Guidelines for the Application of Hybrid Software Design Patterns

Michal Baczyk[1] and Ricardo Pérez-Castillo[2]
[1]*Alarcos Research Group, University of Castilla-La Mancha, Ciudad Real, Spain*
[2]*Alarcos Research Group, University of Castilla-La Mancha, Talavera de la Reina, Spain*

Keywords: Quantum Software Engineering, Patterns, Design Patterns, Circuit Design, Quantum Algorithms.

Abstract: Quantum Software Engineering is increasingly leveraging *design patterns* to codify best practices for developing quantum algorithms and applications. In this work, we conduct an extensive review of academic sources and open-source projects focused on quantum software design patterns. We identify dozens of recurring patterns spanning quantum algorithm structure, state preparation, data encoding, hybrid quantum-classical workflows, variational algorithms, and execution strategies. We organize these patterns into a unified framework, providing a guide detailing each pattern's qubit and gate requirements, classical processing needs, and categorization relevant from the application perspective. We observe a key trend of the expansion of pattern catalogs to support hybrid variational algorithms and NISQ-era challenges (e.g., warm-starting, circuit cutting), and the emergence of patterns to improve modularity, reusability, and interoperability of quantum software. Our findings aim to guide practitioners in applying proven design solutions in quantum application development.

1 INTRODUCTION

Quantum computing promises exponential speedups for certain classes of problems, but designing correct and efficient quantum programs remains challenging. As the field matures, there is an increasing need for systematic software engineering principles tailored to quantum computing (Piattini et al., 2020; Serrano et al., 2022). The *Talavera Manifesto* for Quantum Software Engineering highlights the importance of improving quantum software quality through higher-level abstractions and best practices.

In practice, quantum software rarely exists in isolation but is integrated with classical components, forming *hybrid* quantum-classical systems. A well-defined software architecture for such hybrid systems fosters key benefits: it enables scalable integration of emerging quantum hardware, supports modular and reusable designs, and provides high-level abstractions that simplify maintenance and evolution. Moreover, it facilitates smooth interoperability among different hardware providers (e.g., IBM, Rigetti, IonQ, QuEra, etc.) and helps standardize best practices that promote reliability and predictability in quantum software development.

In classical software engineering, **design patterns** have long been used to capture reusable solutions to common design problems (Gamma et al., 1994; Buschmann et al., 1996). Likewise, researchers have begun identifying **quantum software design patterns**: recurring strategies that arise in quantum algorithm design and hybrid quantum-classical programming (Leymann, 2019; Weigold et al., 2022). In recent years, a nascent pattern language for quantum computing has emerged (Weigold et al., 2021; Jiménez-Fernández et al., 2023), documenting core *quantum algorithm patterns* such as state initialization, oracles, and amplitude amplification (Bühler et al., 2023; Truger et al., 2024), as well as *data encoding patterns* (Weigold et al., 2022; Bühler et al., 2023), patterns for *variational and hybrid algorithms* (Weigold et al., 2021), and *execution strategies* (Bühler et al., 2023; Georg et al., 2023; Bechtold et al., 2023).

These patterns aim to improve reusability, maintainability, and interoperability in quantum software, yet there are few guidelines on how to systematically select and apply them. This gap hinders the effective design of hybrid quantum-classical software architectures, undermining the realization of their benefits. To address this, we conduct a review and synthesis of quantum software design patterns reported in academic literature and gleaned from real-world implementations. Our analysis extends beyond published research to open-source repositories (e.g., Qiskit, PennyLane), extracting practical insights into

Baczyk, M. and Pérez-Castillo, R.
Guidelines for the Application of Hybrid Software Design Patterns.
DOI: 10.5220/0013539500004525
In *Proceedings of the 1st International Conference on Quantum Software (IQSOFT 2025)*, pages 105-111
ISBN: 978-989-758-761-0

how these patterns are employed in actual quantum development workflows.

In this paper, we identify **all relevant design patterns** for hybrid quantum-classical software systems and organize them into logical categories. Table 1 provides a unified view of these patterns, detailing their quantum resource requirements (number of qubits and gates), classical pre-/post-processing, and typical use cases. We exclude low-level error correction or mitigation patterns – we focus on higher-level algorithmic and software-structural guidance.

Section 2 presents a comprehensive catalog of quantum software design patterns, summarizing their resource footprints, classical components, and known limitations, as well as discussing how they address key challenges in the current Noisy Intermediate-Scale Quantum (NISQ) era. We then illustrate, in Section 3, how modern frameworks such as Qiskit, Cirq, PennyLane, and Amazon Braket implement these patterns through abstractions for oracle creation, data encoding, and hybrid optimization loops. By consolidating these insights, we aim to provide a clear set of guidelines for selecting and applying design patterns in the development of robust hybrid quantum-classical software systems.

2 QUANTUM SOFTWARE DESIGN PATTERNS AND THEIR APPLICABILITY

Table 1 summarizes quantum software design patterns. Each entry highlights:

- **Name / Purpose**: Brief description of the pattern and the main algorithmic function.
- **Resource Footprint**: Number of qubits, types of gates, and expected circuit depth.
- **Classical Processing**: Details on any pre-/post-processing or real-time feedback loops.
- **Topology Constraints**: Special hardware connectivity considerations.
- **Typical Applications**: Common use cases and algorithmic settings.
- **Limitations**: Known challenges, such as high gate counts or noise sensitivity.
- **Reference**: Key literature references for further reading.

These patterns can be grouped into high-level categories. The *Algorithm Core* patterns address fundamental quantum operations like *Initialization* or *Oracle* building, crucial to tasks such as search or decision problems. *Data Encoding* patterns define how classical data is loaded into quantum states, impacting resource requirements and circuit depth. *Hybrid Algorithms* focus on parameterized or iterative approaches that integrate classical optimization. *Software Module* patterns promote modular, maintainable code through reusable components, while *Execution and NISQ* patterns attend to real-world implementation details, especially under limited qubit counts and noisy conditions.

2.1 Applicability in Hybrid Quantum-Classical Workflows

The *Classical Processing* and *Topology Constraints* columns in Table 1 are particularly relevant to hybrid approaches. Patterns like **Variational Quantum Algorithm** or **Ad-hoc Hybrid** emphasize iterative feedback loops and repeated measurements. Here, hardware constraints and available classical computation strongly influence pattern selection. For example, **Circuit Translator** patterns are vital when matching a logical circuit to the hardware coupling map or when bridging different quantum frameworks.

2.2 Guidelines for Pattern Selection

In this subsection, we present guidelines for practitioners to help them choose design patterns for hybrid quantum-classical applications. Our approach is based upon the analysis patterns outlined in Table 1, which incorporates both usual considerations (e.g., *Resource Footprint*, *Classical Processing*, *Topology Constraints*) and increasingly important factors (e.g., *Problem Addressed*, *Quantum-Classical Interaction Type*, *Scalability Considerations*). By considering all those factors, we aim to help developers identify patterns that align with their specific goals, hardware resources, and software architecture.

Below we outline the step-by-step process.

1. **Identify the Core Challenge and Relevant Pattern Category:**
   Begin by clarifying the central problem the hybrid system is intended to solve. Whether the goal involves data initialization (*Amplitude Encoding*, *Basis Encoding*), processing functionality (*Oracle*, *Entanglement*), or optimization (*Variational Quantum Algorithm*), identifying this focus narrows down which categories (e.g., Algorithm Core, Data Encoding, Hybrid Algorithms) are most useful.
2. **Estimate the Resource Footprint and Complexity:**

Table 1: Quantum software design patterns: Core algorithmic patterns, data-encoding strategies, hybrid algorithms, modular software components, and NISQ execution strategies.

Name / Purpose	Resource Footprint	Classical Processing	Topology Constraints	Typical Applications	Limitations	Reference
Algorithm Core						
Initialization	Requires qubits for the problem size; no gates for the all-zero state (default for many SDKs). Minimal depth if standard.	No classical overhead unless advanced prep is used (classical data might define angles).	Not sensitive to hardware connectivity if no entangling gates.	Fundamental for most quantum algorithms (simulation, cryptography, etc.).	Gate overhead grows for non-trivial states; prone to prep errors on noisy devices.	(Bühler et al., 2023; Leymann, 2019)
Uniform Superposition	Needs as many qubits as the input size; typically one Hadamard per qubit. Depth about one plus overhead.	No big classical overhead; optional control for adaptive sequences.	Single-qubit operations only, weakly dependent on connectivity.	Common in Grover's search, amplitude amplification.	Phase errors may accumulate; precise single-qubit calibrations needed.	(Bühler et al., 2023; Weigold et al., 2022)
Entanglement	At least two qubits; uses controlled gates (CNOT, CZ). Depth depends on pairs.	Usually no classical overhead. Possibly some coordination in hybrid programs.	Important if qubits are not directly connected; may require gate routing.	Central in teleportation, superdense coding, error correction, EPR pairs.	Sensitive to decoherence and cross-talk; higher error rates.	(Leymann, 2019; Bühler et al., 2023)
Oracle	Qubits depend on the function; custom multi-qubit gates or sequences. Depth follows function complexity.	Classical pre-processing to build the reversible circuit. Post-processing to check oracle results.	Routing needed if connectivity is limited. Multi-controls might need ancillas.	Key for search, Boolean function evaluations, etc.	Large oracles consume resources; mapping from classical logic can be complex.	(Georg et al., 2023)
Uncompute	Often uses ancillas; same gate count as forward subcircuit in reverse. Depth roughly doubled.	No extra classical overhead; purely quantum. Possibly some logic for subcircuit choice.	Constraints mirror the original subcircuit.	Clears ancilla qubits (restoring them to zero). Common in arithmetic, phase estimation.	Doubles depth, increasing overall error.	(Bühler et al., 2023)
Amplitude Amplification	Depends on scale; oracle plus diffuser repeated. Gate count and depth scale with iterations.	Optional measurement at the end. Feedback loops in adaptive variants.	Limited by multi-qubit gate connectivity.	Used in Grover's search, combinatorial optimization.	Iterations grow as square root of problem size; gate errors accumulate.	(Bechtold et al., 2023)
Data Encoding						
Basis Encoding	Uses $\lceil \log_2(k) \rceil$ qubits for k states; may need bit-flip gates or none if already binary. Very low depth.	Classical data in binary. Minimal post-processing unless partial qubit measurement.	Few connectivity demands; mostly single-qubit or simple controls.	For categorical states, classification, simpler quantum tasks.	Inefficient for large k; misses amplitude-based advantages.	(Weigold et al., 2022)
Amplitude Encoding	$\lceil \log_2(M) \rceil$ qubits for M-dim data. Potentially $O(M)$ gates for arbitrary loads.	Classical pre-processing to get angles or normalize data. Post-processing to check fidelity.	Can be connectivity-heavy if multi-qubit gates are needed.	Used in Quantum Machine Learning, Principal Component Analysis, or amplitude-based speedups.	Large data = big gate count; noise sensitivity.	(Bühler et al., 2023)
Angle Encoding	One qubit per feature; single-qubit rotations. Depth is how many rotations.	Feature values mapped to angles. Post-processing checks final correlations.	Low connectivity demands if no entangling layers.	Popular in near-term QML, classifiers, variational circuits.	Limited features per qubit; rotation errors matter.	(Bühler et al., 2023; Georg et al., 2023)
Quantum Random Access Memory Encoding	Needs $\log_2(M)$ address qubits + data qubits. Multi-controls can grow depth.	Classical data must be organized for random access. Post-processing reads out lines.	Complex routing if connectivity is limited; multi-controls.	Enables large-scale QML or database queries in superposition.	Building true QRAM is hard; multi-controlled gates raise error.	(Georg et al., 2023)
Quantum Associative Memory	Qubit count scales with pattern dimension. Summation of basis states. Depth varies by retrieval method.	Might need classical query for retrieval. Post-processing identifies matched pattern.	Controlled gates might be required; connectivity can matter.	Used in pattern matching, content-addressable searches, some QML.	Scalability uncertain; large circuits can be error-prone.	(Weigold et al., 2022)
Hybrid Algorithms						
Variational Quantum Algorithm	Problem-dependent qubits; param. circuit layers (single-/two-qubit gates). Depth depends on layer count.	Classical optimizers update gate params after measurements. Pre-/post- steps for param setup routines.	Needs mid-circuit measurement or repeated runs. Entangling layers may need good connectivity.	Foundational for near-term heuristics: QML, chemistry, optimization.	Sensitive to noise, barren plateaus, slow or stuck classical optimization.	(Weigold et al., 2021)
Variational Quantum Eigensolver/ Quantum Approximate Optimization Algorithm	Qubits match system/graph. Param. ansatz with single-/two-qubit gates. Depth grows with layering.	Needs iterative classical optimization. Post-processing obtains final params/energies.	Connectivity shapes ansatz. Distant qubits may need swaps.	VQE for ground-state energies, Quantum Approximate Optimization Algorithm for combinatorial tasks.	Excess depth undermines advantage; more layers raise gate errors.	(Weigold et al., 2021)
Warm-starting	Same qubit/gate needs as main approach, plus minor overhead for classical solutions.	A classical solver seeds initial parameters; post-processing is standard.	No extra constraints beyond regular variational circuits.	Speeds up hybrid tasks (portfolio, max-cut) with classical seeds.	Depends on seed quality; may fail if guess is poor.	(Truger et al., 2024)
Software Module						
Quantum Module	Qubits/gates depend on module function. Depth varies with routines.	Classical inputs set params; post-processing reads states or outcomes.	Depends on module. Many entangling gates can be connectivity-heavy.	Reusable quantum logic (e.g. arithmetic, oracles, transformations).	Large modules can be resource-heavy; must integrate carefully.	(Bühler et al., 2023)
Hybrid Module	Any size, combining quantum subcircuits + classical routines. Depth includes quantum + classical overhead.	Typically has quantum-classical loops, measurements feed classical logic.	Quantum portion needs connectivity. Classical part may add latency.	Used for end-to-end hybrid solutions, iterative or data-driven workflows.	Complex debugging/tuning; repeated hardware calls can be slow.	(Bühler et al., 2023)
Circuit Translator	Rewrites circuits without direct qubit use. Gate count depends on decomposition.	Uses classical logic to optimize or rewrite gates.	Must match hardware coupling, so connectivity is key.	Enables cross-framework or cross-hardware circuit compatibility.	Poor translations can increase depth or gates, reducing fidelity.	(Georg et al., 2023)
Execution and Noisy Intermediate-Scale Quantum (NISQ)						
Standalone Execution	Follows final compiled circuit's qubits/gates. Depth per chosen algorithm.	Minimal overhead, typically just job submission and result retrieval.	Compile-time connectivity handling, no dynamic feedback.	For quick prototyping of small circuits without advanced mitigation.	Constrained by hardware coherence/fidelity; no built-in mitigation.	(Piattini et al., 2020)
Ad-hoc Hybrid	Same as circuit, plus overhead for repeated loop runs.	Host code runs pre-/post- each quantum run for parameter/data adjustments.	No extra constraints, but repeated calls are time-consuming.	Prototyping or small-scale hybrid demos in research settings.	Inefficient for large param sweeps; lacks advanced resource management.	(Weigold et al., 2021)
Pre-deployed	Supports any qubit/gate scale, subject to practical limits. Repeated instantiation.	Classical logic routes jobs, collects results, handles scheduling.	Connectivity matters for distributed or cloud hardware.	Used by cloud-based enterprise solutions and repeated/persistent quantum jobs.	Scheduling/queue overhead adds latency; limited real-time control or debugging.	(Georg et al., 2023)
Circuit Cutting	Splits large circuits into subcircuits. Each subcircuit fits local qubit limit.	Combines subcircuit measurement results for global outcome.	Connectivity matters inside subcircuits; classical stitching logic is key.	Distributes large computations for resource-limited hardware.	Post-processing grows with cuts; noise accumulates.	(Bechtold et al., 2023)

Next, examine the *Resource Footprint* column for an overview of the qubit count, gate depth, and other hardware demands each pattern may introduce. Patterns with sparse multi-qubit operations

(e.g., **Angle Encoding**) often suit near-term hardware constraints, while resource-intensive approaches (e.g., **Oracle**, **Quantum RAM**) may require more complex routing and additional qubits. Evaluating these needs in light of available resources can avoid bottlenecks and impractical designs.

3. **Determine the Quantum-Classical Interaction Style:**
   Consider how the quantum and classical components will communicate. For instance, **Standalone Execution** assumes minimal back-and-forth, whereas **Variational Quantum Algorithm** and **Hybrid Module** patterns rely on frequent quantum-classical feedback loops (e.g., parameter updates, dynamic data processing). By matching the quantum-classical interface implementations to the system's intended architecture, developers can achieve the right level of efficiency.
4. **Check Hardware Constraints (Topology, Noise, Qubit Count):**
   Many design patterns are restricted by hardware limitations. The *Topology Constraints* column highlights how connectivity or noise characteristics can influence pattern selection. On hardware with linear qubit arrays or high error rates, techniques like **Circuit Cutting** (splitting a large circuit into smaller pieces) or more straightforward patterns (**Initialization**, **Uniform Superposition**) can offer practical trade-offs. However, patterns involving complex multi-qubit gates (**Entanglement**, **Oracle**) may demand additional considerations, such as error mitigation or deeper circuit scheduling.
5. **Weigh Scalability and Performance Considerations:**
   As quantum hardware and application needs scale, certain patterns can become gate-intensive or prone to noise accumulation (e.g., **Amplitude Encoding**, **QRAM Encoding**). It is therefore important to assess how the pattern performs with an increasing number of qubits or in repeated quantum-classical optimization loops.
6. **Review Examples and Known Limitations:**
   Analyzing the *Typical Applications* and *Limitations* columns provides real-world insights into each pattern's strengths and pitfalls. As an example, **Uncompute** is critical for cleaning up ancillas but effectively doubles circuit depth, while **Variational Quantum Eigensolver** can target chemistry or optimization tasks but suffers from barren plateaus if not carefully tuned. Published tutorials (e.g., Qiskit's VQE examples, PennyLane's template library) offer further perspectives on implementation nuances and best practices.
7. **Plan for Modularity and Future Extensions:**
   Finally, we recommend that developers consider a pattern's potential for reuse and evolution. Approaches like **Quantum Module** or **Hybrid Module** enable a modular structure in which parameterized blocks or new classical routines can be swapped in without re-inventing the core quantum logic. When architectures must evolve to accommodate new hardware or newly developed algorithms, using encapsulated designs can simplify transitions and protect long-term code stability.

By following these seven steps, practitioners can systematically navigate the rich spectrum of patterns in Table 1 and select solutions that balance hardware realities, resource availability, and desired functionality. This approach offers a structured path for building quantum-classical workflows that are both efficient and maintainable, helping developers avoid common integration problems and accelerate the path from design to deployment.

3 REAL-WORLD IMPLEMENTATIONS OF QUANTUM SOFTWARE DESIGN PATTERNS

Current quantum software frameworks (e.g., Qiskit (IBM, 2025), Cirq (Google, 2025), PennyLane (Xanadu, 2025), Amazon Braket (AWS, 2025)) provide concrete examples of how these patterns can be integrated in practice. Table 2 illustrates how foundational patterns (*Initialization*, *Oracle*) and hybrid strategies (*Quantum-Classic Split*) appear on platforms.

A notable observation is that some patterns, such as **Initialization** and **Superposition**, are almost universally required in quantum applications and are therefore typically straightforward to implement. On the other hand, certain patterns critical for demonstrating quantum advantage, such as **Entanglement**, can be introduced with only a few gates (e.g., forming Bell or GHZ states) yet demand careful hardware consideration when scaling. Meanwhile, **Oracle** patterns are necessarily problem-specific and often packaged as reusable modules, as exemplified by Qiskit's built-in logic oracles and Braket's Grover implementation. These readily available components allow developers to insert custom functionality while retaining established best practices for circuit composition and execution.

Table 2: Examples of quantum design pattern implementations in major frameworks.

Design Pattern	Example Implementation (Framework)
Initialization (State Preparation)	Qiskit's `initialize()` function prepares arbitrary basis states (e.g., setting an initial state vector) (IBM, 2025). PennyLane provides an `AmplitudeEmbedding` template to load classical data into a quantum state (Xanadu, 2025).
Uniform Superposition	Typically achieved by applying Hadamard gates to all qubits in a register. For instance, Qiskit's Grover algorithm applies $H^{\otimes n}$ to n qubits to create an equal superposition over 2^n basis states (IBM, 2025).
Entanglement	Entangling operations are implemented by multi-qubit gates. A two-qubit Bell state is generated by first applying a Hadamard gate to one qubit, followed by a CNOT gate on the other qubit. More generally, frameworks use a cascade of CNOTs (or CZs) to create GHZ states (one H gate and a chain of CNOTs for n qubits) (Google, 2025). PennyLane offers predefined entangling-layer templates that apply such controlled gates across qubit pairs (Xanadu, 2025).
Oracle (Black Box)	Qiskit includes a `PhaseOracle` class for constructing oracle circuits from Boolean logic expressions. Amazon Braket's algorithm library provides a pre-built Grover's algorithm that accepts a problem oracle and iterates the oracle+diffusion pattern. Cirq and PennyLane enable oracles by allowing custom unitary subcircuits to be defined for a given problem.
Uncompute (Cleanup)	Many algorithms explicitly "uncompute" temporary results to discard ancillary qubits. In practice, this is done by inverting the subcircuit that produced the entangled ancilla. For example, implementations of Shor's algorithm uncompute the modular exponentiation circuit to disentangle the output register before measurement (restoring ancillas to $\|0\rangle$). Frameworks like Qiskit facilitate this via an automatic circuit inverse method (e.g., `qc.inverse()`).
Quantum-Classic Split (Hybrid)	Hybrid variational algorithms intermix quantum circuits with classical optimization. PennyLane intrinsically supports this pattern by feeding quantum circuit evaluations into optimizers and providing gradient computation tools. Similarly, Qiskit's VQE and QAOA implementations run a quantum subroutine to evaluate an objective, then update parameters classically in a loop. Amazon Braket also enables hybrid workflows through its SDK (often via PennyLane integration).

Another insight is the importance of uncomputation (cleanup of ancilla qubits). While initialization and entanglement patterns are ubiquitous, uncomputation is crucial in algorithms with workspace qubits or "garbage" outputs, to avoid leaving residual entanglement. However, an empirical study by Pérez-Castillo et al.(2024) observed that developers sometimes neglect the uncompute step in practice(Pérez-Castillo et al., 2024), which can lead to incorrect results if not handled. Tools and best practices are now emerging to detect and enforce this pattern in quantum programs (Pérez-Castillo et al., 2024).

Finally, the quantum-classical split (hybrid) pattern has become central in the NISQ era, and all major frameworks support it. High-level libraries for variational algorithms (e.g., Qiskit's `algorithm` runtime modules and PennyLane's optimization routines) provide built-in loops that alternate quantum circuit executions with classical computations, automating the training of quantum models (Bergholm et al., 2018). This integration enables complex workflows like VQE with minimal user code, abstracting the optimization loop as part of the framework. Industry use-cases, especially in quantum chemistry and finance, rely on such patterns; for example, Qiskit's chemistry module (now part of Qiskit Nature) uses classical pre- and post-processing around quantum subcircuits to solve molecular problems, exemplifying a domain-specific classical-quantum interface.

Overall, these examples illustrate that quantum software design patterns are not just theoretical proposals from software engineering research, but are actively informing the development of reusable libraries and tools. By encapsulating recurring solutions—whether preparing a uniform superposition or orchestrating a hybrid optimization loop—into framework-level constructs, developers can program at a higher level of abstraction. This leads to more reliable and maintainable quantum software, echoing the Talavera Manifesto's call for principled quantum software engineering (Piattini et al., 2020). Each pattern in Table 2 thus represents a step toward a

standardized body of quantum software engineering knowledge, bridging the gap between high-level algorithm design and low-level circuit implementation.

4 CONCLUSION

In this paper, we have conducted an extensive review of the quantum software design patterns proposed in the literature and observed in open-source frameworks. We identified a wide range of patterns that address the unique challenges of quantum application development, from foundational circuit-building strategies to advanced hybrid paradigms integrating classical optimization. By categorizing these patterns according to their resource requirements, classical processing needs, and typical use cases, we provide a structured guide for developers to select, adapt, and implement solutions that are both conceptually clear and hardware-aware.

Importantly, we examined how these patterns are already implemented in leading quantum SDKs such as Qiskit, Cirq, PennyLane, and Amazon Braket, illustrating real-world applicability. Our findings indicate that many of these frameworks have begun to incorporate design patterns as high-level abstractions, thereby improving maintainability, reducing duplicated effort, and helping developers avoid common pitfalls. We also highlight emerging trends—particularly around NISQ-era constraints and hybrid algorithms by claiming further refinement and expansion of existing design patterns.

Overall, this work aims to strengthen the foundation of quantum software engineering by offering both a practical catalog of proven design solutions and evidence of their real-world adoption. As the field evolves, continuous refinement of these patterns will be essential to align with new hardware capabilities, overcome scaling limitations, and foster best practices that bridge quantum algorithm research with robust software engineering principles.

ACKNOWLEDGMENTS

This work has been supported by projects SMOOTH (PID2022-137944NB-I00), QU-ASAP (PDC2022-133051-I00) funded by MCIU/ AEI/ 10.13039/ 501100011033 and by the “European Union NextGenerationEU/ PRTR”, and financial support for the execution of applied research projects, within the framework of the UCLM Own Research Plan, co-financed at 85% by the European Regional Development Fund (ERDF) UNION (2022-GRIN-34110).

REFERENCES

AWS (2025). Amazon braket documentation. https://docs.aws.amazon.com/braket/. Accessed: 2025-02-07.

Bechtold, M., Barzen, J., Beisel, M., Leymann, F., and Weder, B. (2023). Patterns for Quantum Circuit Cutting. In *Proceedings of the 30th Conference on Pattern Languages of Programs (PLoP ’23)*. The Hillside Group. in press.

Bergholm, V., Izaac, J., Schuld, M., Gogolin, C., et al. (2018). Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968*.

Bühler, F., Barzen, J., Beisel, M., Georg, D., Leymann, F., and Wild, K. (2023). Patterns for quantum software development. In *15th International Conference on Pervasive Patterns and Applications (PATTERNS)*.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., and Booch, G. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Georg, D., Barzen, J., Beisel, M., Leymann, F., Obst, J., Vietz, D., Weder, B., and Yussupov, V. (2023). Execution Patterns for Quantum Applications. In *Proceedings of the 18th International Conference on Software Technologies - ICSOFT*, pages 258–268. SciTePress.

Google (2025). Cirq documentation. https://quantumai.google/cirq. Accessed: 2025-02-07.

IBM (2025). Qiskit: An open-source framework for quantum computing – documentation. https://qiskit.org/documentation/. Accessed: 2025-02-07.

Jiménez-Fernández, S., Cruz-Lemus, J., and Piattini, M. (2023). A systematic mapping study on quantum circuits design patterns. *Proceedings of the 25th International Conf. on Enterprise Information Systems (ICEIS)*.

Leymann, F. (2019). Towards a pattern language for quantum algorithms. In *Quantum Technology and Optimization Problems*, volume 11413 of *Lecture Notes in Computer Science (LNCS)*, pages 218–230, Cham. Springer International Publishing.

Pérez-Castillo, R., Fernández-Osuna, M., Piattini, M., and Romero-Yáñez, E. (2024). A preliminary study of the usage of design patterns in quantum software. In *5th Int. Workshop on Quantum Software Engineering (Q-SE 2024)*. In press.

Piattini, M., Peterssen Nodarse, G., Pérez-Castillo, R., Hevia Oliver, J. L., Serrano, M., Hernández González, G., Guzmán, I., Andrés Paradela, C., Polo, M., Murina, E., Jiménez Navajas, L., Marqueño, J., Gallego, R., Tura, J., Phillipson, F., Murillo, J., Niño, A., and Rodríguez, M. (2020). The talavera manifesto for quantum software engineering and programming.

Serrano, M. A., Pérez-Castillo, R., and Piattini, M. (2022). *Quantum Software Engineering*. Springer Nature. https://link.springer.com/book/10.1007/978-3-031-05324-5.

Truger, F., Barzen, J., Beisel, M., Leymann, F., and Yussupov, V. (2024). Warm-Starting Patterns for Quantum Algorithms. In *Proceedings of the 16th International Conf. on Pervasive Patterns and Applications (PATTERNS 2024)*, pages 25–31. Xpert Publishing Services (XPS).

Weigold, M., Barzen, J., Leymann, F., and Salm, M. (2022). Data encoding patterns for quantum computing. In *Proceedings of the 27th Conf. on Pattern Languages of Programs*, PLoP '20, USA. The Hillside Group.

Weigold, M., Barzen, J., Leymann, F., and Vietz, D. (2021). Patterns for Hybrid Quantum Algorithms. In *Proceedings of the 15th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2021)*, pages 34–51. Springer International Publishing.

Xanadu (2025). Pennylane templates library (v0.40.0). https://docs.pennylane.ai/en/stable/introduction/templates.html. Accessed: 2025-02-07.