Toward an Extensible Quantum Platform-Agnostic Combinatorial Optimization Library

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in the Department of Computer Science

by

José Manuel Ossorio Tróchez

B.Eng. in Software Systems Engineering, Universidad Icesi, Colombia, 2022

We acknowledge and respect the Lək̓ʷəŋən (Songhees and Xʷsepsəm/Esquimalt) Peoples on whose territory the university stands, and the Lək̓ʷəŋən and W̱SÁNEĆ Peoples whose historical relationships with the land continue to this day.

Toward an Extensible Quantum Platform-Agnostic Combinatorial Optimization Library

by

José Manuel Ossorio Tróchez

B.Eng. in Software Systems Engineering, Universidad Icesi, Colombia, 2022

Supervisory Committee:

Dr. Hausi A. Müller, Supervisor
Department of Computer Science, University of Victoria

Dr. Norha M. Villegas, Supervisor
Department of Computer Science, University of Victoria
Department of Computing and Intelligent Systems, Universidad Icesi

Abstract

Combinatorial optimization (CO) problems are computationally challenging as evidenced in various industry and research domains. With recent advances in quantum computing hardware and algorithms, such problems represent an excellent case study for these technologies. Nevertheless, current software tools for CO lack platform-agnostic abstractions to enable researchers and practitioners to utilize quantum resources effectively.

This thesis aims to validate and extend the QPLEX Python library, a platform-agnostic CO package built on DOcplex which integrates execution across multiple quantum providers using various algorithms. We focus on two key software quality attributes: completeness, examining the quantum providers QPLEX supports to look for features that could be added to our library, enhancing its capabilities for handling CO problems; and extensibility, making the library more adaptable for future expansions. We first compile a high-level workflow for solving CO problems to ensure that our elicited software requirements align with the actual process practitioners follow when solving these problems. Subsequently, we evaluate QPLEX through a comprehensive analysis of its completeness by comparing features against alternative solutions including platform-specific SDKs, and its extensibility by examining how easily new features can be integrated without disrupting existing functionality.

Based on the identified functional and non-functional requirements, we design and implement several extensions to QPLEX, including support for Qiskit Runtime Sessions, integration with D-Wave's quantum solvers and implementation of the QAOAnsatz algorithm. Furthermore, we enhance the extensibility of the library through comprehensive documentation, automated testing, and CI/CD pipelines to ensure smooth integration of future open-source contributions.

Validation results demonstrate that these enhancements successfully extend QPLEX's capabilities for solving CO problems using quantum resources, providing a more comprehensive suite of features for quantum-based CO while establishing robust foundations for future development.

This work contributes to the evolving field of quantum software engineering by advancing an abstraction layer that shields practitioners from low-level quantum details, allowing them to focus on problem formulation. As quantum hardware and algorithms continue to advance, such platform-agnostic libraries will play a crucial role in broadening quantum computing adoption, enabling domain experts to leverage quantum resources without requiring deep quantum computing knowledge.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my supervisors Dr. Hausi A. Müller and Dr. Norha M. Villegas for their unconditional support and guidance throughout the journey of completing this thesis; the ideas and reflections that emerged from our conversations played a critical role in shaping this work.

I would also like to thank all the lab members of the Rigi Research Lab, who enriched my thinking with ideas and experiences that have become integral to who I am. Special thanks to Dr. Ulrike Stege for her dedication and guidance; to Tristan, Felipe, Miguel, Priya, Addie, Saasha, and Angadh for their valuable time and efforts; to Samantha for her unwavering support and insightful ideas; and to Juan Fernando for his mentorship and friendship.

Finally, I am deeply grateful to my parents, my sister, Ángela María, and my friends, all of whom carried me through these past three years. I could not have done it without you.

Chapter 1

Introduction

Quantum Computing (QC) is an emerging field of computation that leverages principles of quantum mechanics to perform calculations [1]. It is a multidisciplinary field that uses superposition, interference and entanglement to manipulate quantum bits, or qubits, enabling operations that classical bits cannot perform. Quantum computers can potentially provide a speed up when solving complex problems, such as physics and chemistry simulations, compared to current classical computers [2] [3].

However, despite the potential of QC, current quantum computers are neither reliable nor scalable enough to solve complex, real-world problems faster than classical alternatives [4]. In this context, the scale of a quantum computer refers to the number of logical qubits it possesses, where a logical qubit is an abstract representation of a qubit that is realized using multiple physical qubits. This is done to protect against errors. Complex problems with a large number of variables require a substantial amount of qubits to model them effectively on a quantum computer.

In the case of reliability, present-day quantum computers are highly susceptible to noise, which can be caused by external sources, such as cosmic rays, or internal ones, such as interactions between neighboring qubits. If not handled properly, noise can degrade the accuracy of computations. Thus, advancing quantum error correction is crucial for moving toward fault-tolerant quantum computers.

Consequently, a term commonly used to describe the current state of the QC era is the Noisy Intermediate-Scale Quantum (NISQ) [2] era, where the terms noisy and intermediate-scale refer to the two aforementioned characteristics.

Therefore, quantum computers today are limited in their performance, reducing their applicability to real-world solutions [4]. However, hybrid quantum-classical algorithms have emerged as a promising approach; they have been shown to produce useful results for relevant problems [5][6]. Hybrid algorithms alternate between quantum and classical processing steps to provide a potential speed up compared to purely classical or quantum alternatives [7].

Despite this promise, solving large-scale problems using NISQ-era quantum computers is a challenging task even with hybrid algorithms, and requires suitable software tools that enable scientists and engineers to use quantum resources in the most efficient manner [8]. This is where Quantum Software Engineering (QSE) plays a critical role, as it

focuses on creating models, standards and methods that enable quantum practitioners to develop scalable quantum software systems [9]. As an emerging field, QSE aims to apply knowledge from classical software engineering to adapt existing or create new software development processes, methods, principles and tools for the development of quantum software. This involves classical software engineering knowledge areas such as software design, construction, testing, and maintenance [10][9]. Thus, there is a clear need for tools and methodologies that encompass the complete quantum software toolchain [11].

The development of scalable quantum applications is greatly facilitated by the availability of suitable software tools. Therefore, QC companies often develop their own Software Development Kit (SDK). These SDKs contain software building tools and libraries, alongside the syntactic and semantic rules that programmers must follow to develop quantum software for that platform [12]. Most quantum SDKs are written in Python, which has become the language of choice for quantum computing due to its simplicity and accessibility. Pythons syntax, which differs from the more complex C-style syntax, is particularly beneficial for users who may not come from a computer science background or are new to programming [13]. Its high-level nature allows developers to write readable and concise code, making it easier for them to focus on quantum algorithms rather than implementation details. Additionally, Pythons rich ecosystem of libraries and packages further simplifies the development process, enabling researchers and developers to quickly integrate quantum functionalities with existing software. Given these advantages, Python is well-suited for the needs of the current quantum era, allowing both novice and experienced programmers to engage with quantum development more efficiently.

While Python-based SDKs have simplified quantum software development and increased accessibility, the diversity of available SDKs across different quantum hardware platforms introduces new challenges. Specifically, many SDKs are tied to the quantum hardware developed by the same company, which limits flexibility and forces developers to learn multiple tools to work across different platforms. This makes it harder for newcomers to use different types of quantum hardware. To address this issue, multiple companies design their SDK to be used to write quantum programs that can be executed on various quantum computers from different quantum providers. This type of quantum SDK is sometimes known as platform-inclusive or platform-agnostic [14]; it includes adaptors that facilitate parsing code from one SDK to another. However, these adaptor packages have limited support for the various quantum providers and their unique functionalities.

Given the challenges posed by platform-specific SDKs, our previous work in the field of QSE contributed to enabling quantum and classical practitioners to access different quantum providers more easily in order to solve optimization problems. This contribution, resulting in a previous Master's thesis, consists of a Python programming library called *QPLEX* [15], which acts as a tool for modeling and solving combinatorial optimization (CO) problems. CO is one of the most relevant areas of optimization with practical applications found in every industry, and it is also one of the primary use cases for QC [16][17]. QPLEX's goal is to enable optimization software developers to experiment with different quantum resources and assess performance improvements of hybrid quantum-classical optimization solutions [18] by (1) automatically mapping a problem's QUBO model into a

desired quantum algorithm, (2) handling the parsing of the algorithm to the selected platform's SDK (e.g., IBM's Qiskit[1], D-Wave's Ocean[2], Amazon Braket[3]), and (3) orchestrating its execution and result retrieval.

This thesis builds on top of our previous contribution to validate and extend the QPLEX Python programming library focusing on two software quality attributes: completeness and extensibility. This involves ensuring that QPLEX can handle a wide range of CO problems and quantum platforms, while also making the library more adaptable for future expansions.

1.1 Motivation

Following classical software engineering methods, the QPLEX library was developed to satisfy a set of functional requirements defined in the analysis stage described in [15]. These requirements, which included core functionalities for modeling and solving CO problems, were successfully validated by the original authors. However, further validation is required in two key aspects of the library: completeness and extensibility.

Evaluating completeness involves identifying the current set of features offered by the library and comparing them against alternative solutions to verify that the current system covers all the intended users' objectives [19]. This comparison includes the platform-specific SDKs supported by QPLEX. This assessment will help determine whether the library's features are comprehensive enough to meet the needs of developers working with hybrid quantum-classical optimization methods.

The second area of focus, extensibility, examines the ease with which new features can be added to QPLEX without disrupting existing functionality [20]. Since QPLEX is an open-source project, contributions may come from developers who are unfamiliar with the full codebase. To ensure a smooth integration of new contributions, it is crucial that the library supports extension through modular and well-documented code. Moreover, new code must undergo rigorous testing to confirm compatibility and prevent any regressions before it is merged into the main codebase.

Building on these reflections, this thesis presents a detailed validation of the QPLEX Python programming library, focusing on the two key quality attributes discussed above: completeness and extensibility. In addition to validating the library, the thesis also details the design and implementation of a set of new features that extend QPLEX. The validation proposed in this thesis is intended to guide contributors who wish to extend QPLEX through open-source contributions, ensuring that future modifications align with the librarys design principles and do not disrupt existing functionality. Moreover, the newly introduced features further empower optimization software practitioners, providing them with additional tools to solve complex CO problems effectively.

[1] https://www.ibm.com/quantum/qiskit
[2] https://docs.ocean.dwavesys.com/en/stable/
[3] https://aws.amazon.com/braket/

1.2 Problem Definition and Research Questions

Hybrid quantum-classical algorithms are considered one of the most promising approaches for developing large-scale quantum software applications [21][22]. However, the diversity of quantum providers, the variety of available algorithms, and the steep learning curve associated with different quantum SDKs present significant barriers to implementing these algorithms effectively. This complexity creates a need for tools that can streamline the process of developing hybrid quantum-classical solutions for real-world problems.

In our previous work [18], we introduced the QPLEX library as a step toward addressing these challenges, focusing on combinatorial optimization. However, the quantum computing field is rapidly evolving, with companies continuously deploying new hardware and software. To remain relevant and useful, QPLEX must (1) support the most essential features for solving CO problems, and (2) implement best practices in software engineering, including comprehensive documentation, high test coverage, and robust continuous integration (CI) and continuous deployment (CD) pipelines, known together as CI/CD, to ensure that future contributions and extensions can be integrated seamlessly.

Therefore, we believe that by validating and extending the QPLEX library, we can provide a solid foundation for future open-source contributions. This will not only enhance the library's functionality but also enable swift and efficient development of quantum software, furthering the adoption of hybrid quantum-classical algorithms in solving large-scale problems.

Our research aims to answer the following questions:

RQ1: What are the key components in a workflow for solving combinatorial optimization problems using classical and quantum approaches?

RQ2: How can these workflow insights be applied to analyze and enhance the QPLEX library's completeness and extensibility, ensuring it effectively supports a wide range of quantum providers and facilitates future developments in hybrid quantum-classical optimization?

RQ3: To what extent do the newly implemented features in QPLEX improve its functionality in solving combinatorial optimization problems using quantum resources and enhance its extensibility, facilitating future development?

1.3 Contributions

The following are the contributions of this thesis.

C1: A comprehensive review and analysis of the stages involved in the combinatorial optimization process, condensed into a general high-level workflow, to inform the analysis of QPLEX and the design of new features.

C2: A thorough analysis of the QPLEX librarys completeness and extensibility within the high-level workflow developed in Contribution C1, identifying functional requirements that would enhance the library's effectiveness in solving combinatorial optimization problems (completeness) and non-functional requirements necessary to facilitate future developments in the library (extensibility).

C3: The enhancement and validation of the QPLEX library through the design and implementation of new features based on requirements from Contribution C2, followed by validation through comprehensive testing.

1.4 Research Methodology

To answer RQ1, we will conduct a literature review of the process for solving CO problems, including both classical and hybrid quantum-classical approaches. This review will provide a high-level overview of the CO problem-solving process, outlining its key stages and offering detailed descriptions of each. This analysis will serve as the foundation for identifying which stages of the CO process are best suited for quantum computing, helping us set the stage for the evaluation and extension of the QPLEX library in subsequent steps.

To address RQ2, the methodology is divided into two main parts: completeness analysis and extensibility analysis.

In the first part, we evaluate the completeness of the QPLEX library by analyzing its current set of features for solving CO problems and experimenting with different quantum hardware platforms. QPLEX serves as an abstraction layer over several quantum SDKs (e.g., Qiskit, Ocean, Braket), providing a unified interface for accessing quantum computing resources. However, this abstraction may omit certain features available directly in the underlying SDKs.

To assess completeness, we will:

1. Identify the current features offered by QPLEX, especially those related to formulating and solving CO problems using quantum algorithms
2. Compare QPLEXs features to those provided by the quantum SDKs it integrates with
3. Identify missing functionalities by highlight any key features from these SDKs that are not currently supported by QPLEX but could be valuable additions for CO problem-solving

This part of the analysis will inform our next steps for extending QPLEX to better support a wide range of quantum hardware and ensure that its feature set is comprehensive enough to serve the needs of developers and researchers in the field.

In the second part, we evaluate the extensibility of QPLEX by assessing how easily the library can be adapted or extended by contributors to incorporate new features and support

new quantum platforms. Given that QPLEX is an open-source project, it is crucial to ensure that the codebase is modular, well-documented, and supports smooth integration of new contributions without breaking existing functionality.

To assess extensibility, we will

1. Analyze the modularity and architecture of the QPLEX codebase to determine whether its structure allows for easy addition of new quantum algorithms and SDK integrations

2. Evaluate the documentation: Comprehensive documentation is key for open-source contributions. We will review the current state of QPLEX's documentation to assess whether it provides clear guidelines for developers looking to extend the library.

3. Assess test coverage and CI/CD pipelines: Ensuring that new code can be reliably tested and integrated into the library without introducing bugs is critical. We will evaluate the library's current testing framework and its use of Continuous Integration/Continuous Delivery (CI/CD) pipelines to ensure new contributions are seamlessly integrated. We will also implement additional tests as needed to improve test coverage, particularly around new or less-documented features.

This analysis will help us understand how well QPLEX can accommodate future growth, and we will implement necessary improvements to ensure its long-term adaptability.

To answer RQ3, we will focus on the design, implementation, and validation of new features in QPLEX based on the findings from RQ2. These new features will aim to fill the gaps identified in the completeness analysis and enhance the library's functionality. To validate the implemented features, we will employ a comprehensive testing approach consisting of unit tests and mock tests for each functional requirement. This validation strategy will allow us to (1) verify that each implementation correctly fulfills its corresponding requirement specification, (2) ensure that the features integrate properly with the existing codebase, and (3) confirm that the implementations adhere to the software quality standards established by the non-functional requirements. Our validation will emphasize correctness and completeness of implementation rather than comparative performance evaluation, aligning with the extensibility goals of the project.

1.5 Thesis Outline

The current chapter introduces the motivation, problem definition, and research questions of this thesis, alongside a summary of its contributions. The remaining chapters are structured as follows

Chapter 2: Background and Related Work: This chapter presents the necessary background concepts in CO, open-soure quantum software and the QPLEX library. It

also reviews the state of the art in these fields, laying the groundwork for answering the research questions and framing the subsequent analysis of methodologies and tools.

Chapter 3: A Workflow for Solving Combinatorial Optimization Problems: This chapter provides a high-level overview of the workflow for solving combinatorial optimization problems, from problem analysis and modeling, to algorithm selection, encompassing both classical and quantum approaches. It highlights key stages of the optimization process where quantum computing may offer an advantage, as explored in RQ1.

Chapter 4: Evaluating the Completeness of the QPLEX Library: This chapter evaluates the completeness of the QPLEX library in terms of its support for solving combinatorial optimization problems using different quantum providers. It also elicits a set of functional requirements to enhance the library, addressing part of RQ2.

Chapter 5: Examining the Extensibility of the QPLEX Library: This chapter focuses on assessing the extensibility of the QPLEX library. It identifies areas where the library can be improved in terms of supporting new quantum providers and facilitating future open-source contributions. It also elicits a set of non-functional requirements (e.g., modularity, documentation, testing) to guide these improvements, further addressing RQ2.

Chapter 6: Extending QPLEX: Design and Implementation: This chapter describes the design and implementation of new features based on the functional and non-functional requirements identified in Chapters 4 and 5. These enhancements aim to improve the completeness and extensibility of QPLEX, providing concrete solutions for the gaps and limitations identified earlier.

Chapter 7: Validation and Discussion: This chapter details the validation process for the new features and extensions implemented in QPLEX through comprehensive testing of each functional requirement. It evaluates the correctness and usability of each implementation, discusses the limitations of the validation approach, and examines the implications for QPLEX users.

Chapter 8: Conclusion: This chapter summarizes the contributions of this thesis and reflects on the findings from the research questions. It also provides closing remarks on the impact of this work on quantum software engineering and combinatorial optimization, and discusses potential future directions.

Chapter 2

Background and Related Work

The goal of this chapter is to introduce the foundational concepts relevant to the work presented in this thesis, as well as to present related work. While not intended to be a comprehensive review of the topics mentioned, it aims to cover the essential concepts required for a thorough understanding of the subsequent work.

2.1 Combinatorial Optimization

Combinatorial Optimization involves finding an optimal solution to a problem from a finite set of solutions [23]. In a combinatorial Optimization problem, a set of discrete variables can be combined in various ways to form a solution. The quality of a solution is determined by a cost function specific to the problem, also known as the objective function. This function maps a combination of variables to a value, often represented by a real number. The goal of CO is to find the configuration of variables that optimizes the objective function. As the number of variables increases, the number of possible combinations often grows exponentially, depending on the problem's structure, leading to increased complexity.

CO problems are of interest to researchers and practitioners because they are present in many different fields such as finance, logistics, healthcare, and others. Moreover, many of these problems can be abstracted into general CO formulations, enabling researchers to focus on solving abstract problems while addressing concrete real-life scenarios [24].

As explored in more detail in Chapter 3, a CO problem typically consists of several key components. First, the decision variables define the possible configurations that can be selected; these may be binary, integer, or discrete values, depending on the problem context. Second, the objective function, which evaluates the quality of any given configuration by mapping it to a numerical value. Third, the solution space, which comprises all possible combinations of variable values. And finally, the constraints, which are part of many real-world CO problems and restrict the set of feasible solutions by imposing limits on what configurations are allowed.

Due to their discrete nature and combinatorial explosion of possibilities, many CO problems are classified as NP-hard, meaning they cannot be solved efficiently (in polynomial time) using classical algorithms as the problem size increases. This computational challenge has motivated the exploration of quantum computing approaches, which may offer

advantages for certain problem classes through quantum phenomena such as superposition and entanglement.

2.2 Open-Source Quantum Software Ecosystem

2.2.1 Current Landscape

The ecosystem of QC software has seen significant growth in open-source software development, resulting in a diverse set of tools, libraries and frameworks. This ecosystem spans the entire quantum software stack, from low-level hardware interfaces to high-level application libraries [11]. Major quantum providers such as IBM, D-Wave and Xanadu have released open-source SDKs to facilitate access to quantum hardware.

These open-source projects serve multiple key functions in the emerging QC field. They facilitate access to quantum resources, enabling researchers, educators, and developers without direct hardware access to experiment with quantum algorithms. They also accelerate innovation by allowing the community to contribute new features and integrations, driving the field forward at a pace that few single organizations could achieve alone.

2.2.2 Challenges in Quantum Open-Source Development

Despite these benefits, developing open-source quantum software presents challenges. The rapid evolution of quantum hardware and theoretical approaches means that software must be developed alongside hardware and be highly adaptable to incorporate new advancements. Additionally, the interdisciplinary nature of QC requires collaboration between experts from various fields, including physics, computer science, mathematics, and engineering, each bringing different perspectives and requirements.

The high diversity of quantum hardware implementations and quantum providers has limited standardization across different quantum platforms, with each provider developing their own frameworks and abstractions. This fragmentation increases the learning curve for newcomers and motivates the development of platform-agnostic toolsa gap that libraries like QPLEX aim to address.

2.3 The QPLEX Library

2.3.1 Purpose and Design Goals

The development of the QPLEX library was motivated by the need for abstraction layers in QSE. Such layers help minimize the impact of low-level quantum hardware and software details, leading to swifter incorporation of QC into current optimization pipelines. This goal has two different avenues: (1) in research, to streamline experimentation, and

(2) in industry, to facilitate the adoption of QC for CO. QPLEX aims to accomplish these goals by providing a simple interface to model and solve CO problems on various quantum platforms, and incorporating an extensible repertoire of algorithms [15], [18].

QPLEX can be categorized as a platform-agnostic combinatorial optimization library that provides access to both classical and quantum platforms. The library uses a well-known CO package as its foundation to provide problem modeling capabilities through an established and familiar interface: the DOcplex API [25]. With this foundation in place, the library was designed in a modularized manner, such that cohesively separate components remain decoupled in the codebase. These modules include the `Solvers` module, which implements different solvers for various quantum providers. Each solver encapsulates the logic to communicate with a quantum provider and solve a CO model, abstracting away provider-specific implementations from the user. By utilizing these different solvers, the initial problem model can be sent to the chosen quantum provider, and the results of the execution can be retrieved and displayed to the user without requiring any knowledge of quantum algorithms or specific provider SDKs.

2.3.2 Architecture Overview

The architecture of QPLEX prior to this work, as seen in Figure 2.1, consisted of four primary modules that work together to provide a cohesive solution for quantum optimization problems:

- **Model Module**: Contains the fundamental `QModel` class, which extends the underlying DOcplex package and serves as the entry point for modeling and solving CO problems with QPLEX. This module orchestrates the entire problem-solving workflow by leveraging the functionality provided by the other modules. The QModel class allows users to define their optimization problems using the familiar DOcplex syntax while adding quantum execution capabilities.

- **Commons Module**: Integrates utility classes and functions that are used throughout the problem-solving workflow. A key component is the `SolverFactory` class, which implements the Factory design pattern to instantiate the appropriate solver based on the user's selection. This module facilitates loose coupling between the other components of the library, enhancing maintainability and extensibility.

- **Solvers Module**: Contains provider-specific solver classes that all implement a common interface. Each solver is responsible for three critical tasks: (1) parsing the base problem model into a format compatible with the provider's SDK, (2) orchestrating the execution of the model on the selected quantum backend, and (3) parsing the results returned from the quantum device into a standardized format. This abstraction layer shields users from the complexities of different quantum platforms.

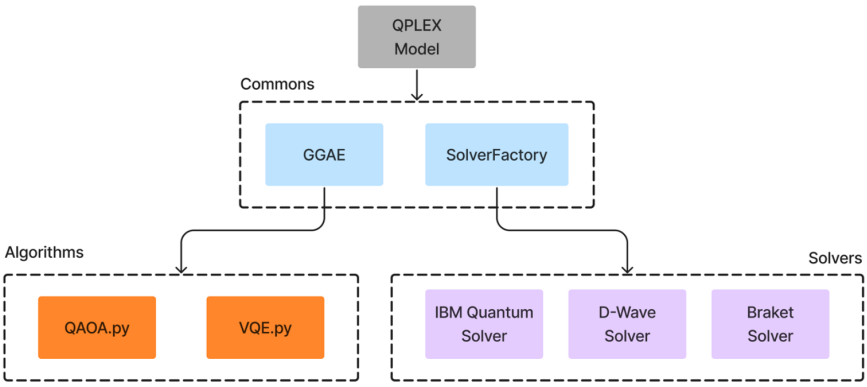

Figure 2.1|: System design of QPLEX prior to this work. Taken from [15].

- **Algorithms Module**: Includes a repository of quantum algorithm implementations that can embed a `QModel` into the algorithm's circuit structure before solver execution. These algorithms translate the mathematical representation of the optimization problem into quantum operations that can be executed on the selected quantum hardware.

This modular architecture was designed to facilitate extensibility, allowing for the addition of new algorithms and quantum providers with minimal changes to the existing codebase. The separation of concerns between modules ensures that modifications to one component do not necessitate changes to others, promoting a robust and maintainable software structure.

It is worth mentioning that the main extension points of the library are the Solvers module and the Algorithms module, offering interfaces for future solvers and algorithms respectively, and making these additions less complex than would be possible with a more coupled architecture.

2.3.3 Workflow for Solving Combinatorial Optimization Problems with QPLEX

The workflow for solving CO problems with QPLEX follows a process that abstracts away much of the complexity involved in quantum execution. Users can solve optimization problems in the following steps:

First, users create a `QModel` instance to formulate their optimization problem using the DOcplex API, which provides a familiar mathematical modeling interface. This involves defining decision variables, constructing an objective function, and specifying any constraints. The resulting model represents the problem in a mathematical form that can be processed by QPLEX.

Finally, users invoke the `solve` method on the `QModel` instance, specifying execution parameters such as the desired quantum provider, algorithm, and any provider-specific options. The `solve` method then:

1. Uses the `SolverFactory` to instantiate the appropriate solver for the selected quantum provider
2. Retrieves the specified algorithm from the Algorithms module (in the case of D-Wave, this step is skipped)
3. Delegates the execution to either the Generalized Gate-based Algorithm Execution workflow or the D-Wave solver
4. Uses the corresponding solver to retrieve and process the results from the quantum execution
5. Returns the solution to the user in a standard format

Users can experiment with different quantum providers or algorithms by simply changing parameters in the `solve` method call, without needing to modify their problem formulation or understand the underlying quantum implementations.

For advanced users who need more control, QPLEX also provides options to customize execution parameters, specify algorithm-specific settings, and configure solver behavior. This flexibility allows researchers to experiment with novel approaches while still benefiting from the abstraction layer that QPLEX provides.

2.4 Related Work

This section presents QPLEX in the broader landscape of quantum optimization tools and provide a future outlook for the field of quantum software.

2.4.1 Additional Open-source Quantum Optimization Libraries

To the best of our knowledge, no other tool currently offers the same combination of modeling capabilities and execution flexibility across multiple quantum providers as QPLEX. However, some related efforts exist in the quantum optimization space that are worth noting.

The OpenQAOA [26] python package developed by Entropica Labs provides an implementation of the QAOA algorithm with support for different problem types. While OpenQAOA offers abstraction layers for QAOA-specific applications and supports execution on various quantum providers including IBM, Google and Rigetti, it does not provide the same breadth of optimization capabilities or the familiar DOcplex modeling interface that QPLEX offers. OpenQAOA focuses specifically on the QAOA algorithm and its variants, whereas

QPLEX aims to be a more general platform for quantum optimization across multiple algorithms and providers.

Another relevant library is Qiskit Optimization,[1] now a separate library which was originally a module of Qiskit. Also referred to as a framework, Qiskit Optimization provides tools for problem modeling and algorithm execution that resemble the capabilities of QPLEX. It also supports translating optimization problems into various formulations. While it offers excellent integration with IBM's quantum hardware and simulators through Qiskit, it lacks the cross-platform capabilities that QPLEX provides. QPLEX enhances this flexibility by abstracting away the provider-specific details and offering a unified interface for executing optimization problems across different quantum platforms out of the box, not limited to IBM's ecosystem.

2.4.2 Future Directions in Quantum Software Engineering

In the commercial space, IBM has outlined a vision for their quantum software stack that aligns with the goals of QPLEX. Their roadmap includes the development of high-level programming platforms that abstract away low-level quantum details and focus on applications. Figure 2.2 shows IBM's quantum software stack roadmap, which illustrates their intention to build layers of abstraction from hardware-specific details up to domain-specific applications. This approach mirrors QPLEX's philosophy of providing accessible abstractions for specific problem domains while shielding users from the complexities of the underlying quantum implementations.

Standardization efforts represent another important direction in quantum software engineering. As the field matures, the need for common interfaces, protocols, and benchmarks becomes increasingly apparent [9]. The Quantum Intermediate Representation (QIR) Alliance,[2] for instance, is developing a common intermediate representation that allows users to write fully interoperable quantum programs.

Such efforts align with QPLEX's goal of providing a unified interface for quantum optimization across different providers. As standards emerge and gain adoption, libraries like QPLEX will play a crucial role in implementing these standards and making them accessible to practitioners. The modular architecture of QPLEX, with clear separation between problem modeling, algorithm implementation, and hardware execution, positions it well to adapt to and incorporate emerging standards in QSE.

The ongoing development of higher-level abstractions and standardization efforts point to a future where quantum resources become more accessible to domain experts without requiring detailed knowledge of quantum mechanics or specific hardware architectures.

[1] https://qiskit-community.github.io/qiskit-optimization/
[2] https://www.qir-alliance.org/

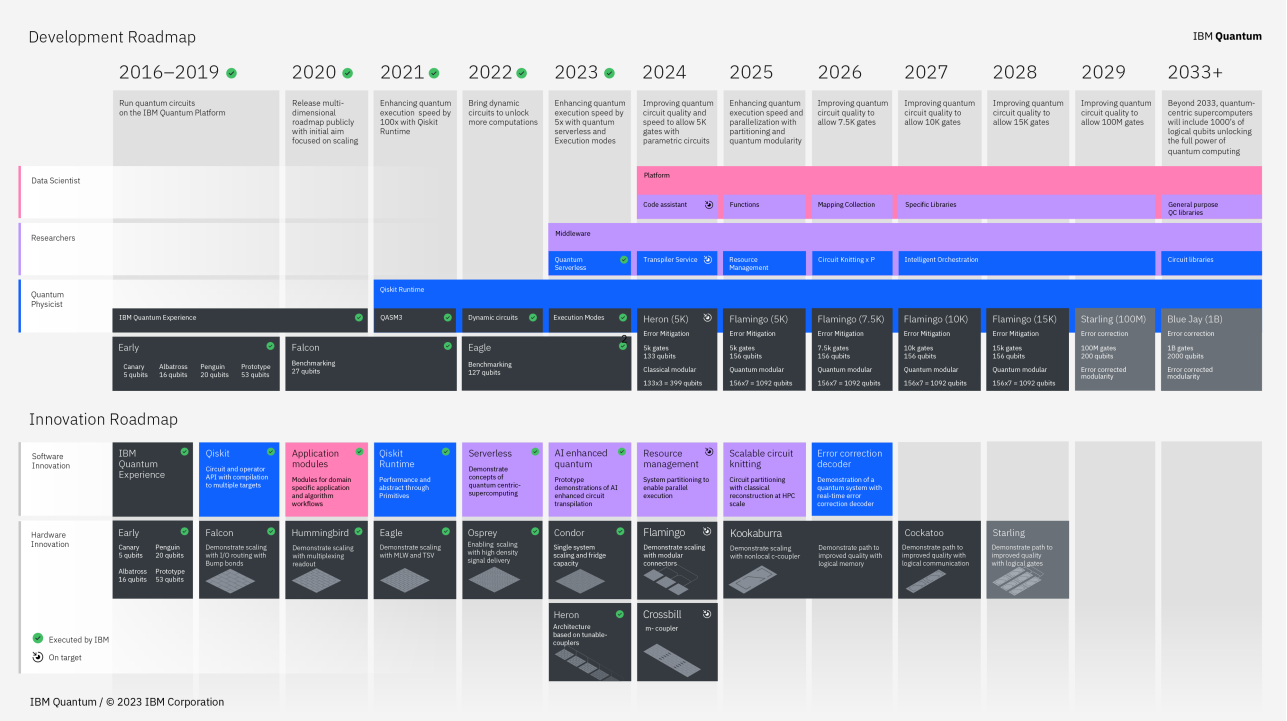

Figure 2.2|: IBM Quantum Development Roadmap to 2033. Taken from [27].

2.5 Chapter Summary

This chapter introduced concepts and tools that aid in the understanding of this thesis, such as CO, the QPLEX programming library, and the current landscape of open-source quantum software, including relevant related work. At the time of writing, there are significant challenges in building software tools to help developers and researchers make use of the full potential of quantum computers. The rapid evolution of quantum hardware and theoretical approaches combined with the interdisciplinary nature of QC are hurdles in the field of QSE. Nevertheless, useful tools that abstract low-level details and enable practitioners to solve problems have emerged, with QPLEX serving as a platform-agnostic library that bridges the gap between different quantum providers and simplifies the process of solving CO problems. The Development Roadmap of IBM further emphasizes the need for such abstraction layers, aligning with QPLEX's approach of shielding users from the complexities of underlying quantum implementations.

Chapter 3

A Workflow for Solving Combinatorial Optimization Problems

Building on the foundational understanding of CO from the previous chapter, this chapter addresses our first research question by developing a high-level workflow for solving CO problems. By breaking down the CO problem-solving pipeline into distinct stages and examining the key considerations at each step, we provide a structured approach that can guide practitioners through the task of solving CO problems. This workflow not only clarifies the fundamental stages of CO problem-solving but also creates a framework for understanding where and how different tools and technologies, including quantum computing resources, can enhance the problem-solving process.

3.1 Introduction

The process of solving CO problems requires a structured and systematic approach, presented here as a general workflow. The intrinsic complexity of many real-world CO problems, combined with their discrete nature and the vast number of possible solutions, demand such an approach to efficiently explore solution spaces and identify optimal or near-optimal outcomes. A clear workflow ensures that these problems are tackled in a methodical manner, enabling the identification of a proper solution through a well-informed choice of solution techniques.

As discussed previously in 2.1, CO problems are of significant interest to researchers and practitioners across various fields, from logistics to finance to biology. This interest, combined with the inherent difficulty of CO problems, establishes the field as a promising area for the development of new methodologiesboth classical and quantumthat aim to provide better solutions or improve efficiency over existing methods. In the field of QC, CO plays a crucial role, driving the development of new quantum algorithms and techniques that push current quantum hardware to its limits [28].

In the following sections, we explore each step of the proposed workflow in detail. Our proposed workflow consists of four main stages:

1. Problem Analysis
2. Problem Abstraction and Modeling
3. Algorithm Selection
4. Solution and Iteration

This structure synthesizes insights from multiple sources: the theoretical framework presented by Yarkoni et al. [29], empirical findings from quantum annealing applications studied by Heng et al. [30], practical implementation strategies from recent optimization studies [31], [32], and our own research experiences. While this workflow builds on established approaches, it specifically considers how quantum computing resources can be integrated into the CO solution process, identifying stages where quantum approaches might offer advantages over classical methods.

The proposed workflow stages are not intended to represent a rigid or exhaustive framework but rather serve as a flexible guide to navigate the CO process. By synthesizing established literature with practical experience, this workflow aims to provide clarity and direction for both classical and quantum methodologies, while remaining adaptable to specific problem contexts and advances in the field.

3.2 Problem Analysis

The problem analysis stage is fundamental to the success of the CO process; it involves identifying and laying out the components of the problem, described in Chapter 2. A deep understanding of the problem at this stage allows us to choose the appropriate abstraction for our problem, as we will see in 3.3, and ultimately guides the entire solution process.

Mathematically, a CO problem is typically defined as follows:

Given a finite set of feasible solutions S and a cost function $f : S \to \mathbb{R}$ that maps each feasible solution to a real number. The goal is to find a solution $s^* \in S$ that minimizes (or maximizes) the cost function f:

$$s^* = \arg\min_{s \in S} f(s) \quad (\text{or } \arg\max_{s \in S} f(s)) \tag{3.1}$$

In other words, analyzing a CO problem involves identifying the variables that define the problem and understanding how they combine to form feasible solutions. These variables represent the key decision points in the problem, and their interactions determine the space of possible solutions. Often, the problem's structure is described qualitatively rather than explicitly, requiring us to extract meaningful insights from such descriptions, as highlighted by Lee et al. in [33]. Furthermore, determining the size of the solution space is rarely straightforward. However, as mentioned earlier, many CO problems can

be abstracted into well-known formulations, for which the size of the solution set can be computed using established results.

Once the variables are identified, the set of feasible solutions alone is not meaningful without a way to evaluate them. This brings us to the objective function, or cost function, which interprets specific configurations of the variables and assigns a numerical value to each solution. The objective function plays a critical role in guiding the optimization process by indicating which solutions are more desirable. It captures the interactions between variables and provides a way to compare different solutions. Identifying the correct objective function is fundamental, as it defines what constitutes an optimal solution. Overlooking important interactions in our problem may cause the algorithm to yield sub-optimal results.

Let us illustrate this first step of the process with an example, paraphrased from Cook et al. in [34]: An oil company has a field consisting of 47 different platforms, which all need to be visited once by a helicopter. Given that operating the vehicle is expensive, the company is interested in finding the path between all platforms that minimizes the distance traveled. Note that the helicopter needs to return to the starting point after finishing its trip.

In this example we have a description of the set of variables of our problem, from which we can derive the set of feasible solutions. Here, the variables are the paths between platforms, the former also known as edges and the latter as nodes, and a feasible solution corresponds to a distinct set of edges that comprise a complete trip. Furthermore, a given order for visiting the platforms requires the helicopter to travel a specific distance. This distance represents the cost of the solution, which we aim to minimize.

Lastly, an essential component to identify in our problem is the set of constraints. Constraints restrict the space of feasible solutions by imposing limits on what configurations are allowed. In our example, constraints could involve limited resources or time windows, which would increase the complexity of the problem by reducing flexibility and narrowing the search space.

Moreover, in some cases, certain constraints are essential to ensure that the mathematical model aligns with real-world behavior. These constraints help capture practical requirements that the objective function or algorithm alone might overlook. For instance, in the example from earlier, additional constraints are necessary to prevent the formation of multiple disconnected loops and ensure that each platform is visited exactly once, with only one incoming and one outgoing path per platform. Without such constraints, the solution would not reflect the real structure of the problem.

At this point, we have established the scope of the feasible solutions, identified our cost function, and recognized the need for constraints. In theory, solving a CO problem could be reduced to listing all possible solutions and evaluating their costs to find the optimal one. However, this brute-force approach quickly becomes impractical due to the combinatorial explosion of possibilities, making it infeasible for all but the smallest problems. This

challenge underscores the need for efficient algorithms that can strategically navigate the solution space and find optimal or near-optimal solutions within reasonable time.

With this foundation in place, we can now proceed to the next step: problem abstraction and modeling, where we refine the problem structure and prepare it for algorithmic exploration.

3.3 Problem Abstraction and Modeling

By making general observations about the problem, as outlined in the previous section, we gain valuable insight into its structure and components. However, to solve the problem effectively, these insights must be captured within a mathematical model. Mathematical models help us understand complex systems underlying empirical observations [35], and are essential for computation, as computers operate exclusively with numerical representations. Modeling serves as a bridge between understanding the problem conceptually and implementing algorithms that can efficiently explore and solve it.

As mentioned in Section 2.1, CO problems can often be abstracted into well-known problem types. This is advantageous because many of these problems already have established formulations, which simplifies the modeling process and makes it more efficient. Therefore, identifying the appropriate abstraction for a concrete problem is a crucial step that precedes mathematical modeling, and it is fundamental for continuing with the CO problem-solving pipeline. This step, however, is non-trivial and requires both experience in mapping real-world problems and knowledge of common CO problems.

Building on the earlier oil platforms example from 3.2, the problem can be identified as a Traveling Salesman Problem [36] (TSP) because it meets the following criteria:

- It has a set of nodes (e.g., platforms, cities) that must be visited exactly once.
- There are paths with associated costs or distances between nodes.
- The objective is to minimize the total travel cost (or distance) while returning to the starting node after visiting all others.
- There are no constraints on order, aside from visiting each node exactly once.

In this example, each platform corresponds to a node, the helicopter must visit all nodes exactly once, and the objective is to minimize the total travel distance.

Thus, we can express the problem mathematically by defining its objective function as follows:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} \cdot x_{ij} \tag{3.2}$$

where

- $x_{ij} \in \{0, 1\}$ is a binary variable indicating whether the path from platform i to j is part of the tour (1) or not (0).
- d_{ij} is the distance between platform i and platform j.
- The goal is to minimize the total distance traveled.
- n is equal to 47 in this example.

This formulation sums the distances d_{ij} weighted by the decision variables x_{ij}. Only those paths where $x_{ij} = 1$ contribute to the total cost, ensuring the correct paths are selected to minimize the travel distance.

It is worth noting, however, that this cost function alone is not enough to represent our problem effectively since it does not capture certain limitations. As discussed earlier in Section 3.2, constraints are often necessary to prevent the algorithm from finding infeasible solutions. In this case, one example of an infeasible solution that minimizes the cost function is the trivial one where no platforms are visited at all, resulting in a total distance of zero. Naturally, this is an undesirable outcome, so we must introduce constraints to prevent this, as we will see further down in this section.

In our example, the decision variables are represented by the binary terms x_{ij}, which indicate whether a path between two platforms is included in the vehicle's trip. However, to ensure the solution forms a valid tour, we need to introduce additional constraints. Specifically, we must prevent the formation of multiple disconnected loops (subtours) and ensure that each platform has exactly one incoming and one outgoing path. This type of formulation is known as an Integer Programming (IP) formulation, where decision variables are represented using integers and constrains are linear [37].

To achieve this, we first introduce auxiliary variables u_i to represent the order in which platforms are visited. These variables will be used in some of our constraints:

$u_i \in \mathbb{Z}$: A continuous variable representing the position of platform i in the sequence (only used for nodes $i > 1$).

We now define the necessary constraints:

1. Flow Constraints: Ensure each platform has exactly one outgoing and one incoming path.

$$\sum_{\substack{j=1 \\ j \neq i}}^{n} x_{ij} = 1 \quad \forall i \in 1, \ldots, n \tag{3.3}$$

$$\sum_{\substack{i=1 \\ i \neq j}}^{n} x_{ij} = 1 \quad \forall j \in 1, \ldots, n \tag{3.4}$$

2. Miller-Tucker-Zemlin (MTZ) Subtour Elimination Constraints [38]: Prevent smaller loops (subtours) from forming.

$$u_i - u_j + n \cdot x_{ij} \leq n - 1 \quad \forall i \neq j,\, i, j \in \{2, \ldots, n\} \tag{3.5}$$

$$1 \leq u_i \leq n - 1 \quad \forall i \in \{2, \ldots, n\} \tag{3.6}$$

It is worth noting that, so far, we have considered the edges between platforms, meaning the path that the helicopter will take, as the variables in our problem. For simplicity, we have intentionally guided the formulation toward a graph-based IP model to illustrate the core concepts of this chapter. However, there are other ways to formulate the TSP that may be better suited for certain problem instances [39].

For instance, in some cases, the variables represent the platforms themselves, rather than the edges between them. For example, in a permutation-based formulation, the cost function evaluates different permutations of platformseach representing a possible order of visits. The edges are represented implicitly through the sequence of platforms. This approach can be particularly advantageous when dealing with time-dependent constraints or precedence relationships. Consider a delivery service operating under strict scheduling constraints: each location must be visited within a specific time window, and certain locations must be visited before others (e.g., picking up from a restaurant before delivering to a customer). In this case, a permutation-based formulation, where variables directly represent the position of each location in the sequence, makes these constraints more natural to express and efficient to evaluate compared to an edge-based formulation.

In our helicopter example, however, we chose to represent the platforms implicitly by focusing on the edges as decision variables, as this formulation better aligns with the primary objective of minimizing total travel distance without additional temporal or precedence constraints.

To summarize, the problem analysis and problem modeling stages are closely related, as the choice of variables directly shapes the structure of the cost function. The way we define variableswhether as edges or nodesdetermines how the problem is formulated and influences the modeling approach, ultimately guiding the selection of algorithms to solve it efficiently. As we will see in the following section, certain formulations align better with specific algorithms. Thus, comprehensive knowledge of available toolssuch as problem formulations and algorithmsenhances the effectiveness of the CO problem-solving process.

3.4 Algorithm Selection

Since many CO problems belong to the NP-hard complexity class, obtaining an exact solution within polynomially bounded computation time is generally not feasible [40]. In most cases, a good approximation suffices, as producing an approximate solution is significantly less expensive in terms of execution time and computational resources. Given the

vast dimensionality of the search space in these problems, approximation is often the only practical approach [41].

Consequently, CO algorithms are broadly classified into exact and non-exact (heuristic) algorithms [40], with metaheuristics providing a higher-level framework to robustly guide the search process [42]. Furthermore, recent advances in quantum and quantum-inspired algorithms offer promising alternatives by leveraging novel computing paradigms. Selecting the right algorithm depends on factors such as problem size, constraints, structure, time sensitivity, and whether an approximate solution is sufficient or an exact result is needed. In the following subsections, we explore each of these categories in greater detail.

3.4.1 Exact Algorithms

Exact algorithms guarantee finding the optimal solution for a CO problem by exhaustively exploring the entire search space or using techniques to reduce it, such as branch-and-bound algorithms [43]. They provide a provable upper and lower bound on the optimal objective value, ensuring precise results. Some other exact approaches include dynamic programming [44] and branch-and-cut algorithms [45], which further improve efficiency by pruning unpromising branches.

However, the runtime of exact algorithms tends to grow exponentially as the problem size increases [43]. This makes them impractical for large problem instances, especially in real-time applications. As a result, they are often used on smaller problem subsets to evaluate heuristic solutions. Knowing the optimal solution for a small version of the problem allows us to benchmark heuristics and metaheuristics, ensuring their solutions are close to the true optimum. This is essential to determine the optimality gapthe difference between a heuristic solution and the optimal one.

Additionally, exact algorithms play a crucial role in hybrid quantum-classical approaches, where they are used to solve subproblems within a larger heuristic framework. For example, in branch-and-price algorithms, exact algorithms are employed to optimize portions of the problem iteratively. Despite their limitations, exact methods remain indispensable in applications where accuracy and verification are critical, such as finance or production planning. In such cases, they are employed strategically to solve a subset of the problem or a simplified model, generating solutions that offer valuable insights into the problem's structure.

3.4.2 Heuristics

On the other hand, heuristic algorithms aim to find good approximations rather than exact solutions. They rely on intelligent techniques inspired by human intuition and analogies to real-world processes [46]. These methods trade optimality for speed, enabling them to handle larger instances efficiently. The goal is to produce solutions with a sufficiently small

optimality gap, validating the effectiveness of heuristics in real-world scenarios where exact solutions are impractical.

An example of a heuristic algorithm relevant to our TSP example is the Nearest Neighbor Algorithm [47]. In the context of our example, the algorithm begins at a given platform, selects the nearest unvisited one as the next stop, and continues this process until all platforms are visited, finally returning to the starting point. While fast, this method is prone to getting trapped in local optima, since it chooses the nearest neighbor without considering the impact of this decision on the overall path. In other words, it lacks a broader view of the solution space that could lead to a better path. This limitation illustrates the trade-off between optimality and speed.

In summary, heuristics are powerful and scalable techniques that can be applied to large problem instances and produce results within a feasible amount of time. However, they do not guarantee optimality and can get stuck in local optima. As we will see in the next section, metaheuristics address some of these limitations by providing a more strategic exploration of the solution space.

3.4.3 Metaheuristics

Metaheuristics, such as Genetic Algorithms and Simulated Annealing (SA) [48], build on heuristics by making the exploration of solution spaces more efficient. As the name suggests, metaheuristics operate at a higher abstraction level than standard heuristics, offering a framework for guiding search strategies. These methods provide flexible templates that can be adapted to various problems, helping to balance exploration and exploitation [49].

Moreover, metaheuristic algorithms reduce the risk of getting stuck in local optima by introducing mechanisms such as randomization in the case of SA and memory structures as in the Tabu Search [50] algorithm.

The performance of metaheuristics, as with other heuristic algorithms, depends heavily on the chosen hyperparameters for the algorithm. For example, the cooling schedule in SA and the length of the tabu list in Tabu Search can significantly influence solution quality. Thus, experimentation and fine-tuning are often required to adapt the hyperparameters to specific problem instances.

In conclusion, metaheuristics are powerful and versatile methods for improving the performance of heuristic algorithms, making them suitable for solving complex optimization problems where exact methods are impractical.

3.4.4 Quantum Algorithms

So far, all the algorithms mentioned have been purely classical, meaning they rely exclusively on classical hardware for computation. However, as discussed in Chapter 1, quantum

algorithms are emerging as promising tools for solving CO problems; they offer potential advantages over classical methods by leveraging quantum properties to explore solution spaces more efficiently. Notably, the most prominent quantum algorithms for CO problems are approximate algorithms, such as Quantum Annealing [51] (QA) and the Quantum Approximate Optimization Algorithm [6] (QAOA). Moreover, many of these algorithms, including QAOA but not QA, fall under the category of variational quantum algorithms (VQAs)a class of hybrid quantum-classical algorithms that utilize classical resources to optimize quantum circuits [52].

These quantum techniques are at the forefront of CO research, partly due to their ability to take advantage of representing many relevant CO problems as Quadratic Unconstrained Binary Optimization (QUBO) problems. The QUBO formulation provides a convenient structure for both quantum annealers and QAOA circuits. For QA, QUBO problems align naturally with the hardwares energy minimization process by encoding problem variables into qubits, allowing the quantum system to search for the minimum energy state. For QAOA, the QUBO formulation ensures that the problem can be expressed as a cost Hamiltonian, which guides the quantum circuit toward optimal or near-optimal solutions through iterative parameter updates.

Nevertheless, given the limitations of current quantum hardware, the applicability of quantum algorithms for solving CO problems remains limited. Large-scale problems with constraints are difficult to address using todays quantum computers. This limitation is one reason why the most prominent quantum algorithms are hybrid approaches, using quantum hardware to solve only the combinatorial part of the problem that is challenging for classical methods, while relying on classical hardware for parameter optimization.

The field of quantum algorithms for CO is still in its early stages, and new approaches are actively being explored. Future algorithms that build on existing ones or adopt novel strategies hold the potential for better results. Furthermore, as quantum hardware continues to improve [53], the size and complexity of CO problems that can be tackled using quantum methods will also increase, paving the way for more practical applications.

3.4.5 Selecting an Algorithm

The Algorithm Selection Problem is a fundamental challenge in CO: determining the most suitable algorithm for solving a specific problem instance. This meta-level decision process aims to identify the algorithm that will perform best for each unique case, recognizing that no single algorithm consistently outperforms all others across different problem instances [54].

Selecting an appropriate algorithm depends on several interconnected factors. Problem size and complexity play a central role, as they directly affect computational feasibility. Available computational resources, such as processing power, memory, and parallelization capabilities, further constrain the choice of algorithm. Time constraints are particularly important in real-time applications, where solution speed may take precedence over optimality. Additionally, the required precision of the solution influences algorithm selection:

exact methods are indispensable in domains like finance, healthcare, and scheduling, where precision is critical, while heuristics and metaheuristics are better suited for large-scale or time-sensitive problems where near-optimal solutions suffice.

In practice, it is common to evaluate multiple algorithms for a given problem instance, comparing their performance to identify the most effective approach.

Problem-specific characteristics play a crucial role in algorithm selection. These include the components of a CO problem introduced in Chapter 2 and detailed in Section 3.2, such as the structure of the feasible set, the objective function, the types of constraints present, and the characteristics of decision variables. Thus, understanding these components by performing a thorough problem analysis enables more informed algorithm selection decisions.

The implementation of systematic algorithm selection approaches has demonstrated substantial performance improvements in combinatorial optimization [54]. However, this benefit must be balanced against the computational overhead of the selection process itself. An effective algorithm selector should be computationally inexpensive while maintaining reliable selection accuracy.

Performance assessment criteria play a vital role in comparing algorithms and validating selection decisions. Key metrics typically include solution quality, computational time, memory usage, and solution stability. For approximate methods, the evaluation must also consider the trade-off between solution quality and computational effort.

The field of algorithm selection continues to evolve, particularly with advances in machine learning and automated algorithm configuration. These developments promise more sophisticated selection mechanisms that can adapt to changing problem characteristics and computational environments.

3.5 Solution and Iteration

When solving real-world CO problems, approximation techniques are often the only feasible approach, as discussed previously. In this context, where results can vary between algorithm runs and hyperparameters significantly influence solution quality, systematic experimentation becomes crucial for achieving approximations with acceptable optimality gaps.

Experimentation is carried out as an iterative process, where baseline solutions are established using standard hyperparameter configurations and algorithms that are commonly used for the problem in hand. The purpose of these solutions is to serve as benchmarks for measuring improvements of subsequent configurations. At this stage, metrics such as solution quality, computational time, and resource utilization are tracked to evaluate performance.

As mentioned previously, hyperparameters can have a substantial impact on the performance of an algorithm. In the case of metaheuristics, multiple hyperparameters may interact in complex ways, making manual tuning impractical. Thus, there exist automated approaches for exploring the hyperparamter space to identify promising configurations. However, the computational cost of this meta optimization problem should be balanced against its potential benefits.

Furthermore, it is common to apply multiple algorithmic approaches to the same problem, as there is always potential for improvement in approximation techniques. Researchers and practitioners often experiment with different algorithms in order to enhance solution quality or improve performance. This emphasizes the need for modern software tools for solving CO problems. Such tools could benefit from built-in experimentation capabilities to streamline the experimentation process, including:

- Automated logging and result tracking
- Statistical analysis of solution quality
- Visualization of solution convergence
- Parameter sensitivity analysis

In summary, while producing solutions and iterating over different algorithm configurations may seem mechanical, it demands a solid understanding of experimental design principles, optimization techniques, and modern software tools. The iterative nature of this whole process can lead to valuable insights about the problem structure and the effectiveness of different solution strategies, contributing to the field of CO.

3.6 Chapter Summary and Discussion

This chapter presents a high-level workflow for solving CO problems, breaking down the process into distinct stages: problem analysis, abstraction and modeling, algorithm selection, and solution and iteration. Each stage was explored in detail to provide newcomers with a structured approach to solving CO problems effectively. Figure 3.1 illustrates the processes that are part of the workflow, each with its input and output.

It is worth elaborating on the components of each input/output of the proposed workflow:

- **Problem Description:** As discussed in Section 3.2, the problem description is typically derived by the practitioner or researcher based on real-world observations. This description must provide sufficient information about all relevant entities and interactions in the problem scenario to enable further analysis.
- **Problem component specification:** The key components of a CO problem should address the following questions:

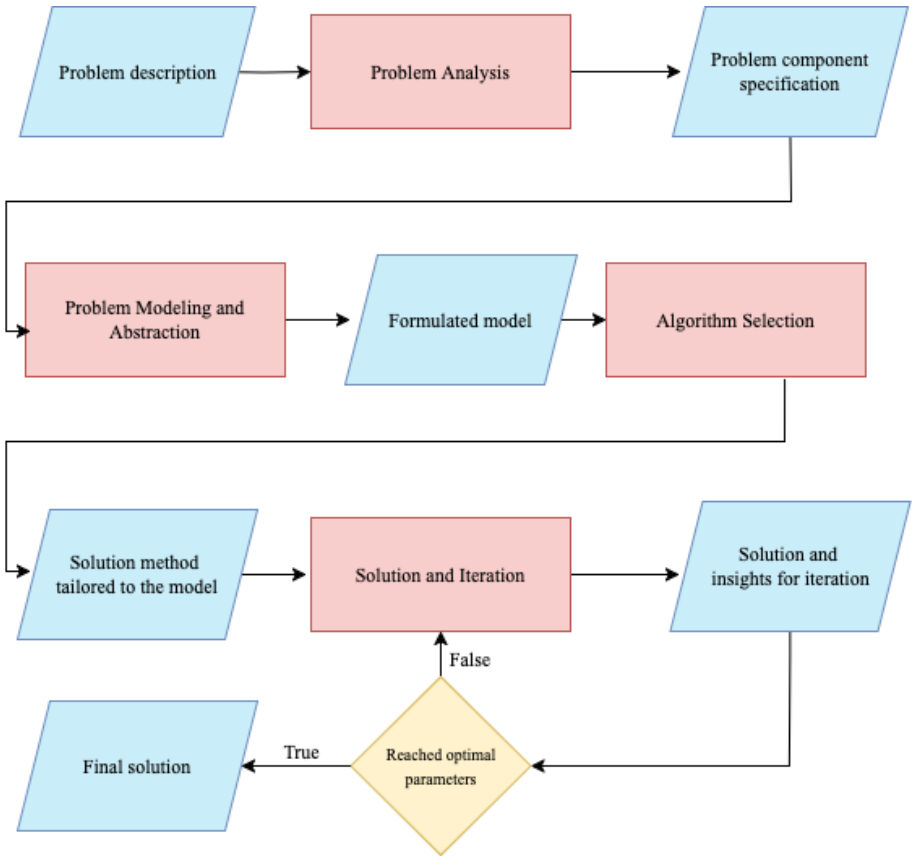

Figure 3.1|: Combinatorial Optimization Workflow Diagram

Blue parallelogram: Input/output of a process
Red rectangle: Process
Yellow rhombus: Decision

  - What constitutes a feasible solution for this problem?
  - What is the scope of the solution space?
  - What is the cost function?
  - What are the decision variables?
  - What constraints, if any, must be satisfied?

- **Formulated Model:** A mathematical representation of the problem, capturing its variables, objective function, and constraints.
- **Solution method tailored to the model:** A method (i.e., an algorithm) with the problem formulation mapped onto it.
- **Solution and insights for iteration:** The current best solution obtained by the algorithm. In the case of approximate methods, the solution provides insights for refining parameters or trying alternative approaches in subsequent iterations.

- **Final solution**: The ultimate solution to the CO problem, derived after the iterative process concludes or when a satisfactory solution is achieved.

Although these stages are described separately to aid the understanding of the overall process, they can be highly interconnected and may influence one another. For example, certain algorithms, such as Quantum Annealing (QA), require the problem to be modeled as a QUBO or Ising formulation, directly affecting the modeling step.

Ultimately, mastering the CO process requires a deep understanding of these interdependencies and familiarity with the tools available at each step. This knowledge enables practitioners to solve optimization problems efficiently and make well-informed decisions throughout the pipeline.

Moreover, this workflow serves as a foundation for evaluating and enhancing software tools designed to solve CO problems. In the context of QPLEX, understanding these distinct stages enabled us to systematically assess the library's capabilities and identify areas for improvement. Each stage of the workflow represents potential touchpoints where QPLEX can provide value to users, from problem modeling to solution iteration. As we will see in Chapter 4, this structured view of the CO pipeline directly informs our analysis of QPLEX's completeness, allowing us to map specific functional requirements to the workflow stages where they add the most value. This mapping ensures that enhancements to QPLEX align with the practical needs of users throughout the CO problem-solving process.

Finally, while this workflow provides a structured approach to solving CO problems, it is important to acknowledge its boundaries and scope. As a high-level overview, it deliberately maintains generality to serve as an entry point for newcomers to the field. This generality means the workflow does not prescribe specific methodologies for particular problem contexts, instead focusing on the fundamental stages common across CO problem-solving. Furthermore, while the chapter introduces several methodologies within each stage, it does not exhaustively cover all available approaches. Rather, it highlights representative examples to illustrate the key considerations and decisions at each step of the process. These characteristics make the workflow valuable as an educational and organizational tool, while leaving room for practitioners to adapt and expand upon it based on their specific needs and contexts.

Chapter 4

Evaluating the Completeness of the QPLEX Library

Building on the workflow established in Chapter 3, this chapter addresses part of our second research question by examining how QPLEX can be enhanced to better support practitioners throughout the CO problem-solving process. By analyzing the completeness of QPLEX against both the workflow stages and the native capabilities of underlying quantum SDKs, we identify specific opportunities for improvement that will strengthen QPLEX's role as a comprehensive tool for quantum-enabled CO. This systematic evaluation ensures that our proposed enhancements align with both the theoretical framework of CO problem-solving and the practical needs of users working with quantum resources.

4.1 Introduction

Quantum open-source software is a growing ecosystem encompassing tools for quantum computing simulation, compilation, and full-stack software development. Notable examples include Python-based libraries such as Qiskit by IBM, PennyLane[1] by Xanadu and OpenQAOA[2] by Entropica Labs. These pieces of software enable practitioners to program quantum computers to solve real-world problems. The first two are general-purpose libraries that allow users to create custom quantum circuits and execute them on QPUs. OpenQAOA, on the other hand, is a quantum optimization library focusing on using the QAOA algorithm for solving CO problems on various quantum backends. As with any open-source initiative, these projects rely heavily on community contributions from both users and developers. Following software engineering best practicessuch as well-maintained documentation, version control, test suites, and issue trackingensures their continued growth and improvement [11].

Within this ecosystem, QPLEX emerges as a high-level abstraction package designed to simplify the modeling and execution of CO problems on quantum hardware. Building on the workflow established in Chapter 3, QPLEX aims to support practitioners through each stage of the CO process from problem modeling to solution and iteration while abstracting the complexity of utilizing quantum resources. However, developing an abstraction layer

[1] https://pennylane.ai/
[2] https://openqaoa.entropicalabs.com/

such as QPLEX inherently involves trade-offs. While it reduces complexity and improves accessibility, it may also hide or limit access to features available natively in its underlying SDKs. Finding the right balance between ease of use and feature completeness is a key challenge when developing such libraries.

Furthermore, evaluating the completeness of QPLEX requires a systematic approach that considers both its role in the CO workflow as well as its effectiveness as an abstraction layer. In this context, completeness refers to how well the library supports essential CO problem-solving features natively and through its underlying SDKs. By comparing QPLEX's current features against the native capabilities of its supported SDKs, we can identify gaps where additional functionality could enhance the library's utility.

The identification of these gaps is crucial for QPLEX's continued relevance and adoption. If critical features are missing, users may be forced to bypass QPLEX often and access the SDKs directly. This undermines the value of QPLEXs abstraction layer and creates barriers to adoption, particularly among researchers and practitioners seeking an integrated, easy-to-use solution. Additionally, as new algorithms emerge, the librarys repertoire of built-in algorithms may become outdated, necessitating the implementation of new, state-of-the-art approaches and potential support for custom algorithms. Thus, identifying and integrating important missing features will ensure QPLEX meets user expectations and encourage wider adoption.

This chapter presents a systematic evaluation of QPLEX's completeness, building on the CO workflow established in Chapter 3. We begin by examining QPLEX's current feature set to establish a baseline understanding of its capabilities. We then analyze each supported quantum SDK to identify valuable features that could enhance QPLEX's functionality at various stages of the CO process. This analysis leads to a set of functional requirements for extending QPLEX.

Currently, QPLEX integrates with three major quantum providers: IBM Quantum, D-Wave, and Amazon Braket (AWS). Each provider has a quantum SDK with evolving feature sets addressing a variety of problem domains, not limited to CO. D-Wave is an exception, focusing primarily on optimization problems. Since our library serves as a bridge between the user and these providers, it wraps only a subset of their functionalitiesthose directly relevant to CO problems. However, because the quantum SDKs are rapidly evolving and QPLEX is still in its early stages of development, some useful SDK CO features remain unavailable through QPLEX.

Using the workflow outlined in Chapter 3 as a foundation, we identified three main areas for potential improvements to the QPLEX library. Two of these areas align directly with specific stages of the proposed workflow for CO, while the third encompasses general enhancements to the library. These areas are:

- **General Improvement to QPLEX:** Broader features that improve compatibility with the latest technologies, enhance performance, and address general usability.
- **Algorithm Selection:** Improvements related to the stage introduced in Section 3.4

- **Solution and Iteration:** Enhancements pertaining to the stage discussed in Section 3.5

The objective of this chapter is to identify desirable but currently unsupported features within these three areas. The proposed features aim to enhance the user experience by reducing the need to work directly with other SDKs and establishing QPLEX as a more comprehensive and accessible tool for CO.

4.2 Current Features of QPLEX for CO Problem-Solving

This section outlines the features offered by QPLEX, focusing on two main categories: high-level features that address general CO problem-solving, and provider execution options that offer configuration settings depending on the quantum backend used. Table 4.1 summarizes the features discussed in this section and serves as a baseline for evaluating the library's completeness and comparing it against native SDKs in the following sections.

4.2.1 High-Level Features of QPLEX

- **Problem modeling using DOcplex:** QPLEX leverages the DOcplex modeling API to allow users to express CO problems in a familiar mathematical format. This integration ensures ease of formulation for users coming from a classical optimization background [18].
- **Model solving using built-in hybrid quantum-classical algorithms:** QPLEX supports the execution of two well-known VQAs: QAOA and VQE, as well as D-Wave's hybrid solvers.
- **Algorithm execution on multiple quantum providers:** QPLEX offers the flexibility to run the built-in algorithms on quantum hardware provided by IBM Quantum, Amazon Braket, and D-Wave without the need for added overhead.
- **VQE Algorithm with a fixed ansatz:** QPLEX currently limits VQE to a single built-in fixed ansatz, which simplifies the interface but may restrict users who need more customization in their problem-solving approach.

4.2.2 Provider Execution Options

The three quantum providers available in our library support options for configuring their execution workflows. However, the current number of provider-specific options available through QPLEX is limited.

- **IBM Quantum and Braket**

  - Selection of QPU (backend): Users can specify which quantum processing unit (QPU) or simulator, if supported, to execute their circuits on.
  - Execution on a local simulator: QPLEX supports the use of local simulators that ship with the Qiskit and Braket SDKs.
  - Number of shots configuration option: Users can specify the number of shots (repeated circuit executions) for quantum measurements.

- **D-Wave**
  - Automated model-based hybrid solver selection: QPLEX implements logic to automatically choose between D-Wave's Binary Quadratic Model (BQM), Discrete Quadratic Model (DQM), and Constrained Quadratic Model (CQM) formulations and their respective hybrid solver based on the problems characteristics. This feature reduces user effort and ensures the most suitable solver is applied, but eliminates flexibility when choosing the hybrid solver to use.

4.2.3 Summary of the Current Scope of QPLEX

The QPLEX library currently provides essential features for solving CO problems across three quantum providers. It offers a CO modeling API that allows users to seamlessly transition from problem formulation to solution [18]. Additionally, QPLEX integrates solvers that handle input parsing, algorithm execution, and results retrieval for each supported quantum provider. The library includes two built-in gate-based VQAsQAOA and VQEthat can be executed on the platforms made available by the two gate-based quantum providers, namely IBM Quantum and Amazon Braket. Model execution on D-Wave's hybrid solvers is also supported, along with local classical execution using the CPLEX solver provided by DOcplex, the optimization library on which QPLEX is built.

QPLEX also offers a set of provider-specific execution options to customize algorithm execution. These options allow users to modify key execution parameters, such as the number of shots for circuit runs and the backend device on which the algorithm will execute. However, the range of execution options currently available through QPLEX is limited compared to the full set of features accessible directly via the native SDKs of these quantum providers.

It is also worth noting that QPLEX supports several configuration options for the classical optimization component of VQAs. These options include:

- **Optimizer:** Users can select a classical optimizer from the set of optimizers available in the SciPy library [55].
- **Maximum iterations:** Users can set a limit on the number of iterations for the optimization process.
- **Tolerance:** Users can define the precision required for the optimizations convergence.

- **Seed:** Users can specify a seed for reproducibility of results.

Additionally, all supported configuration options, including provider-specific ones, come with default values, as detailed in the original thesis [15].

In the following section, we analyze the native SDKs of IBM Quantum (Qiskit), Amazon Braket, and D-Wave to identify additional execution options that could enhance the performance, solution quality, and flexibility of QPLEX.

Table 4.1: Current Features of the QPLEX Library for CO Problem-Solving

Category	Feature	Description	Relevant Providers	Notes
High-Level Features	Problem modeling with DOcplex	Users express CO problems using DOcplex's mathematical modeling framework.	All	Standard feature across all providers.
	Multi-provider execution	Allows execution of algorithms on IBM Quantum (IBMQ), Braket, and D-Wave platforms.	All	Flexibility in provider choice.
	Built-in quantum algorithms (QAOA, VQE)	Supports execution of QAOA and VQE for CO problem-solving.	IBMQ, Braket	VQE limited to a built-in fixed ansatz.
	Hybrid solvers on D-Wave	Users can run optimization models on D-Wave's hybrid solvers.	D-Wave	
	Classical CPLEX solver support	Allows solving the problem model using CPLEXs classical solver.	All	Provides a baseline or benchmark for quantum solutions.
Provider Execution Options	Selection of backend (QPU)	Users can specify the backend (QPU) or simulator for execution.	IBMQ, Braket	
	Local simulator support	Provides local simulation before running on real hardware.	IBMQ, Braket	
	Number of shots option	Users can configure the number of shots for circuit execution.	IBMQ, Braket	
	Automatic solver selection (BQM, DQM, CQM)	Automatically selects the optimal solver based on the model's structure.	D-Wave	Reduces the need for manual solver selection at the cost of flexibility.

4.3 Identification of Missing Features and Improvement Opportunities

In this section, we analyze the features available directly through the underlying SDKs supported by QPLEX, focusing on capabilities that could enhance CO problem-solving. For each SDK, we examine execution options and integration features that could provide significant value if implemented in QPLEX. We also propose improvements to the Algorithms module of QPLEX to expand its optimization capabilities. This analysis is structured according to our three identified improvement areas: general improvements to the library's infrastructure, and enhancements to the algorithm selection and solution iteration stages established in Chapter 3's workflow. This organization ensures our analysis directly maps potential improvements to the stages where they would most benefit practitioners solving CO problems.

4.3.1 Studying the Underlying Quantum SDKs

4.3.1.1 Qiskit

Qiskit [56] is an open-source Python library and framework designed for modeling and executing gate-based quantum algorithms across various hardware platforms, including quantum simulators and QPUs. It provides tools to support different stages of the quantum computing software toolchain, from defining quantum circuits to compiling and executing them. Developed and maintained by IBM with contributions from the open-source community, Qiskit recently reached a major milestone with the release of version 1.0.0[3]. This version emphasizes stability and ensures backward compatibility throughout the 1.x release lifecycle.

In this section, we examine the configuration options provided by the Qiskit SDK. These options enable customization across various stages, including circuit compilation and execution, as well as high-level functionalities such as the Qiskit Runtime Session, which optimizes processing time by reducing queue delays. With the workflow outlined in Chapter 3 and the feature matrix presented in this chapter as reference points, our review of the Qiskit API[4] specification reveals the following missing functionalities in QPLEX. These capabilities could significantly enhance QPLEXs adaptability, performance, and overall user experience when tackling diverse CO problem requirements:

- **Qiskit Runtime Sessions**: Currently, when submitting a circuit for execution on an IBMQ backend as part of a VQA, each job enters the device's queue and must wait its turn during every optimization cycle of the algorithm. Qiskit Runtime Sessions address this problem by prioritizing subsequent jobs in the queue, reducing the overall time needed to complete all optimization iterations, as described in the future work section of the original thesis [15].

[3] https://docs.quantum.ibm.com/api/qiskit/release-notes/1.0#100
[4] https://docs.quantum.ibm.com/api/qiskit

```
39    def select_backend(self, qubits: int) -> Backend:
40        if self.backend != "simulator":
41            provider = IBMQ.get_provider(hub='ibm-q')
42            return provider.get_backend(self.backend)
43        return Aer.get_backend("qasm_simulator")
44
```

Figure 4.1|: Code snippet for the *select_backend* method of the IBMQ solver. The argument *qubits* is not used.

- **Automated Circuit Optimization**: Qiskit provides four levels of optimization through the *qiskit.transpiler* module, with the first level applying no optimization. IBM recommends choosing a higher optimization level (above zero) despite the additional overhead, as it can improve execution efficiency.

- **Sampler Primitive**: The Sampler primitive accepts quantum circuits for execution and samples from the classical registers in those circuits[5]. This approach parallels the current workflow in QPLEX, where circuits are sent to the backend device and sampled to obtain counts. The Sampler and Estimator primitives are now IBMQ's recommended methods for leveraging quantum resources, as they allow users to focus more on obtaining results than on managing hardware intricacies [57].

- **Estimator Primitive**: The Estimator primitive calculates the expectation value of one or more quantum circuits and observables. Unlike QPLEXs current workflow, the Estimator directly computes the expectation value from the sampled counts, which would reduce the parsing workload required in the IBMQ solver. However, integrating this workflow change into QPLEX is not straightforward, as it would require significant architectural adjustments across all gate-based solvers[6].

- **Selection of least-busy backend**: Currently, the *IBMQSolver* class includes a method for selecting a specific backend device based on user input, including a local simulator bundled with Qiskit. However, this method could be improved to select the least busy backend, i.e. the QPU with the lowest number of jobs in the queue, by utilizing the *QiskitRuntimeService.least_busy* method. Additionally, while the *select_backend* method accepts a parameter to indicate the minimum number of qubits required to run the circuit, this parameter is not currently utilized, as shown in Figure 4.1. Addressing this would ensure proper functionality.

It is important to note that, at the time of writing, QPLEX does not support Qiskit version 1.0.0 or later. With this release, Qiskit introduced syntax changes that are not compatible with the current implementation of the IBMQ solver. However, due to the decoupled design

[5] https://docs.quantum.ibm.com/api/qiskit/primitives

[6] Since the Generalized Gate-based Algorithm Execution Manager expects all solvers to return counts from the *solver.solve* function call, enabling the use of the Estimator primitive for the IBMQ solver does not integrate seamlessly with the curent architecture. Furthermore, modifying the GGAEM to accommodate this new workflow would defeat the reusability of the manager.

of our library's solver architecture, adding support for Qiskit 1.0.0 requires updates only to the *IBMQSolver* class. After reviewing the new syntax rules, we identified the following necessary updates for the IBMQ solver:

- **Changes to the Qiskit IBMQ Provider Package**: the package *qiskit.providers.ibmq*[7] has been deprecated and its functionalities have been moved to the *qiskit_ibm_runtime*[8] package. These functionalities are essential for managing communication with IBMQ servers.
- **Aer Simulator**: The Aer Simulator is now imported from the *qiskit_aer*[9] package.

4.3.1.2 Amazon Braket

The Amazon Braket Python SDK is an open-source software package that provides a set of tools for interacting with the Amazon Braket service [58]. This service grants access to QPUs from companies such as IonQ, IQM, QuEra and Rigetti[10], as well as cloud simulators provided by AWS and classical computational resources for executing hybrid algorithms, such as VQAs.

Amazon describes Braket as a managed service that facilitates quantum circuit execution on various devices. The SDK includes tools for building and executing quantum circuits and offers lower-level functionalities tailored to specific quantum hardware, such as qubit allocation on QPU devices[11] and error mitigation on IonQs Aria QPU[12], among others[13].

Thus, our review of the Amazon Braket Python SDK identified the following extension points in QPLEX:

- **Amazon Hybrid Jobs**[14]: Similar to Qiskit Runtime Sessions 4.3.1.1, Hybrid Jobs provide a mechanism for optimizing the execution of multiple quantum circuits by prioritizing them over regular quantum tasks in a QPUs queue. This is particularly advantageous for executing VQAs, such as those supported by QPLEX, which require iterative executions based on previous results. Additionally, Amazon Braket compiles parameterized circuits only once, rather than for each iteration of the variational algorithm, which further improves performance. This feature was also highlighted as a potential future enhancement in the original thesis [15].

[7] https://github.com/Qiskit/qiskit-ibmq-provider
[8] https://github.com/Qiskit/qiskit-ibm-runtime
[9] https://github.com/Qiskit/qiskit-aer
[10] https://aws.amazon.com/braket/quantum-computers/
[11] https://github.com/amazon-braket/amazon-braket-examples/blob/main/examples/braket_features/Allocating_Qubits_on_QPU_Devices.ipynb
[12] https://github.com/amazon-braket/amazon-braket-examples/blob/main/examples/braket_features/Error_Mitigation_on_Amazon_Braket.ipynb
[13] https://github.com/amazon-braket/amazon-braket-examples/tree/main/examples/braket_features
[14] https://github.com/amazon-braket/amazon-braket-examples/blob/main/examples/hybrid_jobs/0_Creating_your_first_Hybrid_Job/0_Creating_your_first_Hybrid_Job.ipynb

- **Provider-specific Execution Options**: Since Amazon Braket itself is a wrapper for the aforementioned providers, there are execution options that are accessible through the Braket SDK, but not through QPLEX as of now. Support for these options would enhance the flexibility of our library.

4.3.1.3 Ocean

The Ocean SDK [59] is a suite of open-source software tools developed by D-Wave, available on their GitHub repository.[15] Similar to other SDKs designed for quantum problem-solving, Ocean offers classes and functions for modeling problems and solving them using both classical and quantum resources. D-Wave provides various *solvers*computational resources that accept problem formulations and return optimized solutionsthrough its *Leap* platform. These solvers include hybrid solvers, which leverage both classical heuristics and QPUs; quantum solvers, which utilize D-Waves *Advantage* QPUs; and purely classical solvers.

The QPLEX library leverages Ocean's modeling tools to translate the initial DOcplex model into a compatible Ocean model. The Ocean SDK offers three distinct models: the Binary Quadratic Model (BQM), which is unconstrained and uses binary variables; the Discrete Quadratic Model (DQM), which is also unconstrained but includes discrete variables; and the Constrained Quadratic Model (CQM), which supports constraints and allows real, integer, and binary variables. Each model serves different use cases, and QPLEX automatically selects the appropriate model based on the structure of the DOcplex input, as mentioned in [15]. Notably, while the BQM can be submitted to the D-Wave quantum solvers (e.g., Advantage) QPLEX currently supports submitting to the hybrid solver only.

Using our workflow and the feature matrix presented in this chapter as reference points, our review of the Ocean SDK identified the following extension points in QPLEX:

- **Quantum solvers support**: D-Waves quantum solvers [60], often referred to as samplers, are accessible through the Leap platform and leverage D-Waves quantum systems. Expanding QPLEX to support additional D-Wave solvers would enhance flexibility, providing more options for benchmarking resultsan important practice when solving CO problems using approximation techniques.

- **Custom minor embedding**: When using a quantum solver on the Leap platform, the problem model must be mapped to the graph structure of the selected samplera process known as minor embedding [61]. Minor embedding is computationally challenging for classical computers and has a direct impact on the solution quality produced by the QPU [62]. D-Wave provides a default embedding when a custom one is not specified; however, generic embeddings may not be optimal for all problem instances and can sometimes compromise result quality. This limitation has motivated researchers to develop custom embeddings tailored to specific problem types [63] [64].

[15] https://github.com/dwavesystems

Adding support for custom minor embedding in QPLEX would significantly enhance its experimental capabilities.

- **Local classical solver support**: Adding support for local classical solvers would be valuable for benchmarking purposes. Classical solvers can serve as a baseline to compare the performance and solution quality of quantum or hybrid quantum-classical solvers. By integrating Ocean's local classical solver support into QPLEX, users can conduct side-by-side evaluations with yet another solver, which is essential for assessing the advantages and limitations of quantum approaches in solving CO problems.

4.3.2 Improvements to the Algorithms Module

The QPLEX Algorithms module contains algorithmic specifications (i.e., OpenQASM3 string representations) of the two gate-based quantum algorithms supported by the library. However, support for novel and promising methods in the CO field such as the Quantum Alternating Operator Ansatz (QAOAnsatz) [65], as well as allowing for a custom VQE ansatz would enhance the QPLEX library, allowing practitioners to experiment further.

4.4 Elicitation of Functional Requirements

After examining the SDKs supported by QPLEX and analyzing the algorithms module of our library, we defined a set of functional requirements following established software engineering methodologies [66]. These requirements serve as key artifacts from the analysis phase of this project, providing foundational input for the design and implementation phases discussed in Chapter 6. Table 4.2 includes a mapping of these functional requirements to specific stages of the workflow outlined in Chapter 3, enhancing the clarity and verifiability of the requirements within the context of QPLEX and CO. To ensure a systematic approach to requirement development, we followed a structured process that considered multiple factors in deriving and prioritizing these requirements.

4.4.1 Requirements Derivation Process

The elicitation of functional requirements emerged from a four-step analysis process:

First, we identified feature gaps by comparing QPLEX's current capabilities against those available in native SDKs. This comparison revealed several areas where QPLEX could be enhanced to better support CO problem-solving. For example, while examining Qiskit's features, we found that QPLEX lacks support for Runtime Sessions, which could significantly improve execution efficiency for variational algorithms. Second, we assessed the user impact of each potential feature. This assessment considered how the feature would enhance the practitioner's ability to solve CO problems effectively. For instance, support for

custom VQE ansatz would enable researchers to experiment with different quantum circuit structures, potentially leading to better solutions for specific problem instances.

Third, we evaluated the technical feasibility of implementing each feature. This evaluation included considerations such as compatibility with QPLEX's current architecture, potential dependencies on other features, and the complexity of integration with existing code. Features requiring substantial architectural changes were noted for their higher implementation complexity.

Finally, we formulated specific functional requirements that captured the essential capabilities needed to address the identified gaps while considering both user needs and technical constraints.

4.4.2 Functional Requirements

The following are the functional requirements (FR):

- **FR-1:** The system shall support executing combinatorial optimization formulations on IBMQ QPUs via Qiskit Runtime Sessions.
- **FR-2:** The system shall support compatibility with Qiskit version 1.0.0.
- **FR-3:** The system shall support quantum circuit execution using Qiskits Sampler primitive on IBM QPUs.
- **FR-4:** The system shall support quantum circuit execution using Qiskits Estimator primitive on IBM QPUs.
- **FR-5:** The system shall support automatic circuit optimization through the Qiskit transpiler with selectable optimization levels.
- **FR-6:** The system shall support executing combinatorial optimization formulations on QPUs provided by the Amazon Braket service using Hybrid Jobs.
- **FR-7:** The system shall support provider-specific execution options accepted by the quantum providers available through the Amazon Braket service.
- **FR-8:** The system shall support executing combinatorial optimization formulations on D-Wave's quantum solvers.
- **FR-9:** The system shall support custom minor embeddings for executing combinatorial optimization formulations on D-Wave quantum solvers.
- **FR-10:** The system shall support user-defined ansatz for the VQE algorithm.
- **FR-11:** The system shall support executing combinatorial optimization formulations using the QAOAnsatz algorithm on gate-based providers.

Table 4.2: Mapping of Functional Requirements to Workflow Stages with Rationale

Requirement	**Stage**	**Rationale**	**Workflow Impact**
FR-1: Runtime Sessions	SI	Reduces quantum execution queue times	Enables faster experimentation cycles
FR-2: Qiskit 1.0.0	GI	Ensures compatibility with latest quantum SDK features	Provides access to improved tools across all workflow stages
FR-3, 4: Sampler, Estimator	GI	Implements modern approach for circuit execution and measurement	Streamlines execution process across IBM backends
FR-5: Circuit Optimization	SI	Improves circuit efficiency	Enhances solution quality
FR-7: Provider Options	SI	Exposes provider-specific configuration options to users	Allows fine-tuning of quantum executions for better results
FR-8: D-Wave Solvers	AS, SI	Provides access to quantum annealing hardware	Expands algorithm options and supports solution validation
FR-9: Custom Embedding	AS, SI	Allows optimization of problem mapping to quantum hardware	Improves solution quality through better hardware utilization
FR-10: Custom VQE	AS, SI	Enables experimentation with different circuit structures	Supports problem-specific algorithm customization
FR-11: QAOAnsatz	AS, SI	Implements state-of-the-art variant of QAOA	Expands algorithm options and supports solution validation

Legend:
SI - Solution and Iteration
AS - Algorithm Selection
GI - General Improvement to QPLEX

The table maps functional requirements to workflow stages while providing rationale for the mapping and explaining the impact on the overall workflow.

4.4.3 Requirement Prioritization

The prioritization of functional requirements followed a systematic evaluation framework based on four key criteria:

1. **User Value (40%):** The direct impact on practitioners' ability to solve CO problems effectively. For example, FR-1 (Runtime Sessions) received a high score in this category because it significantly reduces execution time for variational algorithms.
2. **Implementation Complexity (30%):** The technical effort and dependencies required for implementation. Requirements like FR-2 (Qiskit 1.0.0 compatibility) scored well here as they primarily involve updating existing code rather than developing new architectural components.
3. **Workflow Alignment (20%):** How well the requirement supports multiple stages of the CO workflow. Requirements that enhance both algorithm selection and solution iteration stages, such as FR-8 (D-Wave solvers), received higher scores.
4. **Resource Availability (10%):** The feasibility of implementation given current project constraints and expertise. This criterion helped ensure the selected requirements could be implemented within the project timeline.

Each requirement was scored against these criteria using a weighted scoring system. For example, FR-1 received a high overall priority because:

- It offers significant user value by reducing execution times (40% Œ 9/10)
- It has moderate implementation complexity as it builds on existing Qiskit functionality (30% Œ 7/10)
- It primarily impacts the solution iteration stage but indirectly benefits algorithm selection (20% Œ 7.5/10)
- Implementation expertise is readily available (10% Œ 8/10)
- Final weighted score: 8/10

Based on this evaluation, we selected requirements FR-1, FR-2, FR-3, FR-5, FR-7, FR-8, FR-9, and FR-11 for implementation in this iteration of the project. This selection provides a balanced set of improvements across different workflow stages while remaining feasible within project constraints. The remaining requirements will be addressed in future iterations of the project.

Table 4.3: Functional Requirements Prioritization Analysis

Requirement	UV	IC	WA	RA	Score	Decision
FR-7: Provider Options	7.5	10.0	10.0	8.0	8.8	Selected
FR-5: Circuit Optimization	8.0	10.0	7.5	9.0	8.6	Selected
FR-11: QAOAnsatz	10.0	6.0	10.0	7.0	8.5	Selected
FR-8: D-Wave Solvers	8.0	8.0	10.0	8.0	8.4	Selected
FR-9: Custom Embedding	8.0	8.0	10.0	7.0	8.3	Selected
FR-1: Runtime Sessions	9.0	7.0	7.5	8.0	8.0	Selected
FR-2: Qiskit 1.0.0	8.0	9.0	5.0	9.0	7.8	Selected
FR-3: Sampler Primitive	7.0	7.0	7.5	9.0	7.3	Selected
FR-10: Custom VQE	7.0	7.0	7.5	5.0	6.9	Future
FR-6: Hybrid Jobs	9.0	4.0	7.5	4.0	6.7	Future
FR-4: Estimator Primitive	7.0	3.0	7.5	6.0	5.8	Future

UV - User Value (40%) WA - Workflow Alignment (20%)
IC - Implementation Complexity (30%) RA - Resource Availability (10%)

Requirements were scored on a scale of 1-10 for each criterion. The final score is calculated using the weighted sum of all criteria. Requirements with scores above 7.0 were selected for implementation in this iteration.

4.5 Chapter Summary

This chapter detailed our approach to validating the completeness of the QPLEX library. It included an overview of the librarys current scope and the identification of features to enhance QPLEXs combinatorial optimization capabilities, some of which are currently accessible only through the underlying SDKs. These features were elicited as functional requirements using traditional software engineering practices. Additionally, a subset of these requirements was selected for implementation in this project, with prioritization decisions explained, while the remaining requirements were left for future work.

This chapter concludes the first half of the analysis stage of this project, resulting in the functional requirements list as a key artifact. This artifact will serve as a foundation for the design and implementation stages described in Chapter 6. The following chapter, Chapter 5, continues with the second part of our analysis and validation process, focusing on the extensibility of the QPLEX library.

Chapter 5

Examining the Extensibility of the QPLEX Library

In software engineering, extensibility is a quality attribute that describes a software systems ability to incorporate additional features or enhancements with minimal disruption to existing functionality [20]. A highly extensible system is architected to facilitate efficient implementation of new features, enabling developers to extend the software without significant modifications to its core structure.

However, several factors other than software architecture can affect extensibility by complicating the process of implementing new features. These include code documentation, test coverage, and CI/CD pipelines. In the case of documentationparticularly in open-source projectsinsufficient or poorly maintained documentation can make it challenging to understand the codebase, especially in modularized systems with interconnected logic. Similarly, high test coverage is critical to ensuring that new features do not compromise established functionalities. Finally, CI/CD pipelines optimize and accelerate the development process by promoting frequent, reliable releases.

This chapter evaluates the extensibility of the QPLEX library through a framework that examines these four key areas:

- Documentation: The quality and availability of contributor-facing resources that support code understanding and extension.
- Test Coverage: The breadth and depth of tests ensuring software correctness when implementing new functionalities.
- CI/CD Pipelines: The robustness of automation in testing, integration, and deployment processes.
- Software Architecture: The modularity and scalability of the codebase, enabling ease of modification and integration.

In the following sections, we analyze each of these aspects to evaluate the extensibility of the QPLEX library. Building on this analysis, we elicit a set of non-functional requirements aimed at enhancing QPLEXs extensibility.

5.1 Documentation

Contributor-facing documentation is essential for ensuring software quality and extensibility [67]. Effective documentation minimizes errors when developing and maintaining code, whereas poor or incomplete documentation contributes to technical debt. As a project grows in size and complexity, the importance of well-maintained documentation becomes increasingly evident.

Although the upfront cost of creating high-quality documentation may seem prohibitive, its benefitssuch as reduced debugging time and increased developer productivityfar outweigh the initial investment, as highlighted by Zhi et al. [68]. In their paper, the authors identified five key attributes of highly usable and effective documentation: completeness, consistency, accessibility, readability, and up-to-date-ness. These attributes are crucial for supporting extensibility in a software tool like QPLEX.

To evaluate and improve QPLEXs documentation, this thesis considers these attributes alongside best practices for documenting Python code, as outlined in PEP 257 and PEP 8 [69], [70].

5.1.1 Documentation of QPLEX Prior to This Work

During the initial development of QPLEX, functional requirements were prioritized over documentation due to resource constraints. As a result, the documentation coverage prior to this work was limited. For instance, only a few comments were included in select files, as illustrated in Figures 5.1, 5.2 and 5.3. These figures highlight the state of documentation in QPLEX before our contributions, showing that most modules, classes, functions, and methods lacked detailed descriptions, and only abstract classes were fully documented.

```
1  """
2  This module provides the different gate-based algorithms
3  """
4
5  from qplex.algorithms.qaoa import QAOA
6  from qplex.algorithms.vqe import VQE
7
```

Figure 5.1|: A code snippet showing the documentation in the Algorithms module init file prior to this work

This lack of coverage hindered the librarys extensibility by creating barriers for new contributors attempting to modify or extend the codebase. Addressing this gap was important for supporting the integration of new quantum providers and algorithms and, in general, facilitating future open-source contributions.

In the following subsection, we outline specific improvements to enhance QPLEX's documentation. These include introducing comprehensive module-level and function-level docstrings and integrating automated tools for generating API documentation.

```python
from abc import ABC, abstractmethod
from typing import List

from qplex.solvers.base_solver import Solver
import numpy as np


Juan Giraldo
class Algorithm(ABC):

    """Abstract class for a quantum algorithm"""

    Juan Giraldo
    def __int__(self, model):
        ...

    Juan Giraldo
    @abstractmethod
    def create_circuit(self, model) -> str:
        """Creates a quantum circuit in the form of a OpenQASM2 string from an optimization model.

        Args:
            model: The Qmodel to be converted.

        Returns:
            An OpenQASM2 string.
        """
        ...

```

Figure 5.2|: A code snippet showing part of the documentation in the abstract Algorithm class prior to this work

5.1.2 Improving the Documentation of QPLEX

To enhance the completeness and usability of the QPLEX documentation, all modules, classes, methods, and functions should include consistent, detailed, and readable docstrings. These docstrings should clearly describe the functionality, inputs and outputs of each code element, adhering to standard documentation practices such as those outlined in PEP 257 [69].

Moreover, an API specification is useful for improving navigation and accessibility. This specification should provide users with a searchable interface for exploring available classes and methods. The development of this platform can be largely automated using existing software tools for documentation generation (e.g., Sphinx[1] or MkDocs[2]).

Furthermore, the integration of new documentation into the API specification page can be incorporated into the librarys CI/CD pipeline, discussed in 5.3. This ensures that any updates or additions to the codebase are automatically reflected in the documentation.

[1] https://www.sphinx-doc.org/en/master/
[2] https://www.mkdocs.org/

```python
from abc import ABC, abstractmethod
from typing import Dict, Any


Juan Giraldo
class Solver(ABC):

    """Abstract class for a quantum solver"""

    Juan Giraldo
    @abstractmethod
    def solve(self, formulation) -> Dict:
        """Determines how the solver will execute the problem formulation.

        Args:
            formulation: The formulation to be solved. Can be a QModel or a 2 string.

        Returns:
            A dictionary with the solution for the execution.
        """
        ...

```

Figure 5.3|: A code snippet showing part of the documentation in the abstract Solver class prior to this work

However, this automation relies on the presence of complete and properly formatted docstrings within the codebase.

Lastly, to support this workflow, a standardized contribution guideline should be established, requiring that all new classes, methods, and functions include comprehensive docstrings. This requirement will ensure consistency across the library and maintain high documentation quality as QPLEX evolves.

In summary, the proposed improvements include:

- The addition of consistent, detailed, and readable docstrings to all current and new modules, classes, methods, and functions.
- The creation of an API specification page that integrates CI/CD to automatically update when new documentation is added.
- The establishment of a standardized contribution guideline to ensure consistency and maintainability.

5.2 Test Coverage

Software testing is a critical stage of the software development life cycle (SDLC) aimed at verifying that implemented features function as intended and validating that the software meets the specified requirements. It plays a key role in ensuring software quality by detecting and correcting errors introduced during development or maintenance [71].

The testing process is carried out in a controlled environment using test cases designed with specific preconditions (inputs) and expected postconditions (outputs). These test cases systematically assess different aspects of the software to identify potential issues. To achieve comprehensive testing, various types of testing methodologies are employed, such as unit testing, integration testing, and end-to-end testing, and each type focuses on a particular scope or level of functionality.

Unit tests, in particular, are cost-effective to automate using testing frameworks. They validate the internal functionality of individual functions or methods, assuming correct inputs are provided [72]. Established software testing practices recommend that the majority of tests in a system should be unit tests. These tests form the foundation of the overall test architecture, ensuring software correctness at the unit level (i.e., individual functions and methods) and providing a robust basis for higher-level tests [73], [74].

Moreover, test coverage is a vital metric for evaluating the extent to which the codebase of a software product is verified and validated by the existing test cases. However, achieving and maintaining sufficient test coverage requires careful planning and ongoing effort, especially in a library as dynamic as QPLEX.

Test coverage is commonly expressed as the percentage of total items tested, as defined in Equation 5.1. The term item is subjective and can vary between projects, depending on their context and goals. Commonly used definitions include lines of code, functions, branches, or requirements [75]. In this work, we define branch coverage as the test coverage item or metric to be used in QPLEX. This metric evaluates the total number of logical branches (e.g., if statements) in the codebase and compares it to the number of branches explored by the test cases. Given the size and complexity of QPLEX, this approach provides an effective estimation of test coverage. The execution of a CO formulation using QPLEX involves multiple possible branches, which are determined by the execution configuration. Branch coverage ensures that these variations are thoroughly tested, enhancing the robustness of the library.

$$\text{Test Coverage (\%)} = \left(\frac{\text{Items Tested}}{\text{Total Items}} \right) \times 100 \tag{5.1}$$

This section evaluates the state of test coverage in QPLEX, as left by previous contributors, and proposes strategies for improvement, including the selection and implementation of a testing framework, the adoption of automated tools for coverage analysis, and the enforcement of test coverage thresholds as part of the new development workflow.

5.2.1 Test Coverage of QPLEX Prior to This Work

Prior to this work, the QPLEX library lacked an established testing framework, and no tests had been designed or implemented in the available codebase. A testing framework ensures consistency and facilitates the implementation of tests. This absence of testing hindered the librarys extensibility by increasing the workload for contributors developing new features, as they were required to perform manual testing to verify that existing functionality remained intact after their changes. Addressing this technical debt by implementing automated tests for the existing codebase is critical to enhancing the librarys robustness and extensibility. A comprehensive test suite will act as a safety net, ensuring that new contributions follow a consistent structure when designing and implementing tests.

5.2.2 Enhancing the Test Coverage of QPLEX

Several testing frameworks are available for Python software projects, including Unittest[3], Pytest[4], and Nose[5]. Each has its own advantages and trade-offs. For example, Unittest is included with standard Python installations, eliminating the need for external dependencies and reducing the librarys footprint. Pytest, however, requires installation as a dependency but offers extensive features with minimal code overhead, such as parameterized tests, and plugin support. These features are particularly beneficial for testing the multiple logical branches commonly encountered in QPLEX. Moreover, Pytest is actively maintained and supported by comprehensive documentation, making it a strong candidate for QPLEX's testing framework. For these reasons, Pytest was chosen as the testing framework for QPLEX.

Apart from establishing a testing framework, the design and implementation of unit test cases for all QPLEX modules is essential for enhancing test coverage. Additionally, incorporating a test coverage report into the software development workflow would provide an objective metric for assessing the effectiveness of the implemented tests. This report can guide contributors in identifying untested portions of the codebase and ensuring comprehensive validation.

To reinforce testing standards, CI/CD automation tools, discussed in the next section, should execute all tests, generate test coverage reports, and enforce integration policies. These policies would prevent contributors from integrating new features into the existing codebase unless (1) all tests pass and (2) the established test coverage threshold is met. Such automation ensures consistency, reduces manual effort, and maintains the robustness of the library as it evolves, ultimately enhancing its extensibility.

To summarize, the proposed enhancements include:

- The establishment of a testing framework, leveraging tools such as Pytest for its rich features and active maintenance.

[3] https://docs.python.org/3/library/unittest.html
[4] https://docs.pytest.org/en/stable/
[5] https://pypi.org/project/nose/

- The design and implementation of comprehensive unit tests for all QPLEX modules.
- The incorporation of a standardized test coverage metric, along with a coverage threshold enforced through CI/CD automation tools.

5.3 CI/CD Pipelines

CI/CD is a widely used term in the software development industry that encompasses two distinct but related practices: continuous integration (CI) and continuous delivery (CD). These strategies align with DevOps principles, aiming to deliver value to clients as quickly and frequently as possible [76].

5.3.1 Continuous Integration

The primary goal of CI is to enhance developer productivity and improve software quality by promoting frequent integration of code into a shared repository. This practice enables teams to receive timely and actionable feedback, reducing bottlenecks in the development process [76].

CI achieves this by automating key tasks such as code compilation, testing, and build creation, ensuring that each code contribution is validated early in the development cycle. By bridging the gap between development and deployment, CI minimizes the risk of integration conflicts and helps maintain a deployable state of the codebase [76], [77].

5.3.2 Continuous Delivery

CD builds upon the foundation of CI by automating the process of packaging and deploying production-ready versions of a software system. The goal of CD is to ensure that software can be reliably and seamlessly delivered to users at any time, adding value frequently and minimizing the time between development and deployment [78]. By incorporating these automated steps, CD reduces the risk of errors being introduced during manual deployments and provides a consistent mechanism for delivering updates [76].

5.3.3 Automation Pipelines

GitHub Actions[6] is a platform for creating CI/CD automation workflows, seamlessly integrating with code stored in GitHub repositories [79]. These workflows can be configured to automatically build, test, and deploy code changes submitted through pull requests. Pull requests are a collaborative mechanism where developers propose changes to a repository, enabling reviews that help maintain code quality and consistency across the project [80].

[6] https://github.com/features/actions

By leveraging GitHub Actions, projects can automate critical CI tasks such as code compilation and test execution. Additionally, CD workflows can be configured to build and deploy successful pull requests, creating production-ready artifacts that are immediately usable by end users. This reduces the manual workload for contributors, allowing them to focus on feature development and bug fixes rather than repetitive validation tasks.

Since the QPLEX source code is stored in a GitHub repository, GitHub Actions is a natural and effective choice for automating its CI/CD workflows. Its seamless integration with the repository enables efficient validation of code contributions, fostering a streamlined and collaborative development process.

The integration of CI and CD pipelines through automation tools like GitHub Actions enhances the extensibility of a software project. By streamlining the validation and delivery processes, these pipelines reduce the cognitive and operational load on contributors while maintaining the robustness and reliability of the codebase. For QPLEX, this is particularly important as it evolves to support additional quantum providers and algorithms.

5.3.4 CI and CD in QPLEX Prior to This Work

Prior to this work, the QPLEX library lacked any CI/CD strategies. Without a testing framework or test cases in place, continuous integration was not prioritized by the original authors. Additionally, as the library was still in its early stages of development, no efforts were made to release it as a packaged solution for end-users.

The absence of CI/CD hindered the integration of new features by requiring manual verification and testing, increasing the workload for contributors. Similarly, end-users interested in using the library faced significant barriers, as they were required to manually download the source code and build it themselves.

To address these issues, CI/CD pipelines should be implemented to automate the integration and deployment of new software versions. Moreover, QPLEX should be deployed to the Python Package Index (PyPI),[7] the official repository for third-party Python packages. This would allow users to easily install the library using the Pip[8] package manager. Even at an early stage of development, making QPLEX accessible through PyPI would lower the adoption barrier and enable valuable user feedback to inform future improvements.

5.3.5 Proposed Improvements

The proposed improvements for CI/CD strategies and the integration of the GitHub Actions automation platform include the following:

[7] https://pypi.org/
[8] https://pypi.org/project/pip/

- **Creation of Automated Workflows:** Develop dedicated GitHub Actions workflows to automate key stages of the SDLC, including code compilation, testing, building, and deployment. These workflows will adhere to CI/CD best practices, ensuring that every pull request and code contribution is automatically verified and prepared for integration. Automation will reduce manual effort, enhance consistency, and improve overall development efficiency.

- **The deployment of QPLEX to PyPI:** Package and deploy QPLEX to PyPI. This will make the library accessible via the Pip package manager, allowing users to install it with a simple command. Deploying to PyPI will not only enhance the librarys accessibility but also encourage user adoption and feedback, which can drive further improvements in functionality and usability.

5.4 Software Architecture

The architecture of a software system plays a critical role in its extensibility. It serves as the foundation for incorporating new features by defining extension points and interfaces that enable the addition of functionalities without disrupting existing ones [81]. A well-designed architecture not only facilitates the integration of new components but also ensures the system remains maintainable and scalable as it grows.

In the case of QPLEX, the library implements a modular design with clear interfaces and abstractions for core components such as Algorithm and Solver. These interfaces act as well-defined extension points, allowing developers to introduce new quantum algorithms and solvers for additional quantum providers. By adhering to the patterns defined in the Algorithm and Solver abstract classes, contributors can seamlessly integrate new features into the library while maintaining consistency with the existing architecture.

The library's current architecture includes a Solver Factory, which implements the Factory Method design pattern [82] to create appropriate solver instances based on the selected quantum provider. This decoupling of solver creation from the rest of the system enhances extensibility by allowing new solvers to be added without modifying existing code. The factory pattern effectively isolates the implementation details of different solvers while providing a unified interface for their creation and use.

Furthermore, the modularized design of QPLEX separates the logic and data handling of the algorithms and solvers from other library components. This separation ensures that modifications to one module can be made independently, provided the interfaces between modules remain consistent. Such a design minimizes the risk of cascading changes across the codebase, thereby reducing maintenance overhead and improving stability.

This architectural approach also supports scalability. Minimal interdependence between modules enables the effortless addition or modification of components, allowing the library to grow alongside evolving user needs and technological advancements. Additionally, this

modularity facilitates parallel development, where teams or contributors can work on different modules simultaneously, improving productivity and enabling more frequent deliveries while minimizing conflicts.

While the current architecture provides a solid foundation for extensibility, it could be further enhanced by introducing an Algorithm Factory similar to the existing Solver Factory. This addition would decouple the Generalized Gate-based Algorithm Execution workflow and potential future workflows from the specific algorithm implementations. An Algorithm Factory would provide a centralized mechanism for creating algorithm instances, making it easier to add new algorithms and execution workflows without modifying existing code. This architectural improvement would further strengthen the library's extensibility by providing a more flexible and maintainable approach to algorithm management. The design and implementation of this enhancement as well as others related to the general design of the library's modules are detailed in Section 6.2.

5.5 Elicitation of Non-functional Requirements

Building on the extensibility analysis presented in this chapter and the workflow outlined in Chapter 3, we identified key areas for improvement and followed established software engineering methodologies to elicit non-functional requirements [66].

Unlike functional requirements, which define the specific capabilities of a system, non-functional requirements specify constraints or quality attributes that the system must satisfy. These requirements address aspects such as performance, scalability, maintainability, and usability, which are critical for ensuring the long-term effectiveness of a software system.

As with the functional requirements discussed in Section 4.4, these non-functional requirements are vital artifacts from the analysis stage of this work. They serve as guiding principles for the design and implementation phase, detailed in Chapter 6.

5.5.1 Non-functional Requirements

The following are the non-functional requirements (NFR):

- **NFR-1:** The system shall have consistent, detailed and readable documentation for all modules, classes, methods, and functions by including comprehensive docstrings that adhere to established standards (e.g., PEP 257) for clarity and maintainability.
- **NFR-2:** The system shall provide an API specification page that is automatically updated through CI/CD workflows whenever new documentation is added, ensuring the documentation remains current and accessible.

- **NFR-3:** The system shall include a standardized contribution guideline to ensure consistency and maintainability across the library, providing clear instructions for developers contributing to QPLEX.
- **NFR-4:** The system shall include the design and implementation of comprehensive unit tests to cover all QPLEX modules, following a testing framework and ensuring the robustness and reliability of the library.
- **NFR-5:** The system shall incorporate a standardized test coverage metric, accompanied by a coverage threshold enforced through CI/CD automation tools, to guarantee adequate validation of the codebase.
- **NFR-6:** The system shall implement GitHub Actions workflows to automate key stages of the software development life cycle, including code compilation, testing, building, and deployment.
- **NFR-7:** The system shall package and deploy QPLEX to PyPI, enabling users to install the library via the Pip package manager.

5.6 Chapter Summary

This chapter detailed our approach to analyzing the extensibility of the QPLEX library. It included an evaluation of the librarys current state across four critical aspects: documentation, test coverage, CI/CD pipelines, and software architecture. Using the workflow outlined in Chapter 3 as a foundation, we identified areas for improvement that impact the librarys extensibility. These aspects were assessed using established software engineering practices, leading to the elicitation of non-functional requirements.

This chapter concludes the second half of the analysis stage of this project, resulting in a non-functional requirements set as a key artifact. This artifact will serve as a foundation for the design and implementation stages described in Chapter 6. The following chapter transitions from analysis to design and implementation, detailing how the identified functional and non-functional requirements will be implemented to enhance QPLEXs capabilities and extensibility.

Chapter 6

Extending QPLEX: Design and Implementation

6.1 Introduction

Building on the foundational understanding of the QPLEX library introduced in Chapter 2 and guided by the functional and non-functional requirements selected in Chapter 4 and Chapter 5, this chapter advances the development process by detailing the design and implementation of these requirements. Additionally, it highlights general design changes aimed at reducing the complexity of the software librarychanges that, while not explicitly tied to any specific requirement, are nonetheless significant and deserve discussion.

The chapter is organized into three main parts:

- **System Design:** Discusses modular-level design changes (e.g., updates to classes and file structures), explaining how these modifications reduce the librarys complexity and facilitate future contributions.
- **Functional Requirements:** Explores the design and implementation of functional requirements, categorized by their impact on different modules of QPLEX.
- **Non-functional Requirements:** Addresses non-functional requirements, with sections dedicated to **Documentation**, **Testing**, and **CI/CD**.

By presenting the design and implementation in this structured manner, the chapter provides a comprehensive overview of the technical contributions of this thesis as embodied in the QPLEX library.

6.2 System Design

As outlined in Chapter 2, QPLEX was designed with a modular architecture, separating logic and concerns into distinct classes and files that interact to achieve the library's objectives. Figure 2.1 illustrates the modules that comprised QPLEX prior to this work.

During this project, the QPLEX architecture was analyzed with an emphasis on reducing complexity and enhancing maintainability. This analysis, as well as the development

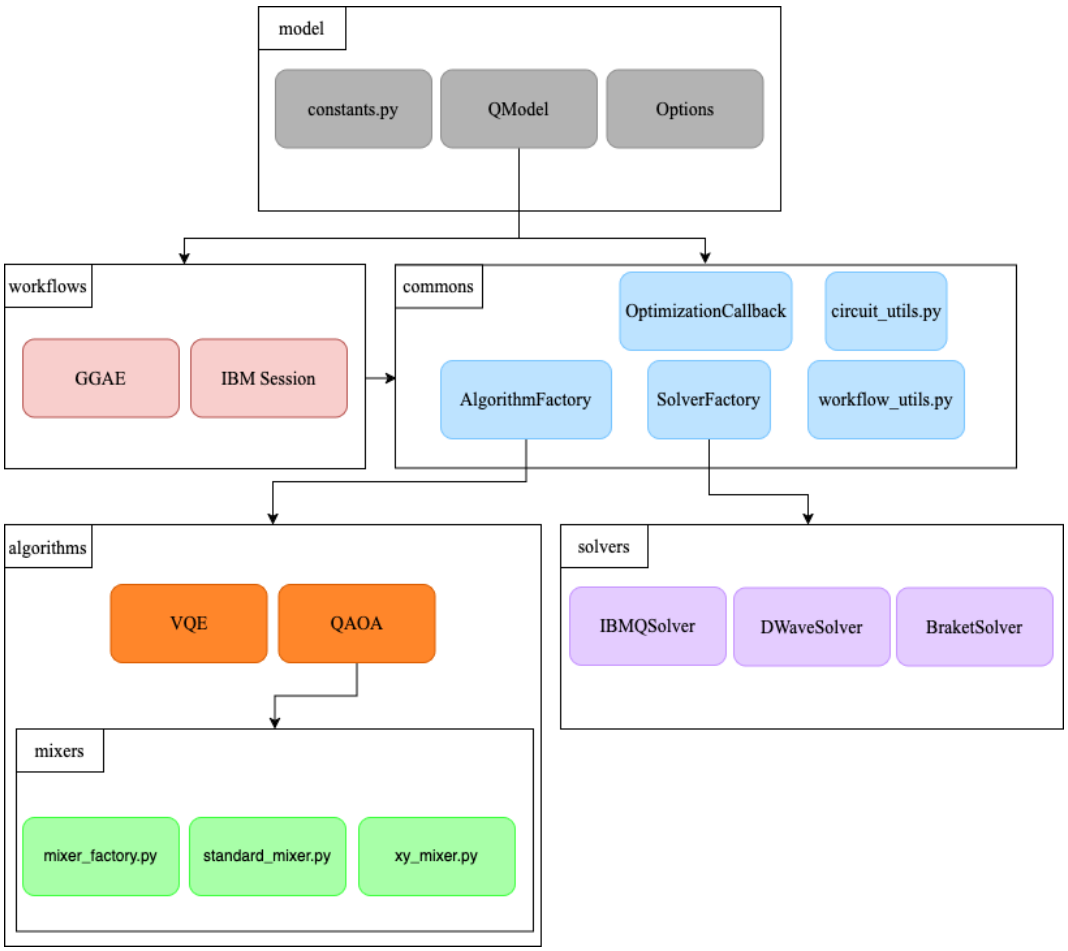

Figure 6.1|: Updated system design of QPLEX

of the functional requirements elicited in this thesis, led to the refactored system design shown in Figure 6.1. The updated design diagram includes unseen classes in some existing modulesone of which, `constants.py`, was previously present in the QPLEX codebase but was omitted in the original manuscriptas well as the introduction of a new module called Workflows. In the diagram, the modules are represented as rectangles labeled in the top-left corner, while submodules (e.g., classes or files within a module) are depicted as colored rectangles with rounded edges. Furthermore, the arrows in the diagram represent associations between modules.

The following subsections provide a detailed explanation of the design changes made to QPLEX, highlighting their purpose and impact on the overall architecture.

6.2.1 Execution Config Class

The purpose of the `QModel` class is to enable users to model a combinatorial optimization problem and solve it using a specified execution configuration (e.g., algorithm, shots, backend device, quantum provider, etc.). These execution options must be passed as arguments

```
24    def solve(self, solver: str = 'classical', provider: str = None, backend: str = None, algorithm: str = "qaoa",
25              ansatz: str = None, p: int = 2, layers: int = 2, optimizer: str = "COBYLA", tolerance: float = 1e-10,
26              max_iter: int = 1000, penalty: float = None, shots: int = 1024, seed: int = 1):
```

Figure 6.2|: Signature of the `solve` method of the `QModel` class, prior to this work.

to the `solve` method, the primary method users call to solve a formulation. Given that QPLEX supports flexible execution configurations across multiple algorithms, providers, and backend devices, the number of required arguments can become large for highly customized execution instances.

Prior to this work, the solve method accepted all execution options as individual parameters in its method signature, as shown in Figure 6.2. This design increased complexity by making the method harder to understand and maintain [83], [84]. Moreover, while default values could be assigned to each parameter to offer flexibility, the need to validate these arguments within the solve method (e.g., ensuring values are within valid ranges) further compounded its complexity, as it extended the method's responsibilities beyond solving the problem formulation.

To address these issues, a dedicated `ExecutionConfig` class[1] was introduced to encapsulate execution parameters. This solution simplifies the solve method by reducing its parameter list and delegating argument validation to the Options class, making the codebase easier to read, maintain, and extend [84].

Figures 6.3 and 6.4 illustrate the difference between the solve method signature before and after the addition of the `ExecutionConfig` class, respectively. Notably, the syntax for users remains largely unchanged, with the only addition being the instantiation of the Options class. This design choice minimizes friction for existing users while improving the maintainability and scalability of QPLEX's codebase.

6.2.2 OptimizationCallback Class and Custom Callbacks

Another addition to the QPLEX codebase is the `OptimizationCallback` class and support for custom callback functions. Callback functions are valuable in optimization routines because they allow users to monitor the progress of an algorithm in a customized manner. By creating their own callback functions, users can perform tasks such as custom logging of algorithm progress, generating visualizations, or dynamically adjusting algorithm hyperparametersdepending on the library's callback implementation. This feature is widely adopted by popular libraries such as MATLAB,[2] TensorFlow,[3] PyTorch,[4] and SciPy.[5]

[1] The source code is available at https://github.com/JuanGiraldo0212/QPLEX/blob/main/qplex/model/execution_config.py

[2] https://www.mathworks.com/help/imaq/creating-and-executing-callback-functions.html

[3] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/Callback

[4] https://pytorch.org/tnt/stable/framework/callbacks.html

[5] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html

```
21      knapsack_model = model_knapsack_problem(values, weights, const)
22
23      execution_params = {
24          "provider": "ibmq",
25          # Change to the desired backend (i.e., ibmq_sherbrooke)
26          "backend": "simulator",
27          "algorithm": "qaoa",
28          "p": 4,
29          "max_iter": 500,
30          "shots": 10000
31      }
32
33      knapsack_model.solve( solver: "quantum", **execution_params)
```

Figure 6.3|: Call to the `solve` method of the `QModel` class, prior to this work. The values of the `execution_params` dictionary are unpacked and passed as arguments to the solve method using the dictionary unpacking operator.

```
3   from qplex.model.execution_config import ExecutionConfig
24      knapsack_model = model_knapsack_problem(values, weights, const)
25
26      execution_config = ExecutionConfig(
27          provider="ibmq",
28          backend="simulator",
29          algorithm="qaoa",
30          p=4,
31          shots=500,
32          max_iter=10000
33      )
34
35      knapsack_model.solve( method: "quantum", execution_config)
```

Figure 6.4|: Call to the `solve` method of the `QModel` class after the addition of the `ExecutionConfig` class

```python
def callback_function(xk):
    print(f"Current parameters: {xk}")

execution_config = ExecutionConfig(
    provider="ibmq",
    backend="simulator",
    algorithm="qaoa",
    p=4,
    shots=1024,
    max_iter=500,
    callback=callback_function
)

knapsack_model.solve( method: "quantum", execution_config)
```

Figure 6.5|: Code snippet showing the use of a custom callback function for the `solve` method

Prior to this work, QPLEX did not provide any user feedback during the optimization process. The absence of intermediate outputs made the process opaque, reducing visibility and making the optimization subroutine resemble a "black box." Results were only displayed after the optimization was complete. Introducing more verbose output, with progress reports or logs, enhances the user experience by increasing transparency. Additionally, enabling custom callback functions further improves flexibility by allowing users to tailor the output to their needs.

As previously mentioned, QPLEX leverages the SciPy library for access to state-of-the-art optimizers. The SciPy optimization subroutine natively supports a callback function parameter, which was integrated into QPLEX in this thesis. Users can now pass a callable function to the `solve` method, which SciPy will invoke during the optimization process. The callback function follows the SciPy framework, reducing the learning curve for users already familiar with the library.

If a user does not provide a custom callback function, QPLEX defaults to using an instance of the newly added `OptimizationCallback` class. This class provides basic logging functionality, such as tracking the iteration number and the current parameters. In the code snippet shown in Figure 6.5, a custom callback function is passed to the `solve` method. Figure 6.6 depicts the resulting console output during execution.

6.2.3 Workflows Module

The concept of workflows in QPLEX was initially limited to the Generalized Gate-Based Algorithm Execution (GGAE) workflow, This workflow is responsible for orchestrating three key steps:

1. Instantiating the selected algorithm to solve the model

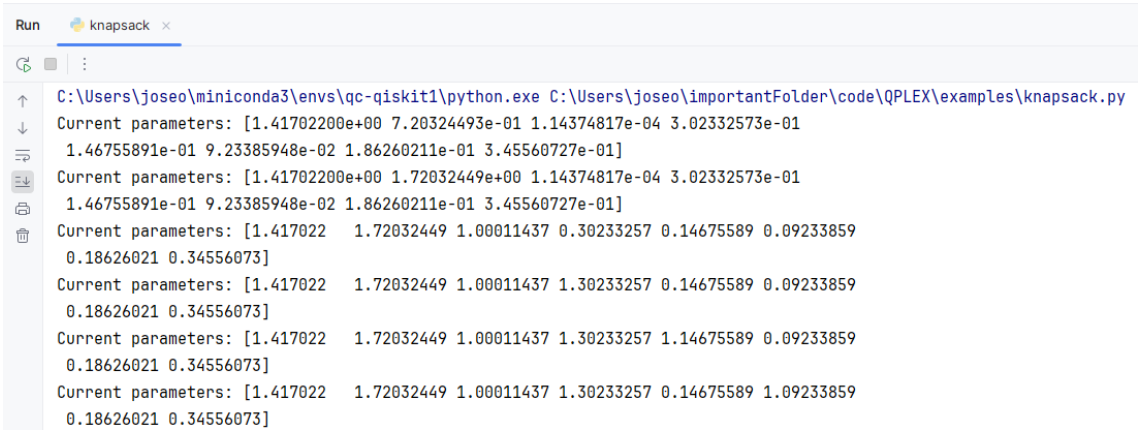

Figure 6.6|: Console output during the execution of a `QModel` formulation using a custom callback that logs the current optimization parameters

2. Initiating the optimization process
3. Returning the optimal counts upon convergence or reaching the maximum number of iterations

However, while designing FR-1 (support for Qiskit Runtime Sessions), it became evident that this feature was incompatible with the GGAE workflow. Qiskit Runtime Sessions replicate the functionality of the GGAE workflow by instantiating a `Session` from the `qiskit_ibm_runtime` package. This Session handles the optimization process, including repeated calls to the QPU, while ensuring prioritized access to the queue.

Given these characteristics, Qiskit Runtime Sessions could not be integrated into the GGAE workflow or the `IBMQSolver` for two reasons:

1. **Provider specificity:** The functionality is not generalizable to all quantum providers, making it unsuitable for inclusion in a generalized workflow.
2. **Solver interface incompatibility:** Adding Qiskit Runtime Sessions to the `IBMQSolver` class would conflict with the base Solver interface.

As a result, the idea of modeling Qiskit Runtime Sessionsand similar methods such as Amazon Hybrid Jobsas a distinct workflow emerged. This new workflow and the possible addition of more workflows in the future motivated the creation of the Workflows module within QPLEX.

6.2.4 Algorithm Factory Class

Prior to this work, algorithm instantiation was handled directly within the GGAE workflow. This approach worked initially since QPLEX only supported two gate-based algorithms, keeping the complexity manageable. However, this design created tight coupling between the workflow and algorithm initialization logic.

The introduction of the Qiskit Runtime Sessions workflow highlighted a potential maintenance challengeany new workflow would need to replicate the algorithm instantiation logic. This duplication would not only violate the DRY (*Don't Repeat Yourself*) principle [85] but also increase the risk of inconsistencies as the codebase evolved.

To address these concerns, we introduced the Algorithm Factory class, implementing the Factory Method design pattern. This pattern provides an interface for creating objects but allows subclasses to alter the type of objects that will be created. In our context, the Algorithm Factory encapsulates the logic for creating algorithm instances, providing a clean interface that workflows can use without needing to know the specific initialization requirements of each algorithm.

As shown in Figure 6.7, the original implementation tightly coupled algorithm creation with the workflow logic. Each algorithm had different initialization requirements, making

```python
def ggae_workflow(model, solver: Solver, shots: int, algorithm: str, optimizer: str, max_iter: int, tolerance: float,
                  ansatz: str, p: int, layers: int, seed: int, penalty: float):
    current_algorithm = None
    if algorithm == "qaoa":
        current_algorithm = QAOA(model, solver, p=p, shots=shots, penalty=penalty, seed=seed)
    elif algorithm == "vqe":
        current_algorithm = VQE(model, solver, layers=layers, shots=shots, penalty=penalty, seed=seed, ansatz=ansatz)
```

Figure 6.7|: The algorithm instantiation code in the GGAE workflow, prior to this work

```python
class AlgorithmFactory:
    """Factory class for creating quantum algorithm instances...."""

    2 usages  joseosso *
    @classmethod
    def get_algorithm(cls, model, config: AlgorithmConfig) -> Algorithm:
        """Creates and returns a quantum algorithm instance based on configuration...."""
        model_constraint_info = get_model_constraint_info(model)

        if config.algorithm == AlgorithmType.QAOA:
            mixer = StandardMixer()
            return QAOA(model, p=config.p, seed=config.seed,
                        penalty=config.penalty, mixer=mixer)

        elif config.algorithm == AlgorithmType.QAO_ANSATZ:
            mixer = config.mixer if config.mixer else (
                MixerFactory.get_mixer(model_constraint_info))
            return QAOA(model, p=config.p, seed=config.seed,
                        penalty=config.penalty, mixer=mixer)

        elif config.algorithm == AlgorithmType.VQE:
            return VQE(model, layers=config.layers, penalty=config.penalty,
                       seed=config.seed, ansatz=config.ansatz)

        raise ValueError(f"Algorithm not supported: {config.algorithm}")
```

Figure 6.8|: The Algorithm Factory class implementation

```python
def ggae_workflow(model, solver: Solver, options: ExecutionConfig):
    """..."""
    shots = options.shots
    verbose = options.verbose
    optimizer = options.optimizer
    callback = options.callback
    max_iter = options.max_iter
    tolerance = options.tolerance

    algorithm_config = AlgorithmConfig(
        algorithm=AlgorithmType(options.algorithm),
        penalty=options.penalty,
        seed=options.seed,
        p=options.p,
        mixer=options.mixer,
        layers=options.layers,
        ansatz=options.ansatz
    )
    algorithm_instance = AlgorithmFactory.get_algorithm(model,
                                                        algorithm_config)
```

Figure 6.9|: The updated algorithm instantiation code in the GGAE workflow

the code difficult to maintain and extend. The new Algorithm Factory class, shown in Figure 6.8, provides a cleaner separation of concerns:

This refactoring provides several benefits:

- Workflows can request algorithm instances through a unified interface
- Algorithm initialization logic is centralized and easier to maintain
- Adding new algorithms requires changes only to the factory class
- Testing becomes more straightforward as algorithm creation is isolated

Additionally, this pattern mirrors the existing Solver Factory in QPLEX, creating a consistent approach to object creation throughout the library. Just as the Solver Factory manages the creation of different quantum provider solvers, the Algorithm Factory handles the instantiation of various quantum algorithms.

The simplified workflow implementations in Figures 6.9 and 6.10 demonstrate how the Algorithm Factory reduces complexity in the workflow layer while maintaining flexibility for future extensions.

6.2.5 Utils Submodules

The submodules with a *utils* suffix, short for utilities, encapsulate functions that are shared across multiple classes in other modules. There are two such submodules in QPLEX: `circuit_utils.py` and `workflow_utils.py`. As their names suggest:

```
11  def ibm_session_workflow(model, ibmq_solver, options: ExecutionConfig):
12      """Executes a quantum optimization workflow using IBM Quantum Runtime...."""
31
32      service = ibmq_solver.service
33
34      verbose = options.verbose
35      optimizer = options.optimizer
36      callback = options.callback
37      max_iter = options.max_iter
38      tolerance = options.tolerance
39
40      algorithm_config = AlgorithmConfig(
41          algorithm=AlgorithmType(options.algorithm),
42          penalty=options.penalty,
43          seed=options.seed,
44          p=options.p,
45          mixer=options.mixer,
46          layers=options.layers,
47          ansatz=options.ansatz
48      )
49
50      algorithm_instance = AlgorithmFactory.get_algorithm(model,
51                                                          algorithm_config)
```

Figure 6.10|: The algorithm instantiation code in the IBM Session workflow

- `circuit_utils.py` provides functionalities related to quantum circuits, which are utilized by all algorithms in the Algorithms module.
- `workflow_utils.py` contains auxiliary functions that support all workflows in the Workflows module, such as GGAE and IBM Session workflows.
- `model_utils.py` contains logic to analyze constraint types in the `QModel` and automatically select an appropriate mixer. For a detailed description, refer to Section 6.3.3.

Most of the functionalities encapsulated in the utility submodules existed in the QPLEX codebase prior to this work but were scattered across multiple files or redundantly implemented in different classes. The primary purpose of these utility submodules is to prevent code repetition by centralizing commonly used functions. This approach reduces complexity by improving code readability, maintainability, and reusability across the library.

6.3 Functional Requirements

For clarity and ease of reference, the functional requirements mentioned in this chapter, previously introduced in Chapter 4, are reiterated below. For the full list, including requirements left for future iterations of the project, refer to Section 4.4.

- **FR-1**: The system shall support executing combinatorial optimization formulations on IBMQ QPUs via Qiskit Runtime Sessions.
- **FR-2**: The system shall support compatibility with Qiskit version 1.0.0.
- **FR-3**: The system shall support quantum circuit execution using Qiskits Sampler primitive on IBM QPUs.
- **FR-5**: The system shall support automatic circuit optimization through the Qiskit transpiler with selectable optimization levels.
- **FR-7**: The system shall support provider-specific execution options accepted by the quantum providers available through the Amazon Braket service.
- **FR-8**: The system shall support executing combinatorial optimization formulations on D-Waves quantum solvers.
- **FR-9**: The system shall support custom minor embeddings for executing combinatorial optimization formulations on D-Wave quantum solvers.
- **FR-11**: The system shall support executing combinatorial optimization formulations using the QAOAnsatz algorithm on gate-based providers.

The following subsections group requirements into the respective QPLEX module they impact.

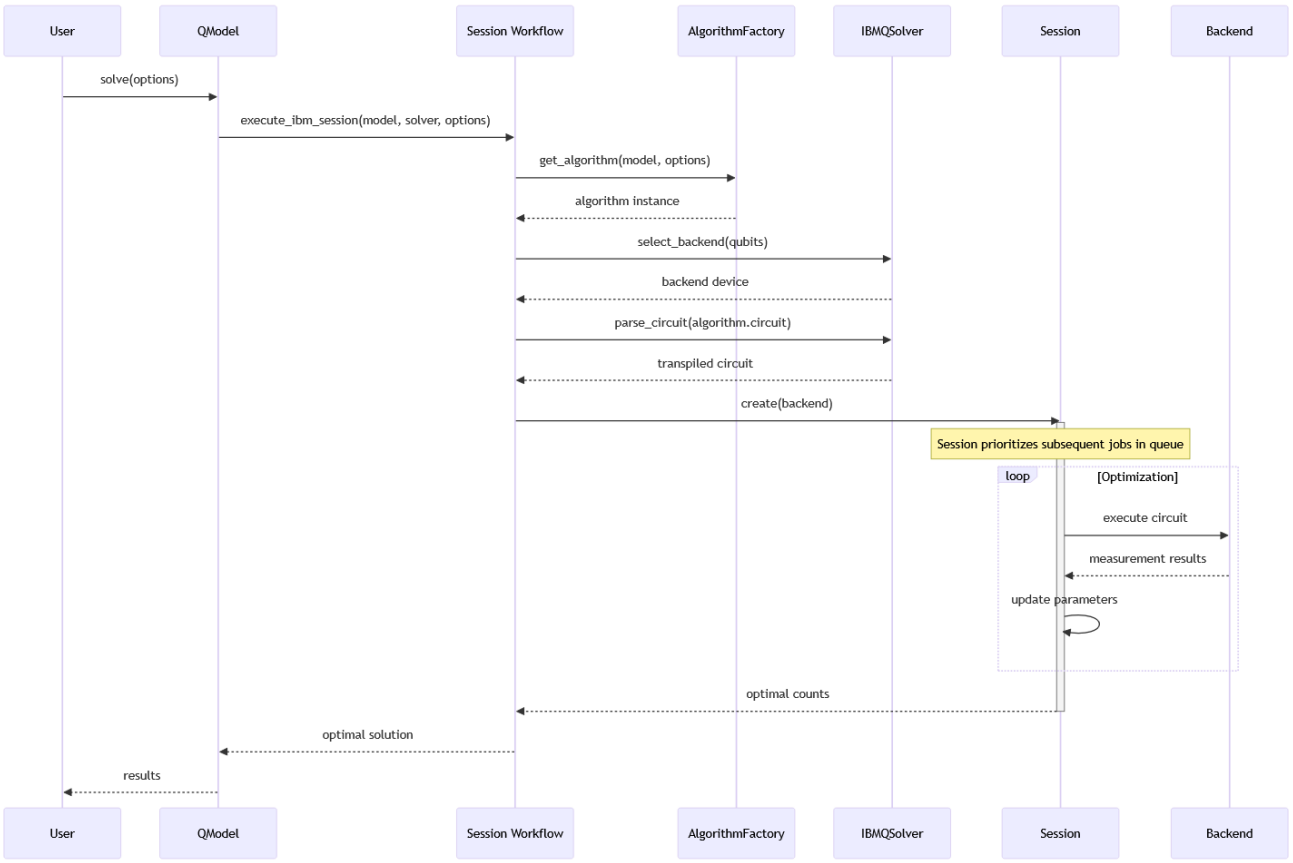

Figure 6.11|: Qiskit Runtime Session UML Sequence Diagram

6.3.1 Workflows Module

FR-1: Qiskit Runtime Sessions

As mentioned in Subsection 6.2.3, Qiskit Runtime Sessions were implemented as a new workflow similar to the existing GGAE workflow. The workflow was implemented as a function that receives three arguments: (1) the `QModel` instance to be solved, (2) an instance of the `IBMQSolver` class and (3) an instance of the `Options` class, described in Subsection 6.2.1. The workflow instantiates the selected algorithm using the given options and model instance. It then proceeds to utilize the solver instance to parse the algorithm circuit into the desired input for the selected IBM backend device. Afterward, it defines the cost function and opens the Qiskit Runtime `Session` that performs the optimization loop and returns the optimal counts. Figure 6.11 shows a diagram with this sequence of actions.

```
10      def __init__(self, token: str, shots: int, backend: str):
11          self.shots = shots
12          self.backend = backend
13          IBMQ.save_account(token, overwrite=True)
14          IBMQ.load_account()
```

Figure 6.12|: Code snippet for the `__init__` method of the IBMQ solver, prior to this work

6.3.2 Solvers Module

FR-2: Support for Qiskit 1.0.0 and FR-3: Qiskit Sampler Primitive

The last contribution to QPLEX prior to this work was made on October 28, 2023.[6] At that time, the Qiskit SDK was at version 0.25.3.[7] Since then, IBM has released Qiskit 1.0.0, a milestone update introducing significant changes to enhance the library's scalability, stability, and extensibility. This update included major refactoring of the library's modules and introduced updated components, such as version 2 of the Sampler and Estimator primitives [86].

The Qiskit 1.0.0 release is incompatible with previous versions, rendering older implementations non-functional without updates. Consequently, since QPLEX had not been maintained beyond version 0.25.3 of Qiskit, it became incompatible with the new release. To enable QPLEX to remain relevant and capable of interfacing with the latest features of the Qiskit library, it was essential to add support for Qiskit 1.0.0.

Fortunately, the modular design of the QPLEX library minimizes overall complexity, making it easier to refactor or modify individual modules. For instance, the Solvers module, which includes the `IBMQSolver` class responsible for interacting with IBM QPUs through the Qiskit API, could be updated to add support for Qiskit 1.0.0 without the need for modifying other modules.

The QPLEX IBMQSolver class manages configuring the IBM Quantum account to be used to submit quantum circuits for execution on IBM QPUs via the cloud. Previously, this was achieved using the `IBMQ` class from the `qiskit.providers.ibmq` package. Figure 6.12 illustrates the `__init__` method of the IBMQSolver class prior to this work, showing the usage of the IBMQ class. However, this package hierarchy required all Qiskit installations to include the providers module, even when users did not intend to submit circuits to IBMQ hardware.

As noted in the Qiskit 0.40 release notes, Qiskit has transitioned to a model where the qiskit package contains only the core functionality for building and compiling quantum circuits, programs, and applications. Any functionality related to specific hardware or simulators is now encapsulated in separate packages [87]. Consequently, all IBMQ-related functionalities were moved to the `qiskit_ibm_runtime` package, which can be installed independently of the core qiskit package. This updated workflow implemented in QPLEX's

[6] https://github.com/JuanGiraldo0212/QPLEX/commit/d7b644403e3e5f5bbe5af1685dfdfbcc5238842c
[7] https://github.com/Qiskit/qiskit/releases/tag/0.25.3

```python
    def __init__(self, token: str, shots: int, backend: str,
                 optimization_level: int):
        """
        Initializes the IBMQSolver with the specified token, number of
        shots, and backend.

        Parameters
        ----------
        token : str
            The IBMQ API token for authentication.
        shots : int
            The number of shots for the quantum experiment.
        backend : str
            The backend to use for solving the problem,
            which can be an IBMQ device or a local simulator.
        optimization_level : int
            The desired optimization level for the Qiskit circuit.
        """
        self.shots = shots
        if backend is None:
            print('No backend specified. Using least busy...')
            self._backend = ''
        else:
            self._backend = backend
        QiskitRuntimeService.save_account(channel="ibm_quantum",
                                          token=token, overwrite=True)
        self.service = QiskitRuntimeService()
        self.optimization_level = optimization_level

```

Figure 6.13|: Code snippet for the updated `__init__` method of the IBMQ solver.

IBMQSolver is depicted in Figure 6.13 and involves the use of the `QiskitRuntimeService` class from the qiskit_ibm_runtime package. An instance of this class, referred to as service, is created and subsequently used throughout the IBMQSolver class to facilitate communication with the IBMQ cloud.[8]

Migrating to the QiskitRuntimeService class required updates to the workflow for submitting circuits to the IBMQ cloud. Specifically, modifications to the `solve` method of the IBMQSolver class were required. Previously, this method utilized the `execute` function from the qiskit package to submit circuits, as shown in Figure 6.14. However, IBM now recommends using the Sampler and Estimator primitives for running circuits and retrieving results. Given that the interface of the base Solver class of the QPLEX library requires the solve method to return execution counts for a given circuit, the Sampler primitive was the logical choice. As shown in Figure 6.15, the updated solve method instantiates the Sampler and delegates circuit execution to an auxiliary `run` method, introduced for improved simplicity and modularity, depicted in Figure 6.16. The updated solve method returns ex-

[8]Qiskit provides a migration guide for transitioning from the qiskit.providers workflow to the qiskit_ibm_runtime workflow: https://docs.quantum.ibm.com/migration-guides/qiskit-runtime-from-ibmq-provider

```
16    def solve(self, model: str):
17        qc = self.parse_input(model)
18        backend = self.select_backend(qc.num_qubits)
19        response = execute(qc, backend, shots=self.shots).result()
20        counts = self.parse_response(response)
21        return counts
```

Figure 6.14|: Code snippet for the `solve` method of the IBMQ solver, prior to this work

ecution counts in the same format as the previous implementation, ensuring compatibility with other QPLEX modules.

Additionally, the `select_backend` method was updated to utilize the QiskitRuntimeService class and consider the qubits argument, which specifies the number of qubits required by the circuit to be executed. As noted in Chapter 4, this parameter was previously ignored. The updated method, shown in Figure 6.17, ensures that the selected QPU meets the circuits qubit requirements. With these updates, the IBMQSolver class is fully compatible with Qiskit 1.0.0 and adheres to the design principles of QPLEX.

FR-5: Automatic Qiskit Circuit Optimization

The Qiskit framework provides a `PassManager` class, which can be used to transpile a quantum circuit with a specific optimization level tailored to a target backend. According to IBM's documentation,[9] the optimization levels are described as follows:

- **Level 0:** no optimization
- **Level 1:** light optimization
- **Level 2:** heavy optimization
- **Level 3:** even heavier optimization

Higher optimization levels typically result in longer transpilation times due to the increased computational effort required.

In this work, Qiskit circuit optimization was integrated as an execution option configurable via the `QModel`'s `solve` method. During execution, this option is accessed as an instance variable of the `IBMQSolver` class. Line 81 in Figure 6.15 illustrates the use of the `optimization_level` field during the circuit transpilation process.

FR-7: Amazon Braket Execution Options

The `solve` method of the Amazon Braket solver class, `BraketSolver`, interfaces with the `braket` library's API to manage the process of input parsing, circuit execution, and response parsing. The implementation of this method is shown in Figure 6.18.

[9] https://docs.quantum.ibm.com/api/qiskit/0.43/qiskit.transpiler.preset_passmanagers.generate_preset_pass_manager

```python
def solve(self, model: str) -> dict:
    """
    Solves the given problem formulation using the specified backend.

    Parameters
    ----------
    model : str
        The quantum circuit as an OpenQASM string to be
        executed.

    Returns
    -------
    dict
        A dictionary containing the measurement counts from the
        backend.
    """
    qc = self.parse_input(model)
    backend = self.select_backend(qc.num_qubits)
    pass_manager = generate_preset_pass_manager(backend=backend,
                                                optimization_level=
                                                self.optimization_level)
    isa_circuit = pass_manager.run(qc)

    if self._backend == 'simulator':
        raw_counts = backend.run(isa_circuit).result().get_counts()
    else:
        sampler = Sampler(backend)
        raw_counts = self.run(isa_circuit, sampler)
    counts = self.parse_response(raw_counts)
    return counts
```

Figure 6.15|: Code snippet for the updated `solve` method of the IBMQ solver

```python
def run(self, qc, sampler):
    """
    Execute the given quantum circuit using the specified sampler.

    Parameters
    ----------
    qc : QuantumCircuit
        The quantum circuit to be executed.
    sampler : Any
        The sampler used to run the quantum circuit. This could be a
        quantum simulator or a quantum hardware backend that supports
        the `run` method.

    Returns
    -------
    dict
        A dictionary containing the raw measurement counts, where the
        keys are the bitstrings and the values are their respective counts.
    """

    pub = (qc,)
    result = sampler.run([pub], shots=self.shots).result()
    data = result[0].data
    bits = data.c
    raw_counts = bits.get_counts()
    return raw_counts
```

Figure 6.16|: Code snippet for the newly added `run` method of the IBMQ solver

```python
def select_backend(self, qubits: int) -> AerSimulator | BackendV2:
    """
    Selects the appropriate backend based on the number of qubits and
    the specified backend name.

    Parameters
    ----------
    qubits : int
        The number of qubits in the quantum circuit.

    Returns
    -------
    Any
        The selected backend, which could be an IBMQ device or a  local
        simulator.
    """
    if self._backend != "simulator":
        if self._backend == "":
            return self.service.least_busy(min_num_qubits=qubits)
        return self.service.backend(self._backend)
    return AerSimulator()
```

Figure 6.17|: Code snippet for the updated `select_backend` method of the IBMQ solver

```
43      def solve(self, model: str) -> dict:
44          """
45          Solves the given problem formulation using the specified backend.
46
47          Parameters
48          ----------
49          model : str
50              The quantum circuit as an OpenQASM string to be executed.
51
52          Returns
53          -------
54          dict
55              A dictionary containing the measurement counts from the backend.
56          """
57          qc = self.parse_input(model)
58          backend = self.select_backend(0)
59          response = (backend.run(qc, shots=self.shots,
60                                  device_parameters=self.device_parameters)
61                      .result())
62          counts = self.parse_response(response)
63          return counts
```

Figure 6.18|: Code snippet for the updated `solve` method of the Amazon Braket solver

To submit circuits for execution, the `BraketSolver` utilizes the `run` method of the `AwsDevice` class. This method accepts an argument called `device_parameters`, which allows users to specify configuration parameters unique to the selected quantum device. The use of this argument is evidenced in line 60 of Figure 6.18. Since the quantum devices available through the Braket platform support different configurable parameters, it is recommended to refer to the respective device documentation for details on their usage.

These execution options are now configurable through QPLEX when calling the `QModel`'s `solve` method.

FR-8: D-Wave Quantum Solvers Support

To enable the execution of D-Wave models on the Leap platform's quantum solvers, such as the Advantage QPU, it was necessary to distinguish between QPUs and Hybrid Solvers as separate backends for execution. To achieve this, an instance variable called `backend` was added to the `DWaveSolver` class, allowing users to specify the intended backend (i.e., a QPU or a Hybrid Solver). Additionally, a configuration field for selecting a QPU based on its topology was introduced to enhance flexibility.

The updated initializer method of the `DWaveSolver` class is depicted in Figure 6.19. Prior to this work, the default initializer was used since instance variables were considered unnecessary for the implementation of this class. In Figure 6.19, we can also evidence other configuration fields such as `time_limit`, `num_reads` and `embedding`. These fields further enhance the flexibility of the execution of combinatorial optimization formulations on D-Wave hardware through the QPLEX interface. The `time_limit` field configures the time limit for the hybrid solver to work on the problem, while `num_reads` specifies the number of times the quantum annealing process is repeated for a given problem. The `embedding`

```
20      def __init__(self, token, time_limit, num_reads, topology,
21                   embedding, backend):
22  >       """Initialize the DWaveSolver with the specified configuration...."""
42          super().__init__()
43          self.token = token
44          self.time_limit = time_limit
45          self.num_reads = num_reads
46          self.topology = topology
47          self.embedding = embedding
48          if backend is None:
49              print("No backend specified for D-Wave solver. Using hybrid "
50                    "solver...")
51              self._backend = 'hybrid_solver'
52          else:
53              self._backend = backend
```

Figure 6.19|: Code snippet for the updated `__init__` method of the D-Wave solver

parameter allows users to use a custom minor embedding for the current problem. This is further discussed in the next part, where the design and implementation of FR-9 (Custom Minor Embedding Support) is explained in detail.

- The `time_limit` field specifies the maximum time the hybrid solver can spend solving the problem.

- The `num_reads` field defines the number of times the quantum annealing process is repeated for a given problem.

- The `embedding` parameter allows users to provide a custom minor embedding for the problem, which is discussed in detail in the next section on the design and implementation of FR-9 (Custom Minor Embedding Support).

Once all execution options have been configured, the `solve` method[10] of the `DWaveSolver` class proceeds with the standard workflow: parsing the input, executing the formulation on the selected device, and parsing the response. However, an additional step has been introduced to select the appropriate backend based on the users configuration.

This selection process is handled by the newly added `select_backend` method, shown in Figure 6.20. As illustrated, the method instantiates the appropriate sampler depending on the specified configuration and returns it to the `solve` method for execution.

[10]The source code for the `DWaveSolver` class, including the `solve` method, is available at https://github.com/JuanGiraldo0212/QPLEX/blob/development/qplex/solvers/dwave_solver.py

```
220     def select_backend(self, parsed_model, model_type) -> Any:
221         """Select the appropriate backend for the given model type...."""
252         hybrid_samplers = {
253             VAR_TYPE['C']: LeapHybridCQMSampler,
254             VAR_TYPE['I']: LeapHybridDQMSampler,
255             VAR_TYPE['B']: LeapHybridBQMSampler,
256         }
257
258         if self._backend == 'hybrid_solver':
259             sampler_class = hybrid_samplers.get(model_type)
260             return sampler_class(token=self.token)
261
262         elif self._backend == 'd-wave_sampler':
263             # User requested a DWaveSampler, but model is not QUBO-compatible.
264             if model_type in (VAR_TYPE['C'], VAR_TYPE['I']):
265                 print(
266                     "The selected backend requires a QUBO-compatible model, "
267                     "but the given model contains constraints or discrete "
268                     "variables.\nSwitching to an appropriate Hybrid Solver to "
269                     "handle this model type..."
270                 )
271                 sampler_class = hybrid_samplers[model_type]
272                 return sampler_class(token=self.token)
273
274             qpu = DWaveSampler(solver=dict(topology__type=self.topology),
275                                token=self.token)
276             self._backend = qpu.solver.name
277             print(
278                 f"Selected {self._backend} with {len(qpu.nodelist)} qubits.")
279
280             if self.embedding is None:
281                 return AutoEmbeddingComposite(qpu)
282             return FixedEmbeddingComposite(qpu, self.embedding)
283
284         raise ValueError(f"Unsupported backend: {self._backend}")
```

Figure 6.20|: Code snippet for the newly added `select_backend` method of the D-Wave solver

FR-9: Custom Minor Embedding Support

The use of a D-Wave Sampler requires embedding the problem graph onto the connectivity graph of the selected device. As discussed in Chapter 4, enabling users to supply a custom embedding enhances the versatility of the QPLEX library, as it allows for greater flexibility compared to relying solely on the default embedding algorithm provided by the Ocean SDK.

The Ocean SDK offers various classes, referred to as composites, for wrapping a D-Wave Sampler object to perform embeddings. These composites differ in their embedding approaches. For instance:

- The `AutoEmbeddingComposite`[11] submits the binary quadratic model directly to the

[11] https://docs.ocean.dwavesys.com/projects/system/en/stable/reference/composites.html#autoembeddingcomposite

sampler and attempts to embed the problem only if an exception is raised, indicating that the model's graph is incompatible with the sampler's connectivity graph.

- The `FixedEmbeddingComposite`[12] allows users to specify a custom minor embedding for the sampling process. This composite is particularly useful for manually controlling the embedding or reusing a known good embedding for a given problem.

After evaluating the available composites[13], the `FixedEmbeddingComposite` and `AutoEmbeddingComposite` were identified as suitable for implementing FR-9. As shown on line 282 in Figure 6.20, the `FixedEmbeddingComposite` wraps the initialized `DWaveSampler` and applies the custom embedding. This embedding, configured as an instance variable of the `DWaveSolver` class, can be specified as an execution parameter when calling the `QModel`'s `solve` method.

If the user does not provide a custom minor embedding, the `AutoEmbeddingComposite` is used as the default.

6.3.3 Algorithms Module

FR-11: QAOAnsatz

The Quantum Alternating Operator Ansatz (QAOAnsatz) generalizes the original QAOA algorithm by allowing flexible mixer operators beyond the standard transverse field mixer. In QAOA, the mixer Hamiltonian enables transitions between different states in the solution space through quantum operations. The conventional QAOA implementation uses a transverse field mixer composed of R_X rotations, as shown in Equation 6.1:

$$U_M(\beta) = e^{-i\beta H_M} = \prod_{i=0}^{n-1} R_{X_i}(2\beta) \tag{6.1}$$

where n is the number of qubits and β is the mixing angle parameter.

The QAOAnsatz framework extends this approach by allowing mixer operators that respect the constraints of the optimization problem while enabling efficient exploration of the feasible solution space. This flexibility can potentially improve the algorithm's performance for certain problem classes by incorporating domain knowledge into the quantum operations.

The relationship between QAOA and QAOAnsatz naturally influenced our software design decisions. Since QAOAnsatz generalizes QAOA by allowing flexible mixer operators, we implemented this as a single class where the mixer operator is configurable. This design

[12] https://docs.ocean.dwavesys.com/projects/system/en/stable/reference/composites.html#fixedembeddingcomposite

[13] https://docs.ocean.dwavesys.com/projects/system/en/stable/reference/composites.html

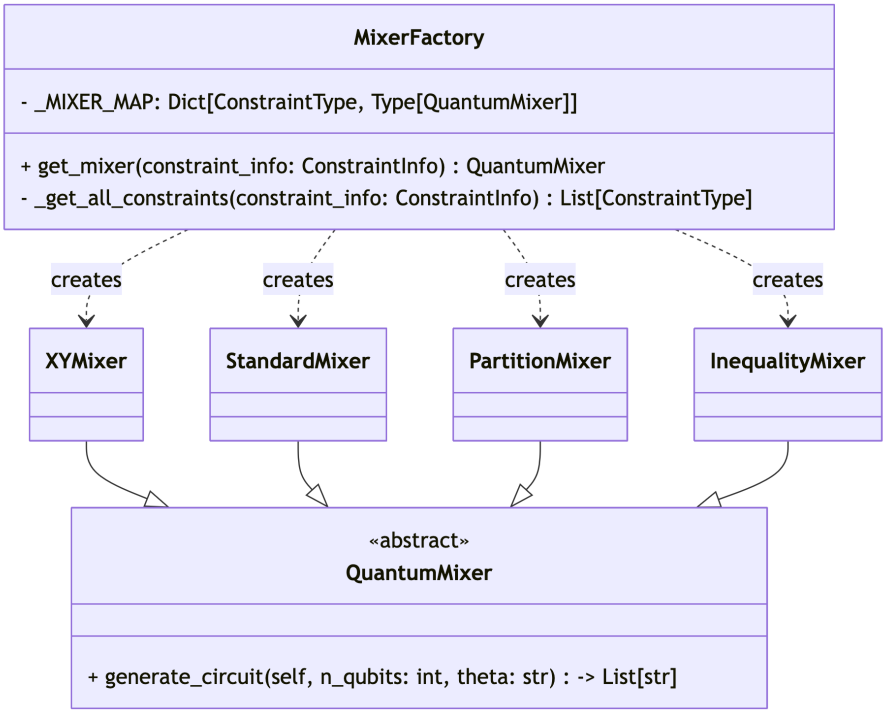

Figure 6.21|: The UML class diagram for the `MixerFactory` class and other classes that it interacts with throughout its lifecycle

choice makes the mixer an instance attribute of the QAOA class, enabling users to instantiate the class with either the conventional R_X mixer for standard QAOA or alternative mixers for QAOAnsatz implementations.

To achieve this flexibility while maintaining clean separation of concerns, we introduced the abstract `QuantumMixer` class. This class defines a common interface that all mixer implementations must follow by implementing a single method: `generate_circuit`. This method produces the mixer portion of the quantum circuit as an OpenQASM3 string, which the QAOA algorithm class then incorporates into the complete circuit.

Figure 6.21 provides a UML class diagram illustrating the relationships between the key components after these additions. The diagram shows the abstract `QuantumMixer` class hierarchy, including its concrete implementations, and demonstrates how it integrates with the larger QPLEX architecture through the factory pattern.

At the center of this design is the `MixerFactory` class, which follows the same factory pattern used elsewhere in QPLEX for algorithms and solvers. The `MixerFactory` works with the `AlgorithmFactory` to create fully configured QAOA instances. This interaction is particularly important when handling the automatic selection of appropriate mixers based on problem constraints.

At the time of writing, only two concrete mixer implementations have been fully realized: the standard Rx mixer used in the original QAOA algorithm and the XY mixer, an

implementation shown to preserve the subspace of one-hot states [88], [89]. The additional mixer classes shown in Figure 6.21 (such as PartitionMixer and Inequality) are not yet implemented but are included in the diagram to illustrate the extensibility of the Factory design pattern. These placeholder classes represent potential future mixers that could be developed for specific constraint types.

To support this automatic selection, we introduced a utility function in the new `model_utils.py`[14] module that analyzes constraint patterns within a `QModel`. As shown in Figure 6.8, this analysis occurs within the `AlgorithmFactory`'s `get_algorithm` method. On line 89, the method first extracts constraint information from the model using these utility functions. Then, on line 98, it passes this information to the `MixerFactory`, allowing it to instantiate the most appropriate mixer for the given problem structure.

While this automatic selection handles many common cases, the design remains flexible enough to accommodate custom implementations. Users can create specialized mixing strategies by extending the `QuantumMixer` class and overriding the `generate_circuit` method, giving them full control over the mixing operations while maintaining compatibility with the broader QPLEX framework.

To further clarify this sequence of actions, the UML sequence diagram depicted in Figure 6.22 shows the process of getting an algorithm instance orchestrated by a QPLEX workflow. In the diagram, the workflow calls the `get_algorithm` method from the algorithm factory and receives the ready-to-use instance at the end. The diagram illustrates several key design patterns working in concert: the Factory pattern handles object creation, while inheritance relationships ensure type safety through polymorphism. When creating a QAOAnsatz instance, the factory coordinates with a secondary factory (the mixer factory) to construct appropriate mixer operators based on the problem constraints. This two-factory approach allows QPLEX to decouple the algorithm configuration from the constraint handling logic, making it easier to extend the framework with new types of constraints or mixers without modifying the core algorithm implementations. The sequence concludes with the initialization of the base Algorithm class before returning the concrete QAOA instance to the workflow.

This modular design maintains full compatibility with QPLEX's existing Algorithm interface while introducing powerful customization capabilities. Users can now experiment with different mixing strategies optimized for their specific combinatorial optimization problems, potentially improving solution quality or convergence speed.

[14] The source code can be found at https://github.com/JuanGiraldo0212/QPLEX/blob/development/qplex/utils/model_utils.py

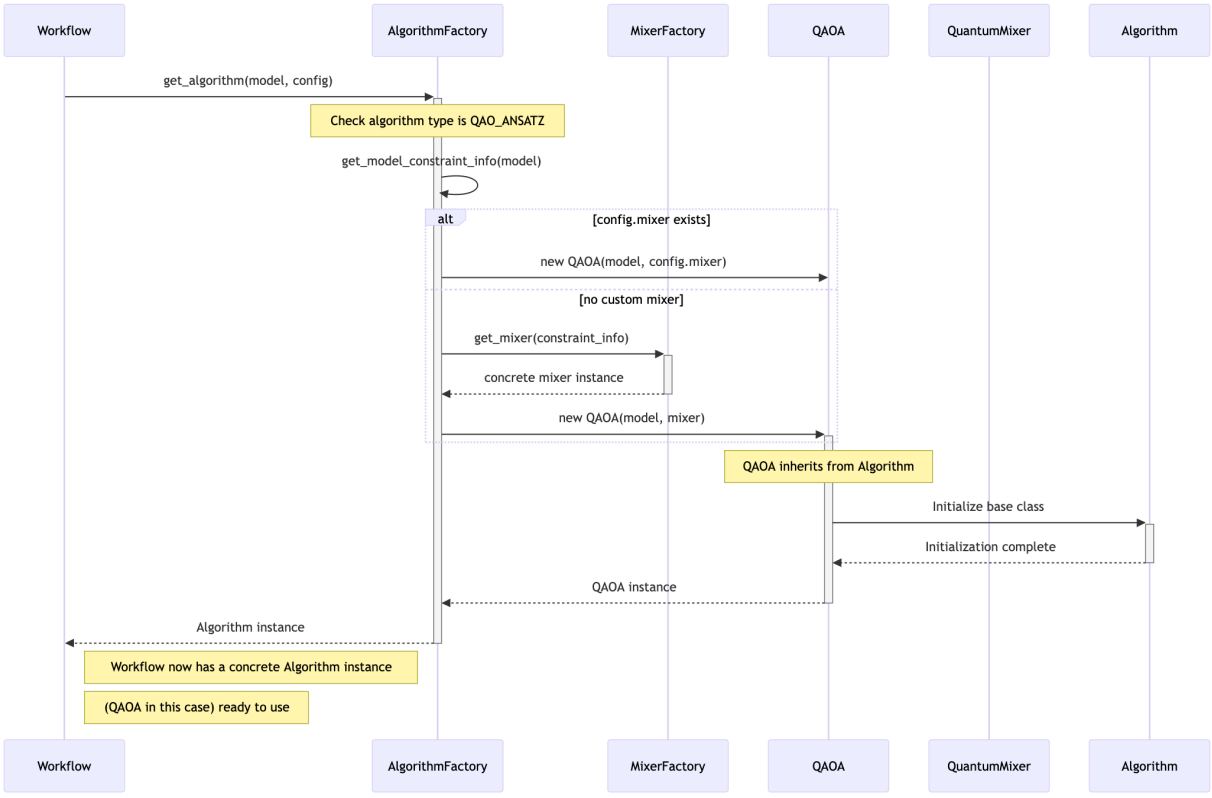

Figure 6.22|: The UML sequence diagram showing the process of getting an algorithm instance orchestrated by a QPLEX workflow

6.4 Non-Functional Requirements

Similarly, for clarity and ease of reference, the non-functional requirements mentioned in this chapter, previously introduced in Section 5.5.1, are reiterated below.

- **NFR-1:** The system shall have consistent, detailed and readable documentation for all modules, classes, methods, and functions by including comprehensive docstrings that adhere to established standards (e.g., PEP 257) for clarity and maintainability.
- **NFR-2:** The system shall provide an API specification page that is automatically updated through CI/CD workflows whenever new documentation is added, ensuring the documentation remains current and accessible.
- **NFR-3:** The system shall include a standardized contribution guideline to ensure consistency and maintainability across the library, providing clear instructions for developers contributing to QPLEX.
- **NFR-4:** The system shall include the design and implementation of comprehensive unit tests to cover all QPLEX modules, following a testing framework and ensuring the robustness and reliability of the library.
- **NFR-5:** The system shall incorporate a standardized test coverage metric, accompanied by a coverage threshold enforced through CI/CD automation tools, to guarantee adequate validation of the codebase.
- **NFR-6:** The system shall implement GitHub Actions workflows to automate key stages of the software development life cycle, including code compilation, testing, building, and deployment.
- **NFR-7:** The system shall package and deploy QPLEX to PyPI, enabling users to install the library via the Pip package manager.

The following subsections group non-functional requirements into categories.

6.4.1 Documentation

NFR-1: Code Documentation

Prior to this work, as discussed in Section 5.1, the QPLEX codebase lacked comprehensive documentation. To address this limitation, we implemented a thorough documentation process following the PEP 257 standard before proceeding with the functional requirements described in this chapter. This documentation effort encompassed all classes, methods, and modules, with additional inline comments added where complex logic warranted further clarification.

The documentation process adhered to the following standards:

- **Triple-quoted Strings:** All docstrings must use triple-quoted string syntax, ensuring consistent parsing and formatting.
- **Summary Lines:** Each class and method documentation begins with a single-line summary that concisely describes its purpose.
- **Parameters and Returns:** Method documentation includes explicitly defined sections for parameters and return values, complete with type hints. These sections are clearly demarcated with section titles followed by dashed lines.
- **Exceptions:** Methods must declare any exceptions they might raise, ensuring users understand potential error conditions.

This documentation standard has been integrated into the project's development workflow. All new contributions must adhere to these documentation requirements before merging, ensuring the codebase maintains consistent and comprehensive documentation. This requirement facilitates automatic inclusion of new functionality documentation in the API specification, which is detailed in the following section.

Figures 6.23 and 6.24 illustrate the impact of this documentation effort through a comparison of the `IBMQSolver` class before and after documentation.[15]

NFR-2: API Specification

To enhance accessibility and maintainability of QPLEX's documentation, we implemented an automated API specification generation process using Sphinx, a powerful documentation generator chosen for its extensive documentation and minimal configuration requirements.

Sphinx operates by analyzing docstrings within the codebase to generate a structured API documentation tree. The tool is installed as a Python dependency through standard package managers, after which it requires configuration to properly parse and format the documentation. Sphinx provides a quickstart utility that streamlines this configuration process through an interactive prompt, generating the necessary configuration files based on user responses.

The Sphinx configuration process establishes a standardized directory structure within the project folder, as illustrated in Figure 6.25. This structure includes the *docs* organizational directoriy and *ReStructuredText* (.rst) files that contain the documentation for each module in the library. The resulting directory hierarchy directly influences the structure and navigation of the generated API documentation.

The effectiveness of Sphinx's documentation generation depends entirely on the presence and quality of docstrings throughout the codebase, underscoring the importance of maintaining comprehensive documentation standards as described in NFR-1.

NFR-3: Standardized Contribution Guideline

[15] The complete source code for the `IBMQSolver` class can be found at.https://github.com/JuanGiraldo0212/QPLEX/blob/main/qplex/solvers/ibmq_solver.py

```python
class IBMQSolver(Solver):

    def __init__(self, token: str, shots: int, backend: str):
        self.shots = shots
        self.backend = backend
        IBMQ.save_account(token, overwrite=True)
        IBMQ.load_account()

    def solve(self, model: str):
        qc = self.parse_input(model)
        backend = self.select_backend(qc.num_qubits)
        response = execute(qc, backend, shots=self.shots).result()
        counts = self.parse_response(response)
        return counts

    def parse_input(self, circuit: str):
        circuit = """
        OPENQASM 3.0;
        include "stdgates.inc";
        """ + circuit
        qc = qiskit.qasm3.loads(circuit)
        return qc

    def parse_response(self, response):
        response = response.get_counts()
        parsed_response = {}
        for sample, count in response.items():
            x = [int(bit) for bit in reversed(sample)]
            parsed_response["".join(str(n) for n in x)] = count
        return parsed_response

    def select_backend(self, qubits: int) -> Backend:
        if self.backend != "simulator":
            provider = IBMQ.get_provider(hub='ibm-q')
            return provider.get_backend(self.backend)
        return Aer.get_backend("qasm_simulator")
```

Figure 6.23|: A code snapshot of the `IBMQSolver` class, prior to this work

```python
class IBMQSolver(Solver):
    """
    Solver for IBMQ. Can execute circuits on IBM backend or local simulators.

    Attributes
    ----------
    shots : int
        The number of shots for the quantum experiment.
    _backend : str
        The name of the backend to be used, which can be an IBMQ
        device or a local simulator.
    service : QiskitRuntimeService
        The Qiskit runtime service instance for interacting with IBMQ's
        backend.
    optimization_level : int
        The desired optimization level for the Qiskit circuit.
    """

    def __init__(self, token: str, shots: int, backend: str,
                 optimization_level: int):
        """
        Initializes the IBMQSolver with the specified token, number of
        shots, and backend.

        Parameters
        ----------
        token : str
            The IBMQ API token for authentication.
        shots : int
            The number of shots for the quantum experiment.
        backend : str
            The backend to use for solving the problem,
            which can be an IBMQ device or a local simulator.
        optimization_level : int
            The desired optimization level for the Qiskit circuit.
        """
        self.shots = shots
        if backend is None:
            print('No backend specified. Using least busy...')
            self._backend = ''
        else:
            self._backend = backend
        QiskitRuntimeService.save_account(channel="ibm_quantum",
                                          token=token, overwrite=True)
        self.service = QiskitRuntimeService()
        self.optimization_level = optimization_level
```

```python
    def solve(self, model: str) -> dict:
        """
        Solves the given problem formulation using the specified backend.

        Parameters
        ----------
        model : str
            The quantum circuit as an OpenQASM string to be
            executed.

        Returns
        -------
        dict
            A dictionary containing the measurement counts from the
            backend.
        """
        qc = self.parse_input(model)
        backend = self.select_backend(qc.num_qubits)
        pass_manager = generate_preset_pass_manager(backend=backend,
                                                    optimization_level=
                                                    self.optimization_level)
        isa_circuit = pass_manager.run(qc)

        if self._backend == 'simulator':
            raw_counts = backend.run(isa_circuit).result().get_counts()
        else:
            sampler = Sampler(backend)
            raw_counts = self.run(isa_circuit, sampler)
        counts = self.parse_response(raw_counts)
        return counts

    def run(self, qc, sampler):
        """
        Executes the given quantum circuit using the provided sampler.

        Parameters
        ----------
        qc : QuantumCircuit
            The quantum circuit to run.
        sampler : Sampler
            The sampler instance to use for executing the circuit.

        Returns
        -------
        dict
            A dictionary with the raw measurement counts.
        """
        pub = (qc,)
```

Figure 6.24|: An image showing two code snapshots of the `IBMQSolver` class side by side after implementing the contributions described in this thesis. The code is read from left to right and top to bottom.

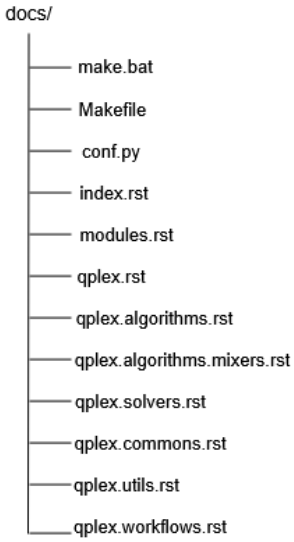

Figure 6.25|: The directory and *ReStructuredText* file structure for the documentation files generated by the Sphinx tool to produce the API specification page

```python
def method_name(self, param1: type, param2: type) -> return_type:
    """Short description of method functionality.

    Detailed description of the method's purpose and behavior.

    Parameters
    ----------
    param1 : type
        Description of first parameter
    param2 : type
        Description of second parameter

    Returns
    -------
    return_type
        Description of return value

    Raises
    ------
    ExceptionType
        Description of when/why this exception occurs
    """
```

Figure 6.26|: An example of the documentation docstrings for a method following PEP257 and the standard style used throughout QPLEX

To further facilitate extensions to the QPLEX library, a comprehensive and structured contribution guide was documented in the repository's `CONTRIBUTING.md` file, as is customary in open-source software projects. The topics covered in this document are:

Development Environment Setup

- Instructions for creating a virtual environment
- Required dependencies and their versions

Code Style and Documentation Standards

- Adherence to PEP 8 for Python code style
- Mandatory comprehensive docstrings following PEP 257
- Required module-level, class-level, and function-level documentation
- Examples of properly documented code, such as the code snapshot depicted in Figure 6.26

Testing Requirements

- Instructions for running the test suite using pytest
- Guidelines for writing new tests

```python
class TestClass:
  def test_method_name():
      """Test description."""
      # Arrange
      input_data = ...

      # Act
      result = method_name(input_data)

      # Assert
      assert result == expected_output
```

Figure 6.27|: An example of the unit test structure for a method in QPLEX

- Required test coverage thresholds (minimum 80% branch coverage)
- Examples of well-structured tests, such as the code snapshot depicted in Figure 6.27

Pull Request Process

- Branch naming conventions (e.g., feature/, bugfix/, docs/)
- Required elements in PR descriptions
- CI/CD pipeline requirements that must pass
- Merge strategy preferences

6.4.2 Testing

NFR-4: Unit Tests

Testing Framework Selection

As mentioned previously in Chapter 5, various testing frameworks were considered for implementing unit tests in QPLEX. Pytest was chosen due to its simplicity (i.e., it does not require a lot of code overhead to implement tests), robustness and up-to-date documentation. Moreover, Pytest allows the use of the built-in Python testing module Unittest, taking advantage of its mocking features, which were essential when testing QPLEX modules such as the Solvers module.

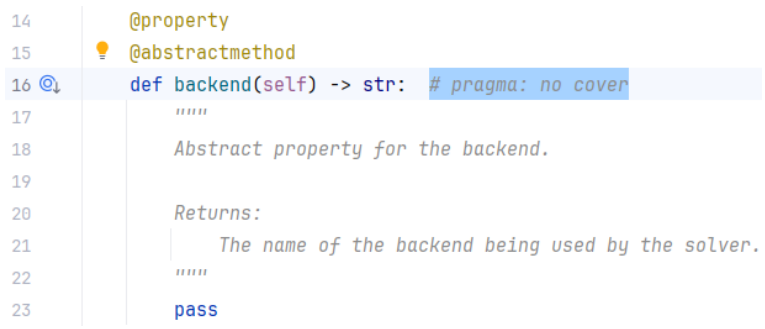

Figure 6.28|: An example of how to exclude a method from the test coverage metric through comments

Test Organization Approach

In order to maintain a clear directory structure, the unit tests were implemented inside a dedicated `tests` directory at the same level as the QPLEX source code directory. The `tests` directory mirrors the structure found in our package's source folder for enhanced discoverability and maintenance.

The test structure, mentioned earlier in the previous section and depicted as a generic example in Figure 6.27 follows a one-to-one approach for classes, methods and functions. In summary, all classes in the QPLEX library that contain one or more methods and all functions that are part of some QPLEX module must have their own test class, with dedicated tests for each method and function deemed worth testing. Methods and functions that act as interfaces for calling external libraries or modules do not need to be tested since they do not add functionality beyond a function call. In order to maintain the desired test coverage of 80%, methods and functions that do not need to be tested must be annotated with the "`# pragma: no cover`" comment on the method or function signature. This also applies to abtract classes that do not include an implementation for their methods as shown in Figure 6.28. This figure shows a method of the abstract `Solver` class.

Test Design Principles

As evidenced in the example shown in Figure 6.27, tests were designed following the Arrange-Act-Assert pattern, with a focus on specific functionalities (unit tests) while minimizing dependencies. For classes with complex initialization requirements, Pytest fixtures were utilized to reduce code duplication across test cases. Moreover, mocking was used to avoid calling external modules. Figure 6.29 illustrates a representative test case for the `IBMQSolver` class, demonstrating how the parse_input method is verified.

Another example in Figure 6.30 shows a test for the Algorithm Factory's ability to generate appropriate algorithm instances based on configuration parameters.

Integration with Development Workflow

The design and implementation of unit tests was integrated into the development workflow as a requirement for successful code merges into the development branch of the library's repository. This requirement is verified automatically by a GitHub Action described

```python
def test_parse_input(self):
    """Test parse_input converts OpenQASM string to QuantumCircuit."""
    test_circuit = """
    qubit[2] q;
    h q[0];
    cx q[0], q[1];
    measure q;
    """

    with patch('qiskit.qasm3.loads') as mock_loads:
        mock_qc = MagicMock(spec=QuantumCircuit)
        mock_loads.return_value = mock_qc

        result = self.solver.parse_input(test_circuit)

        # Check that qasm3.loads was called with the right string
        expected_circuit = """
    OPENQASM 3.0;
    include "stdgates.inc";
    """ + test_circuit
        mock_loads.assert_called_once_with(expected_circuit)
        assert result == mock_qc
```

Figure 6.29|: A code snippet showing the `test_parse_input` unit test as part of the `TestIBMQSolver` test class

```python
@patch('qplex.utils.model_utils.get_model_constraint_info')
@patch('qplex.algorithms.QAOA')
@patch('qplex.algorithms.mixers.mixer_factory.MixerFactory.get_mixer')
def test_get_algorithm_qao_ansatz(self, mock_get_mixer, mock_qaoa_class,
                                  mock_get_model_constraint_info,
                                  mock_model):
    """
    Test that get_algorithm returns a QAOA instance with custom mixer for
    QAO_ANSATZ algorithm type
    """
    mock_constraint_info = Mock(spec=ConstraintInfo)
    mock_get_model_constraint_info.return_value = mock_constraint_info

    mock_mixer = Mock()
    mock_get_mixer.return_value = mock_mixer

    mock_qaoa = Mock()
    mock_qaoa_class.return_value = mock_qaoa

    config = AlgorithmConfig(
        algorithm=AlgorithmType.QAO_ANSATZ,
        penalty=1.0,
        seed=42,
        p=2
    )

    result = AlgorithmFactory.get_algorithm(mock_model, config)

    mock_get_model_constraint_info.assert_called_once_with(mock_model)

    mock_get_mixer.assert_called_once_with(mock_constraint_info)

    mock_qaoa_class.assert_called_once_with(
        mock_model, p=2, seed=42, penalty=1.0, mixer=mock_mixer
    )

    assert result is mock_qaoa
```

Figure 6.30|: A code snippet showing the `test_get_algorithm_qao_ansatz` unit test as part of the `TestAlgorithmFactory` test class

```
1  [pytest]
2  testpaths = tests
3  python_files = test_*.py
4  python_classes = Test*
5  python_functions = test_*
6  addopts = -v --cov=qplex --cov-report=term-missing --cov-branch --cov-fail-under=80
```

Figure 6.31|: The code found in the `pytest.ini` file. This file helps configure the testing process when using Pytest, including test paths and desired test coverage.

in Section 6.4.3 and to pass this check the proposed changes must include passing tests for all new methods and functions and they must maintain the required 80% test coverage.

NFR-5: Test Coverage Metric

For our testing framework, branch coverage was selected as the primary test coverage metric because it provides a more comprehensive assessment of execution paths compared to line coverage. This approach ensures that conditional logic branches throughout the codebase are adequately tested. Being a wrapper for several quantum providers and algorithm implementations, QPLEX contains various logical paths that can be taken depending on the execution parameters chosen by the user. Thus, evaluating the coverage of tests based on the number of possible paths constitutes a solid choice.

A minimum threshold of 80% branch coverage was established as a balance between comprehensive testing and development efficiency. While higher coverage percentages are theoretically desirable, they often yield diminishing returns beyond a threshold of 80-90% [90].

The coverage requirement was implemented using the `pytest-cov`[16] plugin, which integrates seamlessly with our existing Pytest framework. Figure 6.31 shows the configuration in pytest.ini that enforces this coverage threshold on line 6.

When tests are executed, Pytest generates a coverage report that details covered and missed branches, as shown in Figure 6.32.

6.4.3 Continuous Integration and Continuous Deployment

NFR-6: GitHub Actions Workflows

As mentioned previously, GitHub Actions was selected as the CI/CD platform for QPLEX due to its seamless integration with the GitHub repository and minimal configuration requirements. The CI/CD implementation follows a multi-staged approach, separating testing, documentation generation, and deployment into distinct workflows. These workflows are configured in YAML[17] files stored in the project repository.

The test workflow, shown in Figure 6.33, executes on every push and pull request to the development branch. It runs the test suite on a Python environment with version 3.10

[16] https://pypi.org/project/pytest-cov/
[17] https://yaml.org/

Terminal Local

```
Name                                                Stmts   Miss Branch BrPart  Cover   Missing
-----------------------------------------------------------------------------------------------
qplex/__init__.py                                       4      0      0      0   100%
qplex/algorithms/__init__.py                            3      0      0      0   100%
qplex/algorithms/base_algorithm.py                     16      0      2      0   100%
qplex/algorithms/mixers/__init__.py                     7      0      0      0   100%
qplex/algorithms/mixers/cardinality_mixer.py            9      0      4      0   100%
qplex/algorithms/mixers/composite_mixer.py             10      0      2      0   100%
qplex/algorithms/mixers/inequality_mixer.py             8      0      2      0   100%
qplex/algorithms/mixers/mixer_factory.py               51      0     20      2    97%   84->87, 107->105
qplex/algorithms/mixers/partition_mixer.py              8      0      2      0   100%
qplex/algorithms/mixers/quantum_mixer.py                3      0      0      0   100%
qplex/algorithms/mixers/standard_mixer.py               5      0      0      0   100%
qplex/algorithms/qaoa.py                               45      0     14      1    98%   110->108
qplex/algorithms/vqe.py                                38      0      4      0   100%
qplex/commons/__init__.py                               2      0      0      0   100%
qplex/commons/algorithm_factory.py                     33      0      6      0   100%
qplex/commons/optimization_callback.py                 17      0      0      0   100%
qplex/commons/solver_factory.py                        51      1     10      1    97%   134
qplex/model/__init__.py                                 3      0      0      0   100%
qplex/model/constants.py                                9      0      0      0   100%
qplex/model/execution_config.py                        32      0      4      0   100%
qplex/model/qmodel.py                                  75      2     12      1    97%   160-162
qplex/solvers/__init__.py                               3      0      0      0   100%
qplex/solvers/base_solver.py                            3      0      0      0   100%
qplex/solvers/braket_solver.py                         28      0      2      0   100%
qplex/solvers/dwave_solver.py                          93      3     34      3    94%   165->168, 191-192, 238
qplex/solvers/ibmq_solver.py                           55      0     10      0   100%
qplex/utils/__init__.py                                 1      0      0      0   100%
qplex/utils/circuit_utils.py                            7      0      0      0   100%
qplex/utils/model_utils.py                             36      0     10      0   100%
qplex/utils/workflow_utils.py                          24      0     12      1    97%   44->48
qplex/workflows/__init__.py                             3      0      0      0   100%
qplex/workflows/ggae_workflow.py                       23      0      0      0   100%
qplex/workflows/ibm_session_workflow.py                32      0      0      0   100%
-----------------------------------------------------------------------------------------------
TOTAL                                                 737      6    150      9    98%

Required test coverage of 80% reached. Total coverage: 98.08%
```

Figure 6.32|: The output of running all tests using Pytest

```yaml
name: Tests

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

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3
    - name: Set up Python 3.10
      uses: actions/setup-python@v4
      with:
        python-version: '3.10'
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt
        pip install pytest pytest-cov
    - name: Run tests with coverage
      run: |
        pytest
```

Figure 6.33|: The contents of the `tests.yaml` file

and enforces the 80% branch coverage requirement established in NFR-5. If one ore more tests fail or if the required coverage is not met, the workflow is tagged as failed, inhibiting the merge.

Similarly, the documentation workflow in Figure 6.34 regenerates the API documentation using Sphinx whenever changes are made to docstrings or documentation files. This API specification is coded into HTML pages that are automatically deployed on the convenient GitHub Pages service, also integrated into the GitHub platform. This deployment is also performed automatically by the same workflow.

The creation and deployment of these CI/CD workflows help maintain code quality standards throughout the development process.

NFR-7: Deployment to PyPI

For the final non-functional requirement, the process of deploying QPLEX to PyPi required the creation of one of the two standardized files for publishing on the index: a Python file commonly named `setup.py` and the more modern `pyproject.toml` file. Both options are viable, but the latter has become the new standard so it was chosen for QPLEX given that the librar has not been published before. The purpose of this configuration file is

```yaml
name: Build and Deploy Docs

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

  workflow_dispatch:

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.10'
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
          pip install sphinx
      - name: Generate RST files
        run: |
          sphinx-apidoc -f -o docs qplex
      - name: Build Docs
        run: |
          cd docs
          make html
      - name: Deploy to GitHub Pages
        uses: peaceiris/actions-gh-pages@v3
        with
          github_token: ${{ secrets.GITHUB_TOKEN }}
          publish_dir: ./docs/_build/html
```

Figure 6.34|: The contents of the `deploy-docs.yaml` file

```toml
[build-system]
requires = ["setuptools>=45", "wheel"]
build-backend = "setuptools.build_meta"

[tool.setuptools.packages.find]
include = ["qplex"]

[project]
name = "uvic-qplex"
version = "0.1.1"
description = "an open-source Python library that enables developers to implement combinatorial
optimization models and execute them seamlessly on multiple classical and quantum devices using
different quantum algorithms."
readme = "README.md"
authors = [
    { name = "Juan Giraldo", email = "juanfer021299@gmail.com" },
    { name = "Jose Ossorio", email = "joseossorio99@hotmail.com" }
]
license = { text = "GPL-3.0" }
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: GNU General Public License v3 (GPLv3)",
    "Operating System :: OS Independent",
]
requires-python = ">=3.10.0"
dependencies = [
    "docplex~=2.25",
    "qiskit~=1.0",
    "qiskit_aer",
    "qiskit_ibm_runtime",
    "qiskit_optimization",
    "qiskit_qasm3_import",
    "dwave-system~=1.18",
    "amazon-braket-sdk~=1.80",
    "scipy~=1.10.1",
    "dimod~=0.12.4",
    "numpy~=1.23.5",
    "networkx",
]


[project.urls]
Homepage = "https://github.com/JuanGiraldo0212/QPLEX"
Documentation = "https://juangiraldo0212.github.io/QPLEX/"
Repository = "https://github.com/JuanGiraldo0212/QPLEX"
Issues = "https://github.com/JuanGiraldo0212/QPLEX/issues"
```

Figure 6.35|: The contents of the `pyproject.toml` configuration file

to define the package's metadata, dependencies and installable components in Tom's Obvious Minimal Language (TOML)[18] format, a configuration file format meant to be readable for humans. A snapshot of the contents of this file is shown in Figure 6.35, where various details about the library can be observed.

With this configuration set in place, a dedicated deployment workflow implemented as a GitHub Action, triggered on the merge of a pull request into the *main* repository branch, handles the automated packaging and uploading to PyPI as illustrated in Figure 6.36.

This workflow builds both source and wheel distributions before uploading them to PyPI using secure credentials stored as GitHub Secrets.

Users can now install QPLEX directly using pip: `pip install uvic-qplex`. This PyPI

[18] https://toml.io/en/

```yaml
name: Build and Publish to PyPI

on:
  push:
    branches:
      - main

jobs:
  build-and-publish:
    name: Build and publish to PyPI
    runs-on: ubuntu-latest

    steps:
    - name: Check out repository
      uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.10'

    - name: Install build dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt
        pip install build twine

    - name: Build package
      run: python -m build

    - name: Check package with twine
      run: twine check dist/*

    - name: Publish to PyPI
      uses: pypa/gh-action-pypi-publish@release/v1
      with:
        password: ${{ secrets.PYPI_API_TOKEN }}
```

Figure 6.36|: The contents of the `publish.yaml` GitHub Action configuration file

deployment significantly reduces the barrier to entry for new users, eliminating the need to clone the repository and build the package manually.

6.5 Chapter Summary

This chapter detailed the design and implementation of enhancements to the QPLEX library based on the requirements identified in earlier Chapters 4 and 5. It began with general design changes aimed at reducing the library's complexity, including the introduction of an `ExecutionConfig` class to encapsulate execution parameters and the development of the `OptimizationCallback` class to enable custom progress monitoring. The chapter then explored the creation of a new Workflows module to better support specialized execution patterns like Qiskit Runtime Sessions. The implementation of functional requirements is organized by module, showing how features like Qiskit Runtime Sessions support, compatibility with Qiskit 1.0.0, and quantum solver flexibility were incorporated into the codebase. The chapter also covered the implementation of non-functional requirements through comprehensive documentation, a testing framework with a high coverage metrics, and CI/CD pipelines that streamline development and deployment. These enhancements collectively strengthen QPLEX's capabilities for solving combinatorial optimization problems across multiple quantum platforms.

The implementations described in this chapter resulted in the addition of multiple execution options to the QPLEX framework. These options, which can be selected by users when calling the `solve` method, all have a default value, similar to the original set of options described in [15]. Table 6.1 completes the original set by including the newly added fields and their respective default values.

[19] When the D-Wave API token is available.
[20] The default workflow refers to the GGAE workflow.

Table 6.1: List of execution options available for the `solve` method

Option	**Purpose**	**Default Value**
provider	The quantum hardware provider	"d-wave" [19]
workflow	The QPLEX workflow to be used for the execution	"default" [20]
verbose	Determines whether the cost function will print the current cost	`False`
backend	The specific quantum device	Calculated
provider_options	Provider-specific options to be passed down to the chosen provider for the execution	{}
algorithm	The quantum algorithm to use	"qaoa"
ansatz	The ansatz circuit for VQE	Layered Ansatz
p	The p value for the QAOA algorithm	2
mixer	The `QuantumMixer` instance to be used during the QAOA algorithm execution	`None`
layers	The number of layers for the VQE algorithm	2
optimizer	The classical optimizer	"COBYLA"
callback	The callback function for the classical optimizer	`OptimizationCallback`
tolerance	The tolerance value for the optimizer	$1e-10$
max_iter	The maximum number of optimizer iterations	1000
penalty	The penalty constant used for the QUBO	Calculated
shots	The total number of shots	1024
seed	The execution random seed	1

Chapter 7

Validation and Discussion

7.1 Introduction

This chapter presents the framework for validating the functional requirements implemented in this thesis. Following traditional software engineering practices, the goal of the validation stage is to ensure that the implemented features correctly address the requirements elicited during the analysis phase.

Our validation approach follows a systematic methodology for each functional requirement, consisting of:

1. **Implementation Verification:** Validating that the implementation correctly fulfills the requirement

2. **Usability evaluation:** Evaluating the requirement's accessibility in terms of ease of use from the user's perspective

Moreover, a performance analysis was included for a subset of the requirements.

This chapter begins with an individual validation of each implemented functional requirement, concluding with a discussion of the findings and implications for the QPLEX library's effectiveness in enabling quantum-based CO.

7.2 Validation of Individual Functional Requirements

7.2.1 FR-1: Support for Qiskit Runtime Sessions

Qiskit Runtime Sessions provide prioritized access to quantum hardware, reducing queue waiting times between the iterative steps of variational quantum algorithms. As Runtime Sessions are a premium feature, validation focused on verifying the correct implementation structure and integration with the Qiskit API.

```python
# Excerpt from test_ibm_session_workflow.py
@patch('qplex.commons.algorithm_factory.AlgorithmFactory.get_algorithm')
@patch('scipy.optimize.minimize')
@patch('qiskit.transpiler.preset_passmanagers.generate_preset_pass_manager')
@patch('qplex.workflows.ibm_session_workflow.compute_counts')
@patch('qplex.utils.workflow_utils.calculate_energy')
@patch('qiskit_ibm_runtime.SamplerV2')
@patch('qiskit_ibm_runtime.Session')
def test_ibm_session_workflow(self, mock_session_class, mock_sampler_class,
                              mock_calculate_energy,
                              mock_compute_counts, mock_generate_pm,
                              mock_minimize,
                              mock_get_algorithm, mock_model,
                              mock_ibmq_solver, mock_options):
    # Test implementation...
```

Figure 7.1|: Excerpt from test_ibm_session_workflow.py

Implementation Verification

The implementation of the IBM Session Workflow was validated through comprehensive unit tests that mock the interactions with the Qiskit Runtime service. The test suite,[1] as shown in Figure 7.1, verifies that:

- The workflow correctly instantiates a Qiskit Runtime Session with appropriate parameters
- The Sampler primitive is properly initialized within the session context
- The optimization loop correctly binds parameters to the quantum circuit
- Results from the quantum execution are properly processed and returned

The implementation structure was validated against IBM's official documentation for Qiskit Runtime Sessions.[2] Key validation points included:

- **Session Creation:** The implementation correctly creates a Session object using the QiskitRuntimeService and the selected backend
- **Session Context Management:** The implementation uses Python's context management protocol with the Session object.
- **Primitive Usage:** Within the session context, the code correctly instantiates and uses the Sampler primitive.

[1] The source code for the `IBMQSolver`, where the test suite resides, is available at https://github.com/JuanGiraldo0212/QPLEX/blob/main/tests/solvers/test_ibmq_solver.py

[2] https://docs.quantum.ibm.com/api/qiskit-ibm-runtime/session

- **Parameter Binding:** Circuit parameters are correctly bound for each iteration of the optimization.

Usability Evaluation

The Qiskit Runtime Session workflow can be easily accessed by specifying it as the desired workflow to be used during execution as part of the arguments of the solve method call. This ensures that users can access the feature through a simple interface and provides a foundation for future workflows to be added to the QPLEX library.

7.2.2 FR-2: Support for Qiskit 1.0.0

Implementation Verification

This requirement was validated implicitly through the successful execution of tests for FR-1 (Qiskit Runtime Sessions) and FR-3 (Qiskit's Sampler Primitive) with Qiskit 1.0.0 installed as the active version in the testing environment. All IBMQ-related tests completed successfully, confirming the compatibility of the updated `IBMQSolver` implementation with the new version of the Qiskit library.

Usability Evaluation

The migration to Qiskit 1.0.0 is completely transparent to end users, requiring no changes to existing code that uses the QPLEX library. This backward compatibility ensures a smooth transition for users.

7.2.3 FR-3: Support for Qiskit's Sampler Primitive

Implementation Verification

The implementation of the Sampler primitive was validated through unit tests that verify:

- The `IBMQSolver` correctly instantiates the Sampler primitive
- Circuit execution requests are properly routed through the Sampler
- The results returned by the Sampler are correctly parsed into the expected format

To validate the functional behavior, a simple MaxCut problem was solved using both the previous execution mechanism and the new Sampler primitive on a local simulator. The results demonstrated a similar average objective value between the two approaches using the QAOA algorithm with the p parameter set to 4, compared to the classical result using the CPLEX solver, as seen in Figure 7.2.

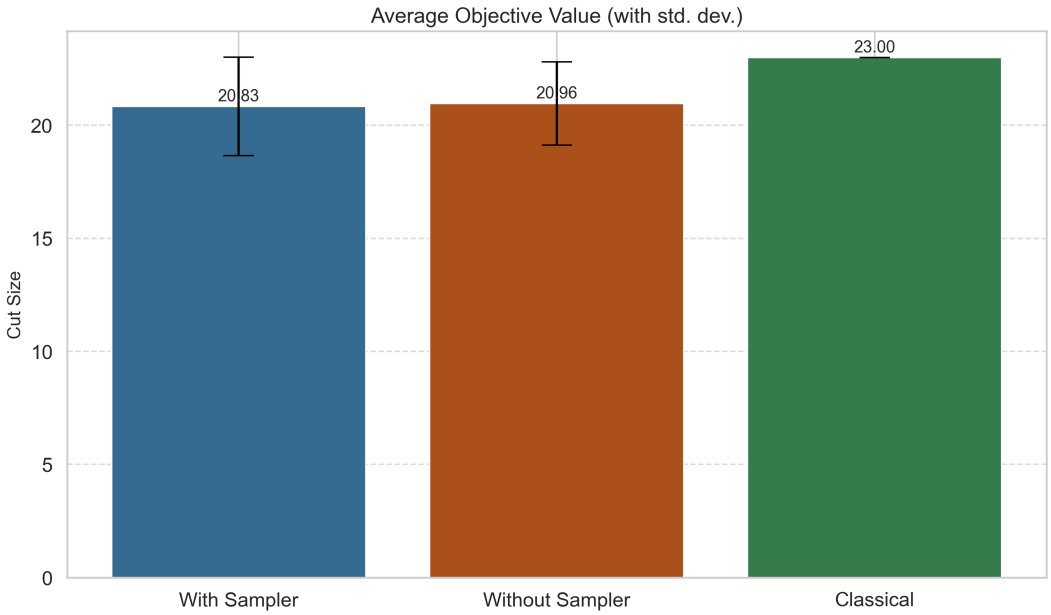

Figure 7.2|: Average objective values for the MaxCut problem with the Sampler primitive, without the Sampler primitive and using the classical CPLEX solver

Usability Evaluation

The Sampler primitive is now the only method used to run Qiskit code on simulators and IBM quantum machines. The change is transparent to users, requiring no adjustments in existing code or practices.

7.2.4 FR-4: Support for Automatic Circuit Optimization

Implementation Verification

To validate this requirement, a knapsack problem formulation was used to create a `QModel` with the following parameters:

$$\begin{aligned} values &= [10, 5, 18, 12, 15, 1, 2, 8, 4, 5, 6, 7] \\ weights &= [4, 2, 5, 4, 5, 1, 3, 5, 4, 1, 2, 3] \\ capacity &= 15 \end{aligned}$$

The model was mapped to a QAOA circuit with p=8 and transpiled multiple times, each with a different circuit optimization level (0-3), targeting the IBM Sherbrooke processor, which has 127 qubits. The objective was to verify that:

- The optimization level parameter was correctly passed to the `IBMQSolver` instance
- Each optimization level produced different circuit configurations

Analysis of the transpiled circuits confirmed significant differences between optimization levels, as shown in Table 7.1. Notably, while all optimization levels replaced CNOT gates with hardware-native gates, higher optimization levels progressively reduced the total gate count. Interestingly, optimization levels 2 and 3 made different trade-offs between gate count and circuit depth compared to level 1.

Table 7.1: Circuit metrics after transpilation with different optimization levels

Configuration	Gate Count	CNOT Count	Circuit Depth
Original Circuit	3,152	1,920	441
Optimization Level 0	61,212	0	7,780
Optimization Level 1	24,602	0	3,250
Optimization Level 2	24,155	0	3,516
Optimization Level 3	21,824	0	4,278

These results confirmed that the optimization level parameter was correctly implemented and that different optimization levels produce meaningfully different circuit transpilations.

Usability Evaluation

The optimization level can be specified as a parameter in the solve method call, with a default value of 1 (light optimization) if not specified. This provides users with a simple interface to control the trade-off between transpilation time and circuit efficiency based on their specific needs.

7.2.5 FR-7: Support for Provider-Specific Execution Options with Amazon Braket

Implementation Verification

The implementation of this requirement was validated through unit tests[3] that verify:

1. Provider-specific options are correctly passed from the QModel's solve method to the BraketSolver
2. These options are properly integrated into the `device_parameters` dictionary when making API calls

Usability Evaluation

This feature can be accessed as an execution configuration parameter for the solve method, similar to other enhancements presented in this work. This ensures consistency and maintains a simple interface for customizing and orchestrating execution.

7.2.6 FR-8: Support for D-Wave Quantum Solvers

Implementation Verification

Due to access limitations to the D-Wave Leap platform, validation was performed using mock objects that mimic the behavior of D-Wave's quantum solvers. This validation is part of the unit tests for the `DWaveSolver` class:[4]

1. Mock DWaveSampler and EmbeddingComposite classes were created to capture and validate parameter passing
2. Test cases verified that:
    - The solver correctly distinguishes between hybrid and quantum solver selections
    - Problem models are properly formatted for submission to quantum solvers
    - The topology parameter is correctly used when initializing the DWaveSampler

[3] The source code is available at https://github.com/JuanGiraldo0212/QPLEX/blob/main/tests/solvers/test_braket_solver.py

[4] The source code for the tests is available at https://github.com/JuanGiraldo0212/QPLEX/blob/main/tests/solvers/test_dwave_solver.py

Usability Evaluation

Users can now pass an additional parameter to the solve method when using D-Wave as the provider: the backend parameter. Through the use of this new parameter, they can select between hybrid and quantum solvers, ensuring that the interface for accessing this new feature requires minimal overhead for practitioners.

7.2.7 FR-9: Support for Custom Minor Embedding for D-Wave Quantum Solvers

Implementation Verification

Similar to FR-8, custom embedding support was validated using mock objects that verify:

1. Custom embeddings provided as parameters are correctly passed to the `FixedEmbeddingComposite`
2. When no custom embedding is provided, the system correctly defaults to `AutoEmbeddingComposite`

Usability Evaluation

Users should refer to the most recent Ocean SDK documentation for the expected embedding format. Provided that they use a format accepted by the `FixedEmbeddingComposite` class, practitioners can simply provide the embedding dictionary as an extra execution option to the solve method. This adds flexibility to the execution while maintaining a simple interface that aligns with the established QPLEX architecture.

7.2.8 FR-11: Support for QAOAnsatz

Implementation Verification

To validate the implementation of the QAOAnsatz algorithm, we solved a constrained optimization problem using both standard QAOA with the default Rx mixer and QAOAnsatz with an XY mixer, executed on a local simulator. The problem was an instance of the portfolio optimization problem with cardinality constraints limiting the number of stocks in the portfolio. Our verification confirmed that the framework successfully detects constraint types and instantiates the appropriate built-in mixer implementations.

Performance Analysis

Testing revealed the following insights:

- The standard Rx mixer provided good results across most executions, reflecting its ability to explore the full solution space.

- The XY mixer produced comparable results but exhibited certain limitations, likely due to its more restrictive exploration of the feasible solution space.

Table 7.2 presents results from 100 executions of each algorithm alongside baseline results from a CPLEX Solver. The execution configuration used was:

- p: 4
- Shots: 8,192
- Max iterations: 500

As shown in the table, both quantum approaches achieved similar average objective values and both successfully reached the maximum objective value found by the classical CPLEX solver. The difference in average execution time between QAOA and QAOAnsatz can be primarily attributed to the additional computational overhead required by QAOAnsatz to identify constraint types in the problem model before circuit creation in each experimental run.

Performance Metrics (valid solutions only)			
Algorithm	**Average objective value**	**Maximum objective value**	**Average execution time**
CPLEX Solver	3100.00	3100.00	0.01 seconds
QAOAnsatz	2082.86	3100.00	74.32 seconds
QAOA	2173.02	3100.00	43.71 seconds

Table 7.2: Comparison of QAOA implementations performance metrics

The observed variability in performance highlights the challenges in practical implementations of generic constraint-preserving mixers, which remains an active area of research in QC.

Usability Evaluation

The implementation provides users with two options for mixer selection:

- Automatic selection based on problem constraints (default)
- Explicit mixer specification, allowing users to override the automatic selection

This hybrid approach balances convenience with flexibility, accommodating both practitioners seeking a built-in solution and those requiring precise control over the quantum algorithm's behavior. The ability to override automatic selection is particularly valuable for complex constraint types where generic implementations may not capture the full problem structure.

Nevertheless, the modular architecture of our implementation ensures that as better mixer alternatives become available for given constraint types they can be integrated into our library.

7.3 Discussion

The validation demonstrates that the implemented features correctly address the elicited functional requirements, extending the QPLEX library's capabilities for solving CO problems using quantum resources, even within the constraints of limited hardware access.

7.3.1 Limitations of the Validation Approach

While the validation approach was effective given the constraints, some limitations were evidenced:

1. **Hardware-specific Behavior:** Without access to actual quantum hardware, hardware-specific behaviors and performance characteristics could not be directly validated.
2. **Edge Cases:** Some edge cases that might only manifest on actual hardware may not have been captured by the validation process.

7.3.2 Implications for QPLEX Users

Despite these limitations, the validation confirms that QPLEX users can benefit from the newly implemented features:

1. **Enhanced Provider Support and Greater Flexibility:** Users now have access to a wider range of execution options available on the supported quantum resources through a consistent interface, allowing for greater control over execution.
2. **Improved Efficiency:** The support for Qiskit Runtime Sessions provide theoretical performance improvements for variational algorithms.

7.4 Conclusion

This chapter presented a comprehensive validation of the functional requirements implemented in the QPLEX library in this thesis. The validation results demonstrate that the enhancements have successfully addressed the identified functional requirements, extending the QPLEX library's capabilities for solving CO problems using quantum resources.

The implemented features provide users with greater flexibility in accessing different quantum resources, more efficient execution workflows, and enhanced algorithms for handling constrained optimization problems. These improvements collectively advance the QPLEX library toward its goal of becoming a versatile platform-agnostic tool for quantum-based CO.

As quantum hardware continues to improve and quantum algorithms evolve, the enhanced QPLEX library and its extensible architecture provide a solid foundation for exploring and leveraging quantum resources for CO problems.

Chapter 8

Conclusions

This thesis has followed a structured software engineering approach to validate and extend the QPLEX library, a platform-agnostic solution for solving combinatorial optimization problems using both quantum and classical resources. The work builds upon the foundation established by the original authors, who successfully took the first step toward developing a platform-agnostic library for solving CO problems using quantum and classical resources, creating an abstraction layer for practitioners to be able to introduce QC into their optimization workflow. However, in the rapidly evolving landscape of quantum computing and software technologies, continuous improvement and adaptation are essential to maintain relevance and usefulness.

This thesis aimed to take the next step toward an extensible quantum platform-agnostic CO library. It emphasized high standards for extensibility, a crucial quality attribute in open-source quantum software, and enhanced the existing QPLEX library by providing greater control over the execution and richer experimentation.

8.1 Summary of the Software Engineering Process

This thesis adopted a systematic software engineering lifecycle approach, beginning with a thorough analysis phase that examined both the completeness and extensibility of the QPLEX library. The analysis yielded two key artifacts: a set of functional requirements addressing gaps in the library's capabilities, and a set of non-functional requirements focusing on improving its extensibility as an open-source project.

The design phase translated these requirements into concrete architectural decisions, including the introduction of new modules such as Workflows, the creation of specialized classes like `ExecutionConfig` and `OptimizationCallback`, and the implementation of design patterns like the Factory Method. These design decisions were documented through various artifacts, including UML class diagrams, sequence diagrams, and updated system architecture diagrams, providing a clear blueprint for the implementation phase.

During implementation, the design was realized through code, resulting in enhancements to the QPLEX library. The implementation extended the library's functionality with support for Qiskit Runtime Sessions, QAOAnsatz framework, D-Wave quantum solvers with custom minor embedding capabilities, and many other features that collectively enhanced

the library's usefulness for CO practitioners. Simultaneously, improvements to documentation, testing infrastructure, and CI/CD pipelines strengthened the library's extensibility, setting a solid foundation for future open-source contributions.

8.2 Contributions

The contributions of this thesis directly address the research questions posed at the beginning of this work:

RQ1: What are the key components in a workflow for solving combinatorial optimization problems using classical and quantum approaches?

Contribution 1: This thesis provided a review and synthesis of the CO process, resulting in a general high-level workflow that organizes the process into four key stages: Problem Analysis, Problem Abstraction and Modeling, Algorithm Selection, and Solution and Iteration. This workflow served as a conceptual framework for understanding the CO process and guided the extension of QPLEX.

RQ2: How can these workflow insights be applied to analyze and enhance the QPLEX library's completeness and extensibility, ensuring it effectively supports a wide range of quantum providers and facilitates future developments in hybrid quantum-classical optimization?

Contribution 2: This thesis mapped QPLEX's capabilities to the general CO workflow components introduced in Chapter 3, revealing specific gaps in completeness and extensibility. The workflow analysis highlighted that the library's strengths resided primarily in the Problem Modeling stage, while lacking critical capabilities in the Algorithm Selection and Solution and Iteration stages. This guided the elicitation of requirements: functional requirements targeted enhancing algorithm diversity, execution flexibility, and performance optimization. Non-functional requirements addressed extensibility barriers through comprehensive documentation, modular design patterns, automated testing, and CI/CD pipeline implementation. This workflow-driven approach ensured that enhancements were prioritized to address the most critical gaps in QPLEX's support for the complete CO process, rather than implementing features in isolation.

RQ3: To what extent do the newly implemented features in QPLEX improve its functionality in solving combinatorial optimization problems using quantum resources and enhance its extensibility, facilitating future development?

Contribution 3: The implementation and validation of new features in QPLEX demonstrated significant enhancements to the library's functionality, performance, and extensibility. On the functional side, several key improvements were made: (1) Qiskit Runtime Sessions improved execution efficiency by reducing queue wait times for iterative algorithms; (2) QAOAnsatz with flexible mixer operators expanded the library's problem-solving capabilities for constrained optimization problems; and (3) support for D-Wave's quantum

solvers with custom minor embedding provided users with more advanced control over the quantum annealing process.

Equally important were the extensibility enhancements implemented through software engineering practices: comprehensive documentation following established standards, extensive unit tests achieving 80% branch coverage, automated CI/CD pipelines for continuous validation, and deployment to PyPI for simplified accessibility. These improvements establish a solid foundation that enables future contributors to extend the library more efficiently, while maintaining code quality and compatibility.

Together, these functional and non-functional enhancements have transformed QPLEX into a more versatile, accessible, and maintainable platform-agnostic combinatorial optimization library.

8.3 Future Work

While this thesis made substantial progress in advancing the QPLEX library, several avenues for future work remain:

Implementation of Remaining Requirements: Some functional requirements identified in the analysis phase were not implemented within the scope of this thesis. These include FR-4 (Support for Qiskit's Estimator primitive), FR-6 (Support for Amazon Braket Hybrid Jobs), and FR-10 (Support for user-defined ansatz in VQE). An evaluation of these requirements and their subsequent implementation in the future would further enhance QPLEX's completeness and flexibility.

Integration with Generative AI: The rise of generative AI presents opportunities to simplify problem modeling in QPLEX. Future work could explore integrating AI assistants that help users translate their real-world problems into mathematical formulations compatible with QPLEX's modeling interface. This could significantly lower the barrier to entry for practitioners without extensive mathematical optimization knowledge.

Experimentation Module: A dedicated module for structured experimentation would be valuable for researchers and practitioners exploring the performance of different algorithms and hyperparameters. This module could automate the process of executing multiple runs with varying configurations, collecting performance metrics, and analyzing results to identify optimal settings for specific problem classes.

Visualization Library Integration: Enhancing QPLEX with integrated visualization capabilities would improve the interpretability of optimization results and algorithm performance. Visualizations could include solution quality over iterations, energy landscapes, and comparative performance across different algorithms or quantum providers.

Extended Provider Support: While QPLEX currently supports three major quantum providers (IBM Quantum, D-Wave, and Amazon Braket), expanding support to include emerging providers would increase the library's versatility.

Bibliography

[1] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, 10th. Cambridge University Press, 2011.

[2] J. Preskill, "Quantum computing in the NISQ era and beyond", *Quantum*, vol. 2, p. 79, 2018.

[3] K. Liegener, O. Morsch, and G. Pupillo, "Solving quantum chemistry problems on quantum computers", *Physics Today*, vol. 77, no. 9, pp. 34–42, 2024.

[4] D. Maslov, Y. Nam, and J. Kim, "An outlook for quantum computing [point of view]", *Proceedings of the IEEE*, vol. 107, no. 1, pp. 5–10, 2018.

[5] A. P. et al., "A variational eigenvalue solver on a photonic quantum processor", *Nature Communications*, vol. 5, no. 1, 2014.

[6] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm", *arXiv preprint arXiv:1411.4028*, 2014.

[7] A. Callison and N. Chancellor, "Hybrid quantum-classical algorithms in the noisy intermediate-scale quantum era and beyond", *Physical Review A*, vol. 106, no. 1, 2022.

[8] L. Mueck *et al.*, "Quantum software", *Nature*, vol. 549, no. 7671, pp. 171–171, 2017.

[9] M. Piattini, M. Serrano, R. Perez-Castillo, G. Petersen, and J. L. Hevia, "Toward a Quantum Software Engineering", *IT Professional*, vol. 23, no. 1, pp. 62–66, 2021.

[10] M. Piattini, G. Peterssen Nodarse, R. Pérez-Castillo, *et al.*, "The Talavera Manifesto for Quantum Software Engineering and Programming", 2020. [Online]. Available: https://www.researchgate.net/profile/Ricardo-Perez-Castillo/publication/339780973_The_Talavera_Manifesto_for_Quantum_Software_Engineering_and_Programming/links/5e64a47e4585153fb3ca221d/The-Talavera-Manifesto-for-Quantum-Software-Engineering-and-Programming.pdf.

[11] M. Fingerhuth, T. Babej, and P. Wittek, "Open source software in quantum computing", *PloS one*, vol. 13, no. 12, 2018.

[12] I. C. Education, *SDK vs. API: Whats the Difference?*, 2021. [Online]. Available: https://www.ibm.com/blog/sdk-vs-api/www.ibm.com/blog/sdk-vs-api (visited on 01/23/2024).

[13] A. Stefik and S. Siebert, "An Empirical Investigation into Programming Language Syntax", *ACM Transactions on Computing Education*, vol. 13, no. 4, pp. 1–40, 2013.

[14] *TKET*. [Online]. Available: https://www.quantinuum.com/developers/tket (visited on 01/23/2024).

[15] J. F. Botello Giraldo, "QPLEX: Towards the Integration of Platform Agnostic Quantum Computation into Combinatorial Optimization Software", MSc thesis, University of Victoria, 2024.

[16] T. Lubinski, C. Coffrin, C. McGeoch, *et al.*, "Optimization Applications as Quantum Performance Benchmarks", *ACM Transactions on Quantum Computing*, vol. 5, no. 3, 18:1–18:44, 2024.

[17] F. Glover, G. Kochenberger, and Y. Du, *A Tutorial on Formulating and Using QUBO Models*, 2019. [Online]. Available: https://arxiv.org/pdf/1811.11538.

[18] J. Giraldo, J. Ossorio, N. M. Villegas, G. Tamura, and U. Stege, "QPLEX: Realizing the integration of quantum computing into combinatorial optimization software", in *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 1, 2023, pp. 1044–1049.

[19] *ISO 25010 Software and Data Quality*. [Online]. Available: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 (visited on 03/24/2025).

[20] R. Kazman, S. Echeverria Galvez, and J. Ivers, "Extensibility", Carnegie Mellon University, Tech. Rep., 2022. [Online]. Available: https://kilthub.cmu.edu/ndownloader/files/34714069.

[21] G. Giacomo Guerreschi and M. Smelyanskiy, "Practical Optimization for Hybrid Quantum-Classical Algorithms", *arXiv e-prints*, 2017.

[22] J. R. McClean, J. Romero, R. Babbush, and A. Aspuru-Guzik, "The theory of variational hybrid quantum-classical algorithms", *New Journal of Physics*, vol. 18, no. 2, 2016.

[23] A. Schrijver *et al.*, *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, 2003, vol. 24, p. 58.

[24] B. H. Korte, J. Vygen, B. Korte, and J. Vygen, *Combinatorial optimization*. Springer, 2011, vol. 1.

[25] *IBMő Decision Optimization CPLEXő Modeling for Python IBMő Decision Optimization CPLEXő Modeling for Python (DOcplex) V2.25 documentation*. [Online]. Available: https://ibmdecisionoptimization.github.io/docplex-doc/ (visited on 05/16/2025).

[26] V. Sharma, N. S. B. Saharan, S.-H. Chiew, *et al.*, "OpenQAOA–An SDK for QAOA", *arXiv Preprint arXiv:2210.08695*, 2022.

[27] *IBM Debuts Next-Generation Quantum Processor & IBM Quantum System Two, Extends Roadmap to Advance Era of Quantum Utility*. [Online]. Available: https://newsroom.ibm.com/2023-12-04-IBM-Debuts-Next-Generation-Quantum-Processor-IBM-Quantum-System-Two,-Extends-Roadmap-to-Advance-Era-of-Quantum-Utility (visited on 03/10/2025).

[28] X. Liu, A. Angone, R. Shaydulin, I. Safro, Y. Alexeev, and L. Cincio, "Layer VQE: A Variational Approach for Combinatorial Optimization on Noisy Quantum Computers", *IEEE Transactions on Quantum Engineering*, vol. 3, pp. 1–20, 2022.

[29] S. Yarkoni, E. Raponi, T. Bäck, and S. Schmitt, "Quantum annealing for industry applications: Introduction and review", *Reports on Progress in Physics*, vol. 85, no. 10, 2022.

[30] S. Heng, D. Kim, T. Kim, and Y. Han, "How to solve combinatorial optimization problems using real quantum machines: A recent survey", *IEEE Access*, vol. 10, 2022.

[31] K. Danach and I. Moukadem, "Enhanced optimization of heat exchanger systems using the branch and cut method: A case study in industrial thermodynamics", *Letters in High Energy Physics*, 2024.

[32] R. K. Nath, H. Thapliyal, and T. S. Humble, "A Review of Machine Learning Classification Using Quantum Annealing for Real-World Applications", *SN Computer Science*, vol. 2, no. 5, 2021.

[33] J. Lee, *A First Course in Combinatorial Optimization*. Cambridge University Press, 2004, vol. 36.

[34] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver, "Combinatorial Optimization", *Unpublished manuscript*, vol. 10, pp. 75–93, 1994.

[35] J. Panovska-Griffiths, C. C. Kerr, W. Waites, and R. M. Stuart, "Mathematical modeling as a tool for policy decision making: Applications to the COVID-19 pandemic", *Handbook of Statistics*, vol. 44, p. 291, 2021.

[36] K. L. Hoffman and M. Padberg, "Traveling salesman problem", in *Encyclopedia of Operations Research and Management Science*, Springer, 2001, pp. 849–853.

[37] S. P. Bradley, A. C. Hax, and T. L. Magnanti, *Applied Mathematical Programming - Chapter 9: Integer Programming*, Companion slides prepared by José Fernando Oliveira and Maria Antónia Carravilla, 1977.

[38] C. E. Miller, A. W. Tucker, and R. A. Zemlin, "Integer programming formulation of traveling salesman problems", *Journal of the ACM (JACM)*, vol. 7, no. 4, pp. 326–329, 1960.

[39] D. Davendra, Ed., *Traveling Salesman Problem: Theory and Applications*. IntechOpen, 2010.

[40] A. Miseviius, T. Blaauskas, J. Blonskis, and J. Smolinskas, "An overview of some heuristic algorithms for combinatorial optimization problems", *Information Technology and Control*, vol. 30, no. 1, 2004.

[41] F. Peres and M. Castelli, "Combinatorial Optimization Problems and Metaheuristics: Review, Challenges, Design, and Development", *Applied Sciences*, vol. 11, 2021.

[42] M. Gendreau and J.-Y. Potvin, Eds., *Handbook of Metaheuristics* (International Series in Operations Research & Management Science). Springer, 2010, vol. 146.

[43] S. Boyd and J. Mattingley, "Branch and bound methods", *Notes for EE364b, Stanford University*, vol. 2006, 2007.

[44] S. R. Eddy, "What is dynamic programming?", *Nature Biotechnology*, vol. 22, no. 7, pp. 909–910, 2004.

[45] A. Lucena and J. E. Beasley, "Branch and cut algorithms", in *Advances in Linear and Integer Programming*, J. E. Beasley, Ed., Oxford University Press, 1996, pp. 187–222.

[46] Z. Michalewicz, *How to Solve it: Modern Heuristics*. Springer Science & Business Media, 2013.

[47] K. Taunk, S. De, S. Verma, and A. Swetapadma, "A brief review of nearest neighbor algorithm for learning and classification", in *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*, IEEE, 2019, pp. 1255–1260.

[48] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing", *Science*, vol. 220, pp. 671–680, 1983.

[49] A. H. Gandomi, X.-S. Yang, and S. Talatahari, *Metaheuristic Algorithms in Modeling and Optimization*. Springer, 2013.

[50] F. Glover, "Tabu search: A tutorial", *Interfaces*, vol. 20, no. 4, pp. 74–94, 1990.

[51] T. Kadowaki and H. Nishimori, "Quantum annealing in the transverse ising model", *Phys. Rev. E*, vol. 58, pp. 5355–5363, 5 1998.

[52] L. Binkowski, G. KoSSmann, T. Ziegler, and R. Schwonnek, "Elementary proof of QAOA convergence", *New Journal of Physics*, vol. 26, 2024.

[53] N. P. De Leon, K. M. Itoh, D. Kim, *et al.*, "Materials challenges and opportunities for quantum computing hardware", *Science*, vol. 372, 2021.

[54] L. Kotthoff, "Algorithm Selection for Combinatorial Search Problems: A Survey", *AI Magazine*, vol. 35, no. 3, pp. 48–60, 2014.

[55] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "Scipy 1.0: Fundamental algorithms for scientific computing in python", *Nature Methods*, vol. 17, no. 3, pp. 261–272, 2020.

[56] R. Wille, R. Van Meter, and Y. Naveh, "IBMs Qiskit Tool Chain: Working with and Developing for Real Quantum Computers", in *Design, Automation & Test in Europe Conference & Exhibition*, 2019, pp. 1234–1240.

[57] Qiskit, *What are Qiskit Primitives?*, 2023. [Online]. Available: https://medium.com/qiskit/what-are-qiskit-primitives-9bf63c1eacc7 (visited on 11/12/2024).

[58] *Amazon-braket-sdk: An open source library for interacting with quantum computing devices on Amazon Braket*. [Online]. Available: https://github.com/amazon-braket/amazon-braket-sdk-python (visited on 11/06/2024).

[59] *Ocean Developer Tools | D-Wave*. [Online]. Available: https://www.dwavesys.com/solutions-and-products/ocean/ (visited on 10/31/2024).

[60] *Solver Properties and Parameters D-Wave System Documentation documentation*. [Online]. Available: https://docs.dwavesys.com/docs/latest/doc_solver_ref.html (visited on 11/05/2024).

[61] V. Choi, "Minor-Embedding in Adiabatic Quantum Computation: I. The Parameter Setting Problem", *Quantum Information Processing*, vol. 7, 2008.

[62] V. Gilbert, J. Rodriguez, and S. Louise, "Benchmarking quantum annealers with near-optimal minor-embedded instances", in *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 1, 2024, pp. 531–537.

[63] M. K. Haghighi and N. Dimopoulos, *Minimum-length chain embedding for the phase unwrapping problem on D-Wave's advantage architecture*, arXiv:2309.10296, 2023.

[64] T. Liu, Z. Li, and M. J. Dinneen, *Graph Minor Embedding for Adiabatic Quantum Computing*. Centre for Discrete Mathematics and Theoretical Computer Science, 2021.

[65] S. Hadfield, Z. Wang, B. OGorman, E. G. Rieffel, D. Venturelli, and R. Biswas, "From the Quantum Approximate Optimization Algorithm to a Quantum Alternating Operator Ansatz", *Algorithms*, vol. 12, p. 34, 2019, Number: 2 Publisher: Multidisciplinary Digital Publishing Institute.

[66] A. Abran, Ed., *Guide to the software engineering body of knowledge: 2004 version*. Los Alamitos, Calif: IEEE Computer Soc, 2005.

[67] C. Cook and M. Visconti, "Documentation is important", *CrossTalk*, vol. 7, no. 11, pp. 26–30, 1994.

[68] J. Zhi, V. Garousi-Yusifolu, B. Sun, G. Garousi, S. Shahnewaz, and G. Ruhe, "Cost, benefits and quality of software development documentation: A systematic mapping", *Journal of Systems and Software*, vol. 99, pp. 175–198, 2015.

[69] D. Goodger and G. van Rossum, *PEP 257 Docstring Conventions | peps.python.org*. [Online]. Available: https://peps.python.org/pep-0257/ (visited on 11/27/2024).

[70] G. van Rossum, B. Warsaw, and A. Coghlan, *PEP 8 Style Guide for Python Code | peps.python.org*. [Online]. Available: https://peps.python.org/pep-0008/ (visited on 11/27/2024).

[71] X. Jia, "The Role and Importance of Software Testing in Software Quality Management", *Journal of Industry and Engineering Management*, vol. 1, no. 4, pp. 39–44, 2023.

[72] Atlassian, *The different types of testing in software*. [Online]. Available: https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing (visited on 12/04/2024).

[73] M. Cohn, *Succeeding with agile: software development using Scrum*. Pearson Education, 2010.

[74] A. Molina, *Crafting Test-Driven Software with Python: Write test suites that scale with your applications' needs and complexity using Python and PyTest*. Packt Publishing Ltd, 2021.

[75] X. Y. Djam, N. V. Blamah, and M. E. Ezema, "A Comparative Evaluation of Test Coverage Techniques Effectiveness", *Journal of Software Engineering and Applications*, vol. 14, no. 04, pp. 95–109, 2021.

[76] M. Jiménez, "An infrastructure for autonomic and continuous long-term software evolution", PhD thesis, University of Victoria, 2022.

[77] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda", *Journal of Systems and Software*, vol. 123, pp. 176–189, 2017.

[78] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*, 7th printing 2013. Upper Saddle River, NJ: Addison-Wesley, 2011.

[79] GitHub, *Understanding GitHub Actions*. [Online]. Available: https://docs.github.com/en/actions/about-github-actions/understanding-github-actions (visited on 11/29/2024).

[80] S. Joshi, "Piqture: A quantum machine learning library for image processing", Master of Science Thesis, University Institute of Engineering and Technology, Panjab University, 2021.

[81] K. Henttonen, M. Matinlassi, E. Niemela, and T. Kanstren, "Integrability and Extensibility Evaluation from Software Architectural Models - A Case Study", *The Open Software Engineering Journal*, vol. 1, no. 1, pp. 1–20, 2007.

[82] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[83] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[84] K. Kralj, *4 Smart Ways to Avoid Pitfall of Too Many Method Parameters*, Section: Software Architecture, 2023.

[85] A. Hunt, *The pragmatic programmer*. Pearson Education India, 1900.

[86] *Qiskit 1.0 release summary | IBM Quantum Computing Blog*. [Online]. Available: https://www.ibm.com/quantum/blog/qiskit-1-0-release-summary (visited on 01/13/2025).

[87] *Qiskit 0.40 release notes*. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/release-notes/docs.quantum.ibm.com/api/qiskit/release-notes/0.40 (visited on 01/13/2025).

[88] Z. Wang, N. C. Rubin, J. M. Dominy, and E. G. Rieffel, "XY mixers: Analytical and numerical results for the quantum alternating operator ansatz", *Physical Review A*, vol. 101, no. 1, 2020.

[89] F. G. Fuchs, K. O. Lye, H. Møll Nilsen, A. J. Stasik, and G. Sartor, "Constraint preserving mixers for the quantum approximate optimization algorithm", *Algorithms*, vol. 15, no. 6, p. 202, 2022.

[90] *What is a reasonable code coverage % for unit tests (and why)?* [Online]. Available: https://www.qodo.ai/question/what-is-a-reasonable-code-coverage-for-unit-tests-and-why/ (visited on 03/03/2025).

[91] *Qiskit 1.0 release notes*. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/release-notes/1.0 (visited on 02/25/2024).

[92] M. Mitchell, *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT press, 1998. [Online]. Available: http://www.ir.juit.ac.in:8080/jspui/bitstream/123456789/5350/1/An%20Introduction%20to%20Genetic%20Algorithms.pdf.