create a comprehensive Software Test Plan (STP) with test cases for each algorithm.
2. **Unit Testing:** Develop and execute unit tests for individual functions and modules.
3. **Integration Testing:** Ensure that different modules interact correctly and perform as expected.
4. **Performance Testing:** Measure algorithm performance, focusing on speed, resource usage, and scalability.
5. **Security Testing:** Validate the security of implementations, particularly focusing on cryptographic robustness.
6. **Quantum Validation:** For quantum algorithms, validate the simulation results and resource estimation.
7. **Bug Fixing:** Identify and fix any issues found during testing.

Step 5: Analysis and Documentation

Objective: Analyze the results and prepare comprehensive documentation for the project.

Tasks:

1. **Performance Analysis:** Analyze the performance data collected during testing, comparing against benchmarks.
2. **Security Analysis:** Evaluate the security features and resilience of each algorithm.
3. **Quantum Resource Estimation:** Document the quantum resource requirements (e.g., qubits, gates, run-time) for the quantum algorithms.
4. **Prepare Documentation:**
   - Detailed reports on the implementation, analysis, and testing of each algorithm.
   - Quantum simulation and resource estimation reports.
   - User manuals for the implemented cryptographic libraries.
   - Technical documentation covering design, implementation, and testing processes.
5. **Review and Finalization:** Review the documents with stakeholders, incorporate feedback, and finalize deliverables.

Roadmap/Execution Plan for Milestone-03

Step 1: Requirements Specification (SRS)

Objective: Define and document the requirements for implementing, Analyzing, and documenting the specified cryptographic and quantum algorithms.

Tasks:

1. **Gather Requirements:** Collaborate with the CAIR team to understand specific requirements for the two code-based KEM and signature schemes and their small scale QSIM/Qiskit implementation. ANSI C/C++ implementation of all seven Information Set Decoding (ISD) algorithms that are mentioned in RFP.

2. **Define Scope:** Identify the specific functionalities and performance metrics for LLL, BKZ.

3. **Functional Requirements:** Document the functional requirements for each algorithm.

4. **Non-Functional Requirements:** Document the non-functional requirements like performance, security, and compliance aspects.

5. **Create SRS Document:** Compile all requirements into a comprehensive Software Requirements Specification document.

Step 2: Software Design (SDD)

Objective: Design the architecture and detailed components for implementing the algorithms.

Tasks:

1. **Architectural Design:** Define the overall architecture for classical (ANSI C/C++) and quantum (QSIM/Qiskit) implementations.

2. **Module Design:**

- **Classical Algorithms:** Design modules for code-based KEMs, signature schemes, LLL, BKZ, ISD algorithms (Prange's, Stern's, BJMM, Dumer's, Both-may, May-Ozerov and Quantum ISD).

- **Quantum Algorithms**: Design frameworks for Quantum ISD algorithms.

3. **Interface Design:** Specify interfaces for interacting with different algorithm components and data flows.

4. **Documentation:** Prepare the Software Design Document (SDD) covering architecture, module designs, and interface details.

Step 3: Implementation (Development)

Objective: Develop the ANSI C/C++ and QSIM/Qiskit code for the algorithms.

Tasks:

1. **Setup Development Environment:** Configure environments for ANSI C/C++ and QSIM/Qiskit.

2. **Code Development:**

- Implementation of LLL and BKZ algorithms in ANSI C/C++. Start with the CAIR-provided code-based KEMs and signature schemes.

- Implementation of ISD algorithms and their variants (Prange's, Stern's, BJMM, Dumer's, Both-may, May-Ozerov and Quantum ISD).

- Finalizing remaining code-based KEMs and signature schemes and their QSIM/Qiskit implementation.

3. **Code Review and Refactoring:** Regularly review code for quality assurance and optimization.

4. **Version Control:** Use Git for version control to manage changes and collaboration.

Step 4: Testing and Validation

Objective: Test the implementation for accuracy, security, and performance.

Tasks:

1. **Test Plan Development:** Create a Software Test Plan (STP) with detailed test cases for each algorithm.

2. **Unit Testing:** Develop unit tests for each function and module.

3. **Integration Testing:** Test the integration of modules and their interactions.

4. **Performance Testing:** Benchmark the performance of each algorithm, focusing on execution time, memory usage, and scalability.

5. **Security Testing:** Validate the cryptographic security of implementations

6. **Bug Fixing:** Identify and resolve any issues found during testing.

Step 5: Analysis and Documentation

Objective: Analyze the results and prepare comprehensive documentation for the project.

Tasks:

1. **Performance Analysis:** Analyze the performance metrics collected during testing, with a focus on classical and quantum implementations.

2. **Security Analysis:** Evaluate the security features and robustness of the cryptographic algorithms.

3. **Quantum Resource Estimation:** Document the resource requirements for quantum algorithms, including qubits, gates, and execution time.

4. **Prepare Documentation:**

   - Detailed reports on the implementation, analysis, and testing of each algorithm.

   - Quantum simulation and resource estimation reports.

   - User manuals for the cryptographic libraries.

   - Technical documentation covering the design, implementation, and testing processes.

5. **Review and Finalization:** Review the documentation with stakeholders, incorporate feedback, and finalize the deliverables.

Step 6: Drafting Final Documents

Objective: Draft comprehensive technical documents covering all activities and results from Milestones 1, 2, and 3.

Tasks:

1. **Compile Existing Documentation:** Gather all reports, design documents, testing results, analysis reports, and any other documentation generated during the previous milestones.

2. **Organize Documentation:** Structure the documents as per the format.

3. **Draft Final Report:**

   - **Introduction:** Overview of the project objectives, scope, and milestones.

   - **Methodology:** Detailed description of the methodologies used for implementation, testing, and analysis.

   - **Results and Analysis**: Summary of the key findings from the implementation and testing of algorithms.

- **Conclusions:** Summarize the overall outcomes, effectiveness of the implemented algorithms, and potential areas for improvement.

- **Recommendations:** Suggest possible future work or improvements based on the project findings.

4. Draft Individual Technical Documents:

- **Milestone-Specific Reports:** Prepare detailed reports for Milestones 1, 2, and 3, each focusing on the specific tasks, implementations, and results related to the milestone.

- **Appendices:** Include additional information such as code snippets, test cases, detailed performance metrics, and references.

Step 7: Review and Validation of Documents

Objective: Ensure the accuracy, completeness, and quality of the drafted documents through a review process.

Tasks:

1. Internal Review: Conduct a thorough review of the draft documents within the project team. Focus on:

- **Technical Accuracy:** Verify that all technical details, methodologies, and results are accurately represented.

- **Completeness:** Ensure that all aspects of the project work are documented.

- **Clarity and Consistency:** Check for clarity in language, consistency in formatting.

2. Peer Review: Seek feedback from peers or external experts who were not directly involved in the project.

3. Validation Against Requirements: Cross-check the documentation against the original requirements specified in the SRS documents to ensure all objectives and deliverables are covered.

Step 8: Finalizing Documents

Objective: Finalize the documentation for submission and ensure it meets the required standards.

Tasks:

1. Incorporate Feedback: Revise the draft documents based on feedback received during the review phase. Address all comments, suggestions, and corrections.

2. Formatting and Proofreading:

- **Consistency:** Ensure consistent formatting across all documents, including headings, fonts, and citation styles.

- **Proofreading:** Conduct a final proofreading pass to eliminate any grammatical or typographical errors.

- **References and Citations:** Verify that all references and citations are accurate and correctly formatted.

3. Final Approval: Obtain the final approval.

4. Documentation Packaging: Compile final versions of all the documents.

Libraries/Packages

S.No	Libraries/Packages	Programming Language	Description
1.	**Liboqs-Python**	**Python**	**liboqs-python** is a Python binding for the liboqs library, which provides post-quantum cryptographic algorithms. This binding allows Python developers to use the post-quantum algorithms implemented in liboqs directly within Python applications. **Key Features of liboqs-python:** Post-Quantum Cryptography: Provides access to various post-quantum algorithms for public key encryption, key encapsulation mechanisms (KEMs), and digital signatures, including schemes like Kyber, NTRU, and SPHINCS+. **Open Source:** Typically, both liboqs and its Python bindings are open source, allowing for community contributions and transparency.
2.	**Pycrypto-Dome**	**Python**	**PyCryptodome** is a self-contained Python library that provides cryptographic functions and algorithms. It's a fork of the PyCrypto library, which is no longer maintained, and it aims to provide a secure and up-to-date set of cryptographic tools for Python developers.

			Key Features of PyCryptodome: **Symmetric Key Algorithms:** - AES (Advanced Encryption Standard) - DES (Data Encryption Standard) - 3DES (Triple DES) - Blowfish - ARC4 (RC4) **Asymmetric Key Algorithms:** - RSA (Rivest-Shamir-Adleman) - DSA (Digital Signature Algorithm) - ElGamal - ECC (Elliptic Curve Cryptography) - EdDSA (Edwards-Curve Digital Signature Algorithm) **Hash Functions:** - SHA-1 - SHA-256 - SHA-384 - SHA-512 - MD5 - RIPEMD-160 **Message Authentication Codes (MACs):** - HMAC (Hash-based Message Authentication Code) - CMAC (Cipher-based Message Authentication Code) **Random Number Generators:** - Cryptographically secure pseudo-random number generators (CSPRNGs) **Key Derivation Functions:** - PBKDF2 (Password-Based Key Derivation Function 2) - Scrypt

			Public Key Infrastructure (PKI): - X.509 Certificate Handling - PKCS#12 **Utilities:** - Encryption/Decryption of files - Conversion utilities (e.g., base64 encoding/decoding)
3.	**Qiskit**	**Python**	**Qiskit** is an open-source quantum computing framework developed by IBM. It provides tools for creating, simulating, and running quantum circuits on quantum computers and simulators. Here's an overview of its key features and components: **Key Features of Qiskit:** **Quantum Circuit Design:** - **Quantum Circuit Creation:** Design quantum circuits using a variety of quantum gates and operations. - **Circuit Visualization:** Tools for visualizing quantum circuits to help understand and debug quantum algorithms. **Quantum Computing Frameworks:** - **Qiskit Terra:** The foundational layer for creating and manipulating quantum circuits and optimizing quantum programs. - **Qiskit Aer:** Provides high-performance simulators for quantum circuits, enabling testing and debugging of quantum algorithms. - **Qiskit Ignis:** Offers tools for quantum error correction and noise analysis, which are crucial for developing reliable quantum algorithms. - **Qiskit Quantum Information Science:** A collection of tools for working with quantum information theory, such as quantum state tomography.

			Quantum Computing Hardware: - **Qiskit Providers:** Interfaces with real quantum hardware, including IBM Quantum's cloud-based quantum computers. This allows users to run their quantum circuits on actual quantum machines. **Quantum Development Environment:** - **Qiskit Notebooks:** Jupyter notebooks for interactive development and testing of quantum algorithms. - **Qiskit SDKs:** Python SDKs to easily integrate quantum computing into applications.
4.	**Cirq**	**Python**	**Cirq** is a Python framework developed by Google for simulating quantum circuits and running them on Google's quantum processors. It is mainly focused on quantum circuits that are compatible with near-term quantum computers, particularly quantum devices using superconducting qubits. **Features:** • **Quantum Circuit Design:** We can create and simulate quantum circuits with qubits, gates, and measurements. • **Device Connectivity:** Circuits can be designed to match the connectivity of actual quantum devices, which is important for optimizing performance on real hardware. • **Simulation and Noise Modelling:** It provides tools for simulating quantum circuits on classical computers and for modelling noise that occurs in real quantum hardware. • **Integration with Quantum Hardware:** Cirq can interface with quantum processors like Google's Sycamore.

5.	**QuTip**	**Python**	**QuTiP** is a Python library designed for simulating the dynamics of open quantum systems. It is widely used in academic research for quantum optics and quantum information theory. QuTiP allows you to study a broad range of quantum systems, from single qubits to more complex systems. **Features:** **Hamiltonian Simulation:** QuTiP allows you to simulate quantum systems by defining Hamiltonians and solving the Schrödinger or Lindblad master equations. **Time-Dependent Dynamics:** You can model and simulate time-evolving quantum systems. **Quantum State Evolution:** Tools for simulating both closed (unitary) and open (dissipative) quantum systems are available. **Visualization:** QuTiP provides visualization tools for quantum states, including Bloch sphere representation, density matrices, and Wigner functions.
6.	**OQS**	**Python**	**OQS** is a Python package that provides tools for modelling open quantum systems, where the system of interest interacts with an external environment. The package allows the simulation of quantum dynamics with various types of noise and decoherence. **Features:** Lindblad Master Equation: OQS provides methods to solve the Lindblad master equation, which is commonly used to describe open quantum systems. **Stochastic Quantum Trajectories:** You can simulate the stochastic evolution of quantum states, capturing the randomness inherent in quantum measurement

			processes. **Noise and Decoherence Modelling:** The package allows you to model various noise processes, such as dephasing and amplitude damping, which are crucial for studying quantum systems in real-world conditions.
7.	**OpenSSL**	**C/C++**	**OpenSSL** is a widely used library for implementing cryptographic functions and protocols. It supports a comprehensive range of cryptographic algorithms and features. Here's an overview of the major algorithms and functionalities provided by OpenSSL: **Symmetric Key Algorithms:** - AES (Advanced Encryption Standard), - DES (Data Encryption Standard), - 3DES (Triple DES), - Blowfish, - Camellia, - RC4. **Asymmetric Key Algorithms:** - RSA (Rivest-Shamir-Adleman), - DSA (Digital Signature Algorithm), - DH (Diffie-Hellman), - ECDSA (Elliptic Curve Digital Signature Algorithm), - ECDH (Elliptic Curve Diffie-Hellman), - EdDSA (Edwards-Curve Digital Signature Algorithm). **Hash Functions:** - SHA-1, - SHA-256, - SHA-384,

			- SHA-512, - MD5, - RIPEMD-160, - SHA-3. **Message Authentication Codes (MACs):** - HMAC (Hash-based Message, Authentication Code), - CMAC (Cipher-based Message Authentication Code). **Key Derivation Functions:** - PBKDF2 (Password-Based Key Derivation Function 2), - Scrypt, - HKDF (HMAC-based Key Derivation Function). Digital Certificates and Protocols: - X.509 Certificates, - PKCS12, - PKCS7, - SSL/TLS Protocols. **Random Number Generators:** dev/urandom (on Unix-like systems), Cryptographically secure pseudo-random number generators (CSPRNGs) **Other Cryptographic Operations:** Elliptic Curve Cryptography (ECC), Padding schemes (PKCS7, ISO 10126), Public Key Infrastructure (PKI)*
8.	**Crypto++**	**C/C++**	**Crypto++** offers a broad range of cryptographic algorithms. Here are some of the main categories and examples: **Symmetric Key Algorithms:**

			- AES (Advanced Encryption Standard) - DES (Data Encryption Standard) - 3DES (Triple DES) - Blowfish - Twofish - RC4 - RC5 - RC6 **Asymmetric Key Algorithms:** - RSA (Rivest-Shamir-Adleman) - ElGamal - DLP (Discrete Logarithm Problem) - DH (Diffie-Hellman) - ECDSA (Elliptic Curve Digital Signature Algorithm) - ECDH (Elliptic Curve Diffie-Hellman) **Hash Functions:** - SHA-1 - SHA-256 - SHA-512 - *MD5 (Message Digest Algorithm 5) - RIPEMD-160 - Tiger **Message Authentication Codes (MACs):** - HMAC (Hash-based Message Authentication Code) - CMAC (Cipher-based Message Authentication Code) **Random Number Generators:** - Fortuna - OS-specific RNGs (like /dev/urandom) **Other Cryptographic Primitives:**

			- Elliptic Curve Cryptography (ECC) - Scrypt (for password-based key derivation) - AES-GCM (Authenticated Encryption with Associated Data)
9.	**Liboqs**	**C/C++**	**liboqs** is a C library designed to facilitate the use of post-quantum cryptographic algorithms. It's part of the Open Quantum Safe (OQS) project, which aims to develop and standardise cryptographic algorithms that are secure against potential quantum computer attacks. Here's an overview of what liboqs provides: **Post-Quantum Cryptographic Algorithms:** **Public Key Encryption:** **Lattice-Based Cryptography:** - **Kyber:** A key encapsulation mechanism (KEM). - **NTRU:** A KEM and digital signature scheme. - **NTRUEncrypt:** A public key encryption scheme. **Code-Based Cryptography:** - **McEliece:** A public key encryption scheme. **Multivariate Polynomial Cryptography:** - **Rainbow:** A digital signature scheme. **Hash-Based Cryptography:** - **SPHINCS+:** A stateless hash-based signature scheme. **Digital Signatures:** **Lattice-Based Cryptography:** - **FrodoKEM:** A KEM and digital signature scheme. **Hash-Based Cryptography:** - XMSS (eXtended Merkle Signature

			Scheme) - SPHINCS+ **Hybrid Schemes:** - Combining classical and post-quantum algorithms for additional security. **Key Features:** - **API for Integration:** liboqs provides a uniform API for integrating post-quantum algorithms into existing systems. - **Testing and Validation**: The library includes tools for testing and validating implementations.
10.	**OpenABE**	**C/C++**	**OpenABE** (Open Attribute-Based Encryption) is an open-source library that provides implementations of attribute-based encryption (ABE) schemes. ABE is a type of public-key encryption where decryption is based on attributes rather than just identities. **Key Features of OpenABE:** **Attribute-Based Encryption Schemes:** - **Key-Policy Attribute-Based Encryption (KP-ABE):** In KP-ABE, the access policy is attached to the encryption key, and the ciphertext is associated with attributes. Decryption is possible only if the attributes match the policy defined in the key. - **Ciphertext-Policy Attribute-Based Encryption (CP-ABE):** In CP-ABE, the access policy is attached to the ciphertext, and the keys are associated with attributes. Decryption is possible only if the attributes in the key satisfy the policy defined in the ciphertext. **Open Source:** Being open source, it allows for community contributions and reviews, enhancing security and functionality over time.

S.No	Development Tools	Description
1.	**Vscode**	**Visual Studio Code** (**VSCode**) is a popular open-source code editor developed by Microsoft. It is known for its lightweight design, powerful features, and extensive customization options. **VSCode** supports a wide range of programming languages and comes with built-in Git integration, debugging tools, and extensions
2.	**Venv/anaconda**	A Python virtual environment is a self-contained directory that allows you to install packages and dependencies specific to a project, without affecting the global Python installation. It helps in managing multiple projects with different dependencies and avoiding conflicts between them. **Anaconda**, is a distribution of Python that comes with a package manager called conda, which is used for managing environments and installing packages. Anaconda simplifies the process of creating isolated environments and managing dependencies, especially for data science and machine learning projects. With Anaconda, you can create virtual environments that include not only Python libraries but also system dependencies, making it a comprehensive solution for managing environments and packages across various projects.
3.	**Qsim**	**Qsim** is a high-performance quantum circuit simulator developed by Google, designed to simulate quantum circuits on classical computers. It is optimized for large-scale quantum simulations and can handle complex circuits efficiently. Qsim supports integration with Cirq and is used to simulate circuits that can later be run on quantum hardware. Its focus on performance makes it ideal for testing and verifying quantum algorithms on classical machines before deploying them on actual quantum processors.
4.	**Qiskit**	**Qiskit** is an open-source quantum computing framework developed by IBM that allows users to create, simulate, and execute quantum circuits.

		Qiskit provides a comprehensive suite of tools for quantum programming, including modules for quantum algorithms, quantum machine learning, quantum chemistry, and quantum finance. It supports both simulation on classical computers and execution on IBM's quantum hardware. Qiskit is widely used for quantum research, education, and the development of quantum applications, making it one of the most popular quantum computing toolkits in the field.

S.No	Testing Tools	Programming Language	Description
1.	Pytest	Python	**Pytest** is a powerful and flexible testing framework for Python that is known for its simplicity and ease of use. It supports simple unit tests as well as complex functional testing. Pytest provides features such as fixtures, parameterized testing, and a rich set of plugins, making it a popular choice for testing in various Python projects. **Key Features:** - Simple syntax and auto-discovery of test cases. - Powerful fixtures for setup/teardown. - Supports parameterized testing. - Extensible with plugins, like for code coverage and parallel execution.
2.	**Unittest**	**Python**	**Unittest**, also known as unittest or PyUnit, is the built-in testing framework in Python. It is modeled after Java's JUnit and provides a standard way to write and organize tests in Python. Unittest is a little more verbose than Pytest but is widely used due to its inclusion in the Python standard library. **Key Features:** - Class-based testing structure. - Provides setup/teardown methods (setUp,

			tearDown). - Assertions to test conditions (assertEqual, assertTrue, etc.). - Supports test discovery and test suites.
3.	**Nose2**	**Python**	**Nose2** is the successor to the nose testing framework and is designed to be a simple and extendable test runner. It is compatible with Unittest and supports plugins for extended functionality. Nose2 aims to improve test discovery and make writing and running tests easier while maintaining compatibility with Unittest's style. **Key Features:** - Automatically discovers tests. - Plugin system for adding features like test coverage, profiling, and more. - Compatible with Unittest, allowing you to reuse existing tests. - Focus on simplicity and ease of use.
4.	**MinUnit**	**C**	**MinUnit** is a minimalistic unit testing framework for C. It is extremely lightweight, with only a few lines of code, and is ideal for small projects where you need basic test functionality without additional overhead. MinUnit provides a simple mechanism to write and execute tests in C programs. **Key Features:** - Very small footprint (around 25 lines of code). Minimal setup and no external dependencies. Suitable for embedding in low-level or embedded systems projects.
5.	**CxxTest**	**C++**	**CxxTest** is a unit testing framework for C++ that is designed to be easy to use and portable. It is a header-only framework, meaning it doesn't require linking with a separate library. CxxTest supports various testing features, including test discovery and exception

			handling, and it works without requiring runtime type information (RTTI). **Key Features:** - Header-only framework (no need for external libraries). - Test discovery and automatic test suite generation. - Works without requiring RTTI, making it suitable for a wide range of C++ environments. - Generates test runners automatically from test files.
6.	**Libcester**	**C**	**Libcester** is a testing framework for C, designed to be lightweight and simple. It focuses on providing a straightforward way to write tests without the complexity of larger frameworks. Libcester is particularly suitable for testing low-level C programs and libraries. **Key Features:** - Lightweight and simple to use. - Provides basic assertions for testing. - Focused on being efficient for low-level C projects.
7.	**Criterion**	**C**	**Criterion** is a cross-platform unit testing framework for C that emphasizes simplicity and flexibility. It supports automatic test registration, parallel test execution, and provides detailed output with colored formatting. Criterion also includes features like mocking and test isolation, making it a more advanced option for C projects. **Key Features:** - Automatic test registration (no need for manual test setup). - Parallel test execution to speed up testing. - Detailed and colorful output for easy debugging.

			- Supports mocking and test isolation. - Cross-platform support for Linux, macOS, and Windows.

S.No	Configuration Management Tools	Description
1.	**Open Project**	**Open Project** is an open-source project management tool designed to facilitate collaboration and efficient project planning. It supports various project management methodologies, including Agile and Waterfall. **Features:** • Task Management: Create, assign, and manage tasks within projects. Tasks can be organized into lists, assigned priorities, due dates, and tracked through completion. • Gantt Charts: Visualize project timelines and dependencies. Gantt charts allow for tracking progress and identifying potential delays. • Roadmap and Milestones: Set up project milestones and roadmaps to track project phases and ensure deadlines are met. • Time Tracking: Log time spent on tasks and projects. This helps in monitoring productivity and project costs. • Bug Tracking: Report, track, and resolve bugs within the project. • Document Management: Store and manage project-related documents. This feature ensures all project documentation is centralized. • Collaboration and Communication: Use the in-built wiki, forums, and messaging to communicate with team members and stakeholders. • Customizable Workflows: Define workflows tailored to the needs of the project. This includes setting up different task statuses and transitions. **Use Cases:** • Project Planning and Tracking: Ideal for setting up

		project timelines, defining tasks, and monitoring progress. • Collaboration: Teams can collaborate on tasks, share documents, and discuss project details. • Resource Management: Manage the workload of team members and ensure optimal allocation of resources.
2.	**Project Libre**	**Project Libre** is an open-source project management software that offers similar functionalities to Microsoft Project. It is widely used for planning, scheduling, and tracking project progress. **Features:** • **Gantt Chart**: Provides a visual representation of project schedules, showing tasks, durations, dependencies, and milestones. • **Task Management**: Allows for creating and managing tasks, setting dependencies, and assigning resources to tasks. • **Resource Management**: Track resource availability and allocation to ensure efficient use of personnel and materials. • **Cost Management**: Calculate and manage project costs, including labour, materials, and other expenses. • **Critical Path Method (CPM)**: Identify the longest stretch of dependent tasks and ensure critical tasks are prioritized to meet project deadlines. • **Network Diagram**: Visualize task dependencies and workflow using a network diagram view. **Use Cases:** • **Scheduling**: Plan project schedules using Gantt charts, ensuring that tasks are completed on time. • **Resource Planning**: Allocate resources efficiently to avoid overloading team members and ensure balanced workloads. • **Cost Control**: Manage project budgets and costs, making it suitable for projects with strict financial constraints.
3.	**Git (Local Version Control)**	**Git** is a distributed version control system widely used for tracking changes in source code during software development. Using Git locally enables version control without needing an

		internet connection or remote server. **Features:** • **Version Control:** Track changes to code over time, allowing for easy rollback to previous versions if needed. • **Branching and Merging:** Create branches for new features or experiments without affecting the main codebase. Merge changes back into the main branch once they are stable. • **Commit History:** Maintain a detailed history of changes, including what was changed, who made the changes, and why. • **Conflict Resolution:** When merging changes from different branches, Git helps in identifying and resolving conflicts. • **Staging Area:** Review changes before committing them to the repository, allowing for selective commits. **Use Cases:** • **Code Collaboration:** Multiple developers can work on the same project simultaneously without overwriting each other's changes. • **Backup and Recovery:** Keep a backup of all changes, making it easier to recover from mistakes. • **Experimentation:** Use branches to experiment with new ideas and features without affecting the main codebase.

S.No	Statical Code Analysis Tool	Programming Language	Description
1.	**Cppcheck**	**C/C++**	**Cppcheck** is an open-source static analysis tool designed for C and C++ code. It focuses on identifying bugs, undefined behaviour, and non-compliance with coding standards. **Key Features:** **Error Detection:** Detects issues like memory leaks, buffer overflows, and null pointer dereferences. **Customizable Checks:** Users can tailor

			checks to their specific coding standards. **Command-Line and GUI:** Available as both a command-line tool and with a graphical user interface. **MISRA Compliance:** Supports checking for compliance with MISRA C/C++ guidelines. **Use Case:** Ideal for C/C++ developers looking for an efficient, easy-to-use static analysis tool to improve code quality and ensure compliance with coding standards.
2.	**Parasoft**	**C/C++**	**Parasoft** is a comprehensive suite of testing and static analysis tools designed to automate software quality and compliance processes. It includes tools for static code analysis, unit testing, and runtime error detection. **Key Features:** **Static Analysis:** Provides deep static analysis for various programming languages, including compliance with safety-critical standards like MISRA, DO-178B, and ISO 26262. **Test Automation:** Supports automated unit testing, integration testing, and continuous testing. **Reporting and Dashboards:** Offers detailed reporting and dashboards to track code quality metrics over time. **Use Case:** Suitable for organizations requiring a robust, enterprise-grade solution for maintaining high code quality, ensuring compliance, and automating testing processes.
3.	**Klocwork**	**C/C++, Python**	**Klocwork** is a static code analysis tool designed for large-scale, safety-critical, and security-sensitive software development. It detects vulnerabilities, memory issues, and coding standard violations.

			Key Features: **Security Vulnerability Detection:** Identifies security issues like buffer overflows, SQL injection, and cross-site scripting (XSS). **Code Review Integration:** Integrates with code review tools and CI/CD pipelines to enforce quality gates. **MISRA and CERT Compliance:** Supports coding standard compliance for safety-critical industries. **Scalability:** Designed to handle large codebases and complex software architectures. **Use Case:** Ideal for enterprise-level development teams working on safety-critical and security-focused applications, particularly in industries like automotive, aerospace, and defense.
4.	**Pylint**	**Python**	**Pylint** is a widely-used static code analysis tool for Python, focusing on improving code quality by enforcing coding standards, detecting bugs, and identifying code smells. **Key Features:** **PEP 8 Compliance:** Enforces Python coding standards, particularly PEP 8. **Error and Warning Detection:** Identifies issues like undefined variables, unused imports, and complex code. **Customizable Rules:** Allows users to configure checks and disable or enable specific warnings. **Code Refactoring Suggestions:** Provides hints for improving code readability and maintainability. **Use Case:** Ideal for Python developers who want to ensure their code adheres to best practices and is free from common mistakes, especially in collaborative or large projects.

5.	**SonarQube**	C/C++, Python	**SonarQube** is an open-source platform that performs continuous inspection of code quality, providing static code analysis, bug detection, code smells, and security vulnerability identification across multiple programming languages. **Key Features:** **Code Quality Metrics**: Tracks code coverage, duplications, complexity, and potential bugs. **Security Vulnerabilities:** Detects and reports on security issues based on OWASP standards. **Extensive Language Support:** Supports more than 25 programming languages, making it a versatile tool for diverse codebases. **Integration with CI/CD:** Seamlessly integrates with CI/CD pipelines to enforce code quality gates. **Dashboards and Reporting:** Provides comprehensive dashboards for tracking code quality over time. **Use Case:** Suitable for organizations looking to maintain high code quality and security across diverse languages and projects, with integration into continuous integration workflows.