Quantum Computing needs Classical Software (Engineering)

Lukas Burgholzer [1]

Abstract: To unlock the full potential of quantum computing, the field must develop a robust software ecosystem grounded in classical software engineering principles—a discipline that itself needs to be adapted to the quantum domain. Currently, however, the advancement of quantum computing is hampered by a growing "software gap," where a fragmented and non-standardized ecosystem of tools hinders progress. Much of the existing quantum software originates from academic proofs-of-concept that often lack the robustness, maintainability, and scalability required for practical applications. This article argues that bridging this gap requires the systematic application of principles from classical software engineering and design automation. We present the Munich Quantum Toolkit (MQT) as a case study, showcasing a suite of open-source tools built on these principles. By tracing its evolution, we distill key lessons learned in testing, deployment, and community-driven development. We demonstrate that adopting a rigorous software engineering culture is essential for building the reliable and sustainable software infrastructure needed to unlock the full potential of quantum computing.

Keywords: Quantum Computing, Software Engineering, Quantum Software, Quantum Software Engineering

1 Introduction

The remarkable capabilities of modern classical computing are built upon a mature and sophisticated ecosystem of hardware and software. For decades, Moore's Law drove exponential growth in hardware performance, but it is the parallel advancement in software that has truly unlocked its potential. This synergy has yielded powerful design automation tools, integrated development environments (IDEs), compilers, simulators, and verifiers—all underpinned by robust software engineering principles. This foundation enables the efficient development of complex systems, from high-level specifications down to low-level implementations, and forms the bedrock of our digital world.

As quantum computing (QC) transitions from a theoretical discipline to a practical technology, it will increasingly depend on a similarly robust classical software infrastructure. However, the principles of classical software engineering cannot simply be transferred; they must be re-examined and adapted for the unique challenges of the quantum realm. To unlock the potential of quantum hardware, a significant amount of classical software is required to tackle challenges analogous to those in the classical realm: algorithm and circuit generation (synthesis), optimization, simulation, compilation, verification, error correction, and application development. However, quantum computing is still in its nascent stages. While

[1] Chair for Design Automation, Technical University of Munich, Munich, Germany,
Munich Quantum Software Company GmbH, Garching near Munich, Germany,
lukas.burgholzer@tum.de, https://orcid.org/0000-0003-4699-1316

doi:10.18420/FIXME2017_04

quantum hardware is advancing at an impressive pace, the quantum software landscape remains fragmented, characterized by a plethora of languages, tools, and approaches with little standardization.

This situation is compounded by two factors. First, as an interdisciplinary field, many quantum software developers are physicists, mathematicians, or other scientists rather than trained software engineers. Second, a substantial portion of quantum software originates in academic settings, where the primary output is often a proof-of-concept or prototype. While invaluable for research, this software is frequently not developed with the rigor required for long-term use, scalability, or maintainability.

Without applying the proven principles of classical software engineering, the quantum domain risks a "software gap"—a scenario where powerful quantum hardware exists, but the software needed to operate it effectively is lacking. This article argues for the critical need to bridge this gap by systematically applying techniques from classical software engineering and design automation to the quantum computing stack. Drawing on over five years of experience from the Johannes Kepler University (JKU) Linz, the Munich Quantum Valley (MQV) initiative, the Chair for Design Automation at the Technical University of Munich, and the Munich Quantum Software Company (MQSC), we present the Munich Quantum Toolkit (MQT) as a case study. Through it, we showcase a successful application of these principles and distill key lessons learned for the future of quantum software development.

2 The Quantum Software Challenge

The "software gap" in quantum computing is a multifaceted challenge that begins with the hardware itself. The quantum hardware landscape is a diverse and rapidly evolving ecosystem. Quantum computers are being developed using various physical technologies—including superconducting qubits [DWM04], trapped ions [BCO23], neutral atoms [Wi23], and photonics [OFV09]—each with unique characteristics, native gate sets, and limitations. These platforms vary widely in scale and maturity, from small-scale academic testbeds to commercial systems with hundreds of qubits deployed in supercomputing centers.

This hardware diversity is mirrored in the software domain, creating a fragmented and complex environment. Major technology players have introduced their own software development kits (SDKs)—toolchains designed to streamline development for a specific platform—such as IBM's Qiskit [Ja24], Microsoft's Azure Quantum with Q# [Sv18], Google's Cirq [De25], and NVIDIA's CUDA-Q [te25b], each fostering its own ecosystem. Alongside these industrial efforts, a vast number of academic initiatives produce valuable but often isolated pieces of software, be it specialized compiler passes, efficient simulators, or novel optimization techniques. These contributions are frequently tied to a specific framework or hardware or are developed as standalone proofs-of-concept that are not maintained after the corresponding paper is published. This fragmentation extends down to the level of intermediate representations (IRs)—standardized data formats for representing quantum programs that

enable interoperability between different tools—where competing formats make it difficult to transfer quantum programs between different toolchains [SBW25].

At the same time, there is a natural division of labor between hardware-agnostic and hardware-specific tools. Vendor SDKs (e.g., Qiskit, Cirq, Azure Quantum, CUDA-Q) understandably optimize for their respective platforms and often expose device-specific functionality. In contrast, general-purpose, interoperable tools can focus on portable IRs, cross-framework translation, comparative benchmarking, and algorithm- and architecture-independent optimizations. The MQT is designed squarely for this latter space: it complements rather than replaces vendor stacks by offering fast simulators, mappers, verifiers, benchmarks, and an IR that aims to be a neutral exchange layer between ecosystems. Other end-to-end Python toolchains and libraries exist as well; our intent here is to highlight where open, vendor-agnostic components add the most value and how they can interoperate with platform SDKs.

The absence of a common software stack with clearly defined interfaces and exchange formats leads to significant negative consequences. The learning curve for newcomers is unnecessarily steep, as they must navigate a disconnected tool landscape. Effort is duplicated as teams independently solve fundamental problems like circuit optimization or data serialization. Crucially, this lack of standardization complicates verification and benchmarking. The current ecosystem often feels like a collection of tools stitched together with patchwork solutions, making it difficult to rigorously compare the performance of different algorithms, compilers, or hardware platforms.

This situation is further exacerbated by the cultural divide between academic research and industrial software development. Much of the quantum software today originates as a by-product of academic work, where the primary goal is a proof-of-concept for a publication. While invaluable for demonstrating new ideas, these prototypes often lack the documentation, testing, and stable APIs required for long-term maintainability and scalability. As authors move on to new projects, the software is often abandoned—a practice that is particularly problematic in a field as fast-moving as quantum computing. This creates a bottleneck where promising research fails to translate into robust, production-ready tools. Consequently, many new academic publications default to benchmarking against well-known but potentially outdated baselines, simply because the true state-of-the-art tools are either unavailable or no longer functional, hindering reproducible scientific progress. Ultimately, this fractured and immature software landscape forms a critical barrier, preventing the field from capitalizing on hardware advances and realizing the full potential of quantum computing.

3 A Design Automation Approach for Quantum Computing

To bridge the software gap, we advocate for a systematic approach rooted in the principles of classical Design Automation (DA), often called Electronic Design Automation (EDA) in the context of integrated circuits [WCC09]. In essence, design automation provides a

structured methodology for managing complexity in system design. It establishes a "design flow"—a sequence of automated steps that progressively transforms a high-level, abstract description of a problem into a low-level, detailed implementation for a specific physical system. This is achieved by defining clear abstraction layers, creating specialized tools for each step (e.g., synthesis, placement, routing), and using standardized data formats to ensure interoperability. Applying this philosophy to quantum computing means replacing ad-hoc development with a structured process that is both more reliable and efficient. This includes using efficient data structures and algorithms, defining clear abstraction layers, and emphasizing rigorous, automated testing—all hallmarks of good software practice.

A critical step in this design flow is *compilation*. In quantum computing, compilation is the process of translating an abstract quantum algorithm into a sequence of operations that can be executed on a specific quantum computer. This is a multi-stage process that includes decomposing high-level operations into the hardware's native gate set; optimizing the circuit to reduce its size and depth; and, crucially, mapping/logically placing qubits onto the device and scheduling operations under connectivity and timing constraints. While some substeps (e.g., basis decomposition) are mostly algorithmic, several core subproblems—such as qubit mapping/placement, routing under limited connectivity, and many optimization objectives—are NP-hard or NP-complete in general. Consequently, effective compilers and heuristics are essential for harnessing the power of today's noisy, intermediate-scale quantum hardware.

The **Munich Quantum Toolkit** (MQT; [Wi24]) serves as a case study for this philosophy. The MQT is a collection of free and open-source software (FOSS) tools developed by the Chair for Design Automation at the Technical University of Munich and supported by the Munich Quantum Software Company (MQSC) that embodies the design automation approach. Among others, it is part of the Munich Quantum Software Stack (MQSS; [Bu25b]) ecosystem, which is being developed as part of the Munich Quantum Valley (MQV) initiative. The MQT is not intended to replace existing tools but to complement them, fostering a healthy, interoperable ecosystem. The toolkit is built on over a hundred peer-reviewed publications with thousands of citations and has achieved significant adoption, with over 1,100 GitHub stars and 2.8 million downloads from the Python Package Index (PyPI). This illustrates that a design-automation-centric approach, combined with rigorous software engineering, can yield high-performance, reliable tools that benefit the entire community. The following is an illustrative overview of key MQT components; for a deeper dive, we refer readers to the MQT documentation [Wi24] and the underlying publications for each tool.

The foundation of the toolkit is **MQT Core** [Bu25a], which provides the essential data structures and algorithms for representing and manipulating quantum circuits. It features a unified intermediate representation (IR) that serves as the backbone for many other tools in the MQT ecosystem. This IR facilitates seamless translation to and from various quantum programming languages, including Qiskit and OpenQASM. MQT Core also incorporates state-of-the-art packages for decision diagrams (DDs; [WHB23]) and ZX-diagrams [va20], enabling the efficient representation and manipulation of quantum computations.

Building on this foundation, **MQT DDSIM** [WHL22] provides a high-performance classical simulator for quantum circuits. By using decision diagrams—a technique adapted from Binary Decision Diagrams (BDDs) in classical design automation [Br92]—DDSIM can efficiently simulate certain classes of circuits that are intractable for conventional statevector-based simulators. In one notable instance, this approach reduced the simulation time for an instance of Shor's algorithm from an estimated 30 days to a single minute [ZW19]. This demonstrates how established classical methods can be powerfully repurposed for the quantum domain.

For compilation, **MQT QMAP** [WB23] offers a powerful collection of techniques to map quantum circuits onto the specific constraints of diverse hardware platforms, including those based on superconducting qubits and neutral atoms. Since most compilation tasks are computationally hard, QMAP relies on sophisticated heuristics and advanced methods from classical computer science. It employs techniques such as satisfiability (SAT) solving [Bi21], heuristic search algorithms like A*, and graph theory to find optimized solutions in a reasonable timeframe [BM]. The mapping techniques pioneered in [WBZ19; ZPW19] were among the first efficient methods for this task and are still widely used as baselines, while its methods for neutral-atom architectures represent the state of the art [St24; St25].

To ensure the correctness of these complex compilation and optimization processes, **MQT QCEC** [BW21] offers an efficient tool for the formal equivalence checking of quantum circuits. While trivial for small circuits, this task becomes computationally hard for larger ones. QCEC leverages advanced techniques, including decision diagrams and the ZX-calculus [va20], to formally verify that a circuit's functionality remains unchanged after compilation. The methods underlying QCEC were the first to enable scalable and efficient verification of compilation flows for quantum circuits [BRW20].

To facilitate robust and reproducible evaluations, **MQT Bench** [QBW23] provides a comprehensive benchmark suite of quantum circuits and the corresponding generation logic. It spans the entire quantum software stack, offering a wide variety of quantum algorithms and compilation options for numerous target devices. Its extensible design allows researchers to add their own algorithms and platforms, promoting fair and thorough benchmarking of quantum software tools. MQT Bench has been widely adopted by the community, with well over 100 publications citing it.

To help users navigate the complex hardware landscape, **MQT Predictor** [QBW24] offers automatic device selection and device-specific circuit optimization. The framework combines classical compiler optimization techniques with machine learning to predict the most suitable quantum device for a given circuit. Practically, the MQT Predictor consumes publicly available backend specifications and calibration data via provider APIs where available (e.g., connectivity graphs, gate fidelities, gate times) and augments this with synthetic micro-benchmarks when needed. Predictions include both device choice and compilation options and are validated using held-out benchmarks to mitigate overfitting.

Finally, **MQT QECC** provides tools and methods for Quantum Error Correction (QEC; [DNM13]) and fault-tolerant quantum computing. In a field often dominated by theory, MQT QECC delivers practical implementations of QEC techniques, including novel decoders and synthesis methods for fault-tolerant circuits [Be24a; Be24b; Pe25]. This makes advanced QEC concepts accessible to a wider audience for practical application.

The MQT exemplifies how applying established techniques from classical design automation and adhering to robust software engineering practices—such as continuous integration and deployment (CI/CD), comprehensive documentation, and a rigorous testing culture—can lead to a reliable and accessible toolkit. The following section provides a brief recollection of the development of the MQT over the past five-plus years, highlighting some of the key lessons learned during its evolution.

4 Lessons Learned from Developing the MQT

The story of the MQT offers a firsthand look at the evolution of a software project in a rapidly advancing scientific field. It is a journey from academic proofs-of-concept to a robust, community-focused toolkit, illustrating the tangible benefits of embracing good software engineering practices.

Choose the right tool (and language) for the job. The MQT project began in late 2019 at JKU Linz under the "JKQ" (JKU Tools for Quantum; [WHB20]) label. Initially, the tools were developed as proofs-of-concept for academic publications—a means to an end for producing results in papers. Adhering to the community's call for more transparency, the code was made public on GitHub from the start. The initial focus was on performance, tackling computationally hard problems with efficient C++ implementations, as Python was deemed too slow and Rust was not yet a mature option. This early decision highlights a foundational lesson: for performance-critical scientific software, choosing a low-level language can be a pragmatic necessity, even if it diverges from the broader ecosystem's trends. More generally, the principle guided technology choices across the stack (e.g., data structures, verification methods, packaging) in service of usability and performance.

Automated testing is essential for correctness and reliability. As is common in academia, the initial code lacked a testing infrastructure; a successful run for a publication was often considered sufficient validation. This approach proved inadequate when subtle bugs led to incorrect results that were only discovered much later. This experience underscored a critical lesson: rigorous, automated testing is not optional overhead but is essential for ensuring scientific correctness and reliability. Adopting continuous integration (CI) through GitHub Actions to automatically run unit tests across Linux, macOS, and Windows revealed and helped fix numerous bugs and instances of undefined behavior before they could compromise research outcomes.

Enforce code quality to enable collaboration and scalability. The project's growth from individual efforts to a collaborative endeavor introduced new challenges. With multiple contributors, the codebase became a mixture of personal coding styles, hindering readability and maintenance. This led to another key lesson: as a team grows, enforcing code quality through automation is crucial for scalability and collaboration. We introduced a consistent coding style using automated formatters like `clang-format` and enforced best practices with linters like `clang-tidy`. These tools eliminated debates over formatting, improved code consistency, and served as an educational resource for new contributors, allowing the team to focus on the substance of their work.

A disciplined workflow is vital for stability and collaboration. Initially, development was done directly on the main branch, a workflow that quickly became unsustainable with more contributors. This necessitated a more structured process, leading to a further lesson: a disciplined contribution workflow is vital for maintaining code stability and enabling effective collaboration. We adopted the GitHub Flow model based on branches and pull requests (PRs) for all contributions, enabling code reviews to ensure quality and correctness before merging. Furthermore, using GitHub's issue tracker to manage bugs and feature requests brought structure to the development process, replacing ad-hoc communication and ensuring that important tasks were prioritized and addressed systematically.

Accessibility is as important as performance. While C++ provided performance, it created a barrier to entry in a Python-dominated ecosystem. This highlighted the importance of meeting users where they are. To bridge this gap, we developed Python bindings for our C++ code using the `pybind11` library [JRM17] and provided automatic, lossless translations to and from Qiskit's circuit objects. This allowed users to benefit from our high-performance tools within their familiar Python workflows. The lesson learned was clear: accessibility is as important as performance. Providing a convenient interface for the target community is essential for driving adoption and maximizing impact.

Own the complexity of your stack to lower the barrier to entry. However, Python bindings alone were not enough. The need to build the C++ code from source remained a significant hurdle for many users, particularly those without a traditional software engineering background. This led to a crucial step in usability: we established a continuous deployment (CD) pipeline using `cibuildwheel` [te25a] to pre-build and package the tools as Python wheels. This made the MQT tools "just a pip install away" on all major platforms and architectures. The lesson here is that developers must own the complexity of their stack. By handling the intricacies of the build process, we lowered the barrier to entry and allowed users to focus on their research instead of wrestling with installation issues.

Documentation is a first-class citizen. With the tools now easily accessible, comprehensive documentation became paramount. Retroactively adding documentation was a painful but necessary process that taught us another valuable lesson: documentation must be a first-class citizen in the development process. We now write documentation alongside the code, covering everything from installation and getting-started guides to detailed API descriptions, all automatically built and hosted online. This commitment ensures that the tools are not just available, but truly usable by the community.

Sustainability requires continuous investment. Finally, the fast-paced nature of quantum computing means that software can quickly become obsolete. In academia, this often leads to abandoned projects. To ensure the MQT's longevity, we made a deliberate choice to build it as a free and open-source software (FOSS) project with active maintenance and community engagement. The final lesson is that sustainability requires more than just good code; it requires continuous investment in the project, including regular updates and community support. These experiences illustrate how a project can evolve from a collection of proofs-of-concept into a robust toolkit by embracing the principles of classical software engineering and design automation.

5 Conclusion

The advancement of quantum computing is at a critical juncture where progress is no longer solely dependent on hardware breakthroughs but is increasingly limited by the maturity of its software ecosystem. This article has argued that the prevailing "software gap"—characterized by a fragmented landscape of tools, a lack of standardization, and a disconnect between academic prototypes and production-ready software—poses a significant threat to realizing the potential of quantum technology.

To address this challenge, we have advocated for the systematic application of principles from classical software engineering and design automation. By establishing a structured quantum design flow, we can create a more coherent, reliable, and efficient software stack. The Munich Quantum Toolkit (MQT) serves as a concrete case study of this philosophy in action. Its evolution from a collection of academic proofs-of-concept to a comprehensive, open-source toolkit demonstrates the transformative power of adopting rigorous software engineering practices. The lessons learned throughout its development—from implementing automated testing and deployment pipelines to prioritizing documentation and community engagement—underscore that building high-quality quantum software is as much about process and discipline as it is about novel algorithms.

As the field moves towards fault-tolerant quantum computing, the need for robust and sustainable software will only intensify. The challenges ahead require a collective commitment from the community to move beyond isolated prototypes and embrace a culture of collaborative, high-quality software development. By building on the proven foundations of classical

software engineering, we can construct the resilient software infrastructure necessary to unlock the future of quantum computing.

Interoperability will benefit from, but also requires more than, a common IR. Ongoing community efforts such as OpenQASM 3 and QIR/MLIR-inspired approaches aim at portable representations; yet, achieving true interoperability also demands stable APIs (such as the hardware-software interface QDMI [**qdmi**] that has been jointly developed with partners in the Munich Quantum Valley), conformance test suites, reference translators, and shared benchmarks. A pragmatic path forward is a layered standard: a small core IR, well-defined extension points, and rigorous compliance tooling.

While this article focused on the gate-based (universal) model, several ideas and components (e.g., benchmarking methodology, parts of the IR, verification concepts) may extend to other paradigms such as quantum annealing or photonic computing. Assessing the scope and ROI of such extensions is promising future work.

Finally, successful adoption hinges on integrating QC into classical applications and scientific software as well as traditional High-Performance Computing (HPC). This includes mapping business/scientific problems to hybrid quantum-classical workflows, developing user-friendly interfaces, and ensuring seamless integration with existing infrastructures. Today, the MQT focuses on the core QC tooling; moving forward, we plan to extend its capabilities to better support these full-stack applications as part of the Munich Quantum Software Stack (MQSS; [Bu25b]).

References

[BCO23] Bernardini, F.; Chakraborty, A.; Ordóñez, C.: Quantum computing with trapped ions: a beginner's guide, 2023, arXiv: 2303.16358.

[Be24a] Berent, L. et al.: Analog Information Decoding of Bosonic Quantum Low-Density Parity-Check Codes. PRX Quantum 5 (2), p. 020349, 2024, arXiv: 2311.01328.

[Be24b] Berent, L. et al.: Decoding quantum color codes with MaxSAT. Quantum 8, p. 1506, 2024, arXiv: 2303.14237.

[Bi21] Biere, A. et al., eds.: Handbook of Satisfiability - Second Edition. IOS Press, 2021.

[BM] Bondy, A.; Murty, U. S. R.: Graph Theory. Springer International Publishing.

[Br92] Bryant, R. E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Compute. Surv. 24 (3), pp. 293–318, 1992.

[BRW20] Burgholzer, L.; Raymond, R.; Wille, R.: Verifying results of the IBM Qiskit quantum circuit compilation flow. In: Int'l Conf. on Quantum Computing and Engineering. 2020.

[Bu25a] Burgholzer, L. et al.: MQT Core: The Backbone of the Munich Quantum Toolkit (MQT). Journal of Open Source Software 10 (108), p. 7478, 2025.

[Bu25b] Burgholzer, L. et al.: The Munich Quantum Software Stack: Connecting End Users, Integrating Diverse Quantum Technologies, Accelerating HPC, 2025, arXiv: 2509.02674.

[BW21] Burgholzer, L.; Wille, R.: Advanced equivalence checking for quantum circuits. IEEE Trans. on CAD of Integrated Circuits and Systems, 2021.

[De25] Developers, C.: Cirq, version 1.6.0, 2025, https://github.com/quantumlib/Cirq.

[DNM13] Devitt, S. J.; Nemoto, K.; Munro, W. J.: Quantum error correction for beginners. Rep. Prog. Phys. 76 (7), p. 076001, 2013.

[DWM04] Devoret, M. H.; Wallraff, A.; Martinis, J. M.: Superconducting qubits: A short review, 2004, arXiv: cond-mat/0411174.

[Ja24] Javadi-Abhari, A. et al.: Quantum computing with Qiskit, 2024, arXiv: 2405.08810.

[JRM17] Jakob, W.; Rhinelander, J.; Moldovan, D.: pybind11 – Seamless operability between C++11 and Python, https://github.com/pybind/pybind11, 2017.

[OFV09] O'Brien, J. L.; Furusawa, A.; Vučković, J.: Photonic quantum technologies. Nature Photon 3 (12), pp. 687–695, 2009.

[Pe25] Peham, T. et al.: Automated Synthesis of Fault-Tolerant State Preparation Circuits for Quantum Error-Correction Codes. PRX Quantum 6, p. 020330, 2025.

[QBW23] Quetschlich, N.; Burgholzer, L.; Wille, R.: MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. Quantum 7, p. 1062, 2023.

[QBW24] Quetschlich, N.; Burgholzer, L.; Wille, R.: MQT Predictor: Automatic Device Selection with Device-Specific Circuit Compilation for Quantum Computing. ACM Transactions on Quantum Computing, 2024.

[SBW25] Stade, Y.; Burgholzer, L.; Wille, R.: Towards Supporting QIR: Steps for Adopting the Quantum Intermediate Representation. In: Workshops of the Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. 2025.

[St24] Stade, Y. et al.: An Abstract Model and Efficient Routing for Logical Entangling Gates on Zoned Neutral Atom Architectures. In: Int'l Conf. on Quantum Computing and Engineering. 2024, arXiv: 2405.08068.

[St25] Stade, Y. et al.: Optimal State Preparation for Logical Arrays on Zoned Neutral Atom Quantum Computers. In: Design, Automation and Test in Europe. 2025, arXiv: 2411.09738.

[Sv18] Svore, K. M. et al.: Q#: Enabling scalable quantum computing and development with a high-level domain-specific language. Proc. of the RWDSL, 2018, arXiv: 1803.00652.

[te25a] cibuildwheel development team, T.: cibuildwheel, version 3.1.3, 2025, https://github.com/pypa/cibuildwheel.

[te25b] development team, T. C.-Q.: CUDA-Q, version 0.12.0, 2025, https://github.com/NVIDIA/cuda-quantum.

[va20] van de Wetering, J.: ZX-calculus for the working quantum computer scientist, 2020, arXiv: 2012.13966.

[WB23] Wille, R.; Burgholzer, L.: MQT QMAP: Efficient quantum circuit mapping. In: Int'l Symp. on Physical Design. 2023.

[WBZ19] Wille, R.; Burgholzer, L.; Zulehner, A.: Mapping quantum circuits to IBM QX architectures using the minimal number of SWAP and H operations. In: Design Automation Conf. 2019.

[WCC09] Wang, L.-T.; Chang, Y.-W.; Cheng, K.-T., eds.: Electronic Design Automation. Elsevier, 2009.

[WHB20] Wille, R.; Hillmich, S.; Burgholzer, L.: JKQ: JKU tools for quantum computing. In: Int'l Conf. on CAD. 2020.

[WHB23] Wille, R.; Hillmich, S.; Burgholzer, L.: Decision Diagrams for Quantum Computing. In (Topaloglu, R. O., ed.): Design Automation of Quantum Computers. Springer, 2023.

[WHL22] Wille, R.; Hillmich, S.; Lukas, B.: Tools for quantum computing based on decision diagrams. ACM Transactions on Quantum Computing, 2022.

[Wi23] Wintersperger, K. et al.: Neutral atom quantum computing hardware: Performance and end-user perspective, 2023, arXiv: 2304.14360.

[Wi24] Wille, R. et al.: The MQT Handbook: A Summary of Design Automation Tools and Software for Quantum Computing. In: IEEE International Conference on Quantum Software (QSW). 2024, arXiv: 2405.17543, A live version of this document is available at https://mqt.readthedocs.io.

[ZPW19] Zulehner, A.; Paler, A.; Wille, R.: An efficient methodology for mapping quantum circuits to the IBM QX architectures. IEEE Trans. on CAD of Integrated Circuits and Systems, 2019.

[ZW19] Zulehner, A.; Wille, R.: Advanced simulation of quantum computations. IEEE Trans. on CAD of Integrated Circuits and Systems, 2019.