Qiskit Code Migration with LLMs

José Manuel Suárez[1,2], Luís Mariano Bibbó[1,2],

Joaquín Bogado[1,2], Ignacio Schwindt[1,2], and Alejandro Fernandez[1,2,3]

[1] Laboratorio de Investigación y Formación en Informática Avanzada (LIFIA)
[2] Facultad de Informática - Universidad Nacional de La Plata, Buenos Aires, Argentina
[3] Comisión de Investiación Científica - CIC Bs.As.

Abstract. The rapid evolution of Quantum Development Kits (QDKs) introduces a specific form of technical debt that compromises code maintainability and hinders software reuse. In the specialized domain of Quantum Software Engineering (QSE), this challenge is intensified by the scarcity of high-quality training data and the high volatility of emerging frameworks, which often lead general-purpose Large Language Models (LLMs) to produce unreliable or hallucinated results. This paper proposes a hybrid approach integrating LLMs with Retrieval-Augmented Generation (RAG) to automate the migration of Qiskit code across versions. The proposed methodology enhances the precision and reliability of migration suggestions by leveraging an automatically generated taxonomy of migration scenarios as the structured, version-specific knowledge source to guide the models. The approach is implemented through an automated, extensible workflow evaluating LLMs (Google Gemini Flash-2.5 and OpenAI Gpt-oss-20b) under different retrieval schemes (unconstrained and restrictive). Results demonstrate that the taxonomy-based RAG architecture, particularly under the restrictive scheme, significantly reduces hallucinations and improves descriptive quality, with Google Gemini Flash-2.5 showing superior performance in detecting complex refactoring scenarios. These findings confirm the potential of this data-centric methodology to foster technological independence and provide robust, intelligent assistants that mitigate API obsolescence, ensuring the long-term availability of quantum algorithms within a rapidly shifting ecosystem and flattening the learning curve within Quantum Software Engineering (QSE).

Keywords: Quantum Software Engineering · Code Migration · Large Language Models · Qiskit · Generative Artificial Intelligence

1. Introduction

Driven by hardware innovations, simulation, and optimized techniques in error correction and fault tolerance (Adermann et al., 2026; Wildon, 2026; Aasen et al., 2025), the rapid progress of QC is accelerating the evolution of QDKs (Ahmad et al., 2025; Gill et al., 2025). This expansion has diversified the QC community beyond physics and mathematics to include fields like computer science, biology, and engineering. However, this interdisciplinary shift imposes significant challenges on development tools, particularly regarding accessibility and the learning curve. Moreover, it highlights the need to address legacy issues from classical software engineering (CSE), such as API obsolescence triggered by disruptive version releases.

The convergence of QC and CSE has given rise to an emerging discipline: quantum software engineering (QSE) (Khan et al., 2025; Mandal et al., 2025; Murillo et al., 2025), a field that adapts empirical methodologies to the software lifecycle of hybrid quantum-classical systems within NISQ-era constraints (Lammers et al., 2025; Lau et al., 2022; Preskill, 2018). Concurrently, AI agents—specifically Large Language Models (LLMs) (Minaee et al., 2025; Xiao and Zhu, 2025)—have proven effective in managing complex tasks, prompting a fundamental rethink of workflows and tools within the discipline.

Previous stages demonstrated the constructive feasibility of a taxonomy that consolidates migration scenarios (Suárez et al., 2025b) and established its efficacy in guiding LLMs for scenario detection and the generation of refactoring suggestions (Suárez et al., 2025a). The present work integrates: (i) generative artificial intelligence (GenAI [4]) elements (Liang et al., 2024; Sengar et al., 2024; Weisz et al., 2024), through the configurable download and execution of LLMs; (ii) low-code/no-code (LCNC) tools [5], aiming to automate the experimental workflow and ensure its extensibility; (iii) a retrieval-augmented generation (RAG) architecture (Gao et al., 2024; Gupta et al., 2024; Mombaerts et al., 2024), to configure a knowledge base that complements the models' generic information; and (iv) the traceability of our experimental base, supported by the semi-automatic execution of scenario detection and suggestion generation tasks, as well as code migration and metrics analytics previously developed for the Qiskit QDK (Aragonés-Soria and Oriol, 2025; Pathak et al., 2025; «Qiskit/qiskit», 2025) [6].

API obsolescence and the resulting technical gap are critical challenges in CSE, particularly in QC due to the accelerated evolution of QDKs, which threatens algorithmic functionality and service availability (Z. Chen et al., 2026; Panter and Eisty, 2026; Liang et al., 2024; C. Wang et al., 2024; X. Wang et al., 2024). While LLMs streamline development, they face reliability issues in data-scarce emerging domains like QC (Dobariya and Kumar, 2025; Zheng et al., 2025; J. Chen and Mueller, 2023). This research develops AI assistants to address QSE challenges related to API obsolescence and quantum algorithm availability,

[4] IBM - Generative AI https://www.ibm.com/es-es/think/topics/generative-ai

[5] IBM - LCNC https://www.ibm.com/think/topics/low-code

[6] IBM - Qiskit https://www.ibm.com/quantum/qiskit

aiming for agile adaptation and robust migration criteria in volatile ecosystems (Aragonés-Soria and Oriol, 2024).

Once the components of the solution-oriented approach were defined, the continuity of our research line was outlined, as illustrated in Figure 1. In the initial stage (Suárez et al., 2025b), we evaluated the utility of a taxonomy designed to consolidate authoritative information on API evolution following disruptive (major) Qiskit releases. This stage employed a hybrid constructive approach (manual and LLM-automated) to establish comparative baselines and verify the potential of LLMs in building comprehensive, high-quality information sources. In the subsequent stage (Suárez et al., 2025a), metrics were generated to compare effectiveness in detecting migration scenarios and the precision of adaptation suggestions, using a battery of synthetic Python [7] code snippets. In the current stage, the effort focuses on integrating the automated migration scenario taxonomy into a RAG architecture to complement the models' general knowledge with contextually relevant, authoritative, and updated information. Additionally, this approach is channeled through an automated workflow that streamlines the experimental framework.

This work contributes: (i) a taxonomy-based RAG architecture using semantic databases; (ii) an automated, collaborative workflow; (iii) an impact analysis of different retrieval schemes; and (iv) automated code validation to augment expert review. The rest of this paper is structured as follows: Section 2 reviews related work; Section 3 details the methodology; Section 4 presents the results; Section 5 discusses the findings. Section 6 concludes with the main conclusions and outlines future research directions. It also includes a section of appendices for those who wish to explore the technical details mentioned in greater depth.

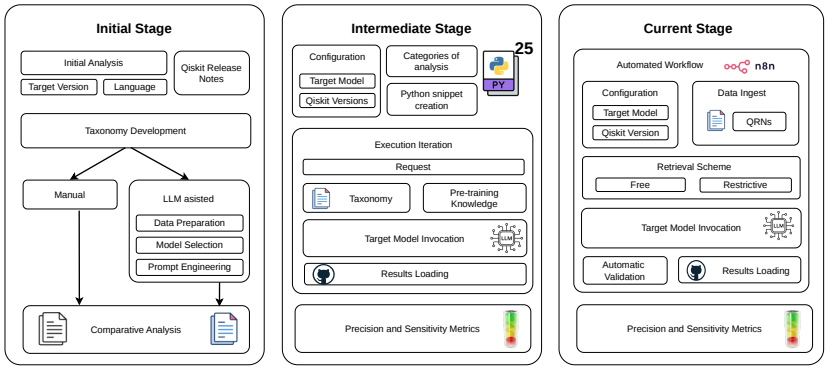


Fig. 1. Key stages of our research line.

Initial Stage -Taxonomy of migration scenarios for Qiskit refactoring using LLMs (Suárez et al., 2025b).

Intermediate Stage - Automatic Qiskit Code Refactoring Using LLMs (Suárez et al., 2025a).

[7] Python Documentation - https://docs.python.org/3/

2. Related work

Regarding the intersection of code refactoring, RAG and LLMs, Bellur et al., 2025 introduced MM-ASIST, an IDE-integrated assistant. However, their study focuses exclusively on 'Move Method' refactoring, while our work is not intended to be limited to any specific technique, but rather to cover migrations related to custom libraries, deprecations, updates, and changes in control flow. Hostnik and Robnik-Šikonja, 2025 compared RAG strategies for Python code completion using small local models to prioritize privacy. While this work also addresses reliability, custom metrics and expert evaluation are utilized, noting that small models may prove insufficient for QSE complexities. Closser and Kabala, 2025 evaluated four public LLMs for quantum circuit synthesis, identifying Gemini—utilized in these experiments—as the most efficient. Siavash and Moin, 2025 explored Qiskit code generation from UML via MDE and RAG. Although they aim to mitigate hallucinations through authoritative repositories, their work does not address refactoring. Finally, Li et al., 2025 investigated strategies to enhance LLM capabilities and reduce hallucinations. This approach can be seen as a domain-specific instance of their proposed schemes, though further study is required to fully evaluate the impact of specialized prompting within QSE.

Considering studies with a higher degree of correlation, Henderson et al., 2025 evaluated LLMs' ability to write Qiskit code to flatten the learning curve in QC. While general models may suffice for basic algorithm generation, we argue that migrating existing code across specific versions requires a RAG-based architecture with domain-specific knowledge. Campbell et al., 2025 compared Chain-of-Thought (CoT) strategies with RAG for Qiskit-based algorithm generation and error correction. They noted that standard RAG often fails due to outdated information—a limitation our version-specific taxonomy architecture specifically addresses. Other approaches include fine-tuning for high-quality Qiskit code generation Dupuis et al., 2024 and specialized data generation for Pennylane Asif et al., 2025. In the industry, Microsoft's Azure Quantum Copilot incorporates RAG but does not target refactoring or version migration. Similarly, Kheiri et al., 2025 introduced QSpark, a fine-tuned assistant for generating Qiskit code from scratch. Unlike these model-centric approaches, our work is data-centric, focusing on taxonomies to refactor existing code, thereby avoiding the complexity and cost of fine-tuning. Finally, Abufarha et al., 2026 proposed a hybrid assistant to reduce prompting dependency in refactoring and testing. While these studies share contact points with our research, they do not specifically address code migration within the context of API obsolescence in QSE.

3. Methodology

The experimental design centers on constructing a semantic database through a retrieval technique that provides models with automatically generated, structured information (Minaee et al., 2025; Yuan et al., 2024). This approach offers a pragmatic alternative to relying solely on the model's internal knowledge and avoids costly, complex strategies such as fine-tuning (Gao et al., 2024). By incorporating RAG, we aim to steer retrieval by prioritizing reliable data from the underlying taxonomies, thereby reducing hallucinations and improving refactored code accuracy. To operationalize this, we implemented a highly flexible workflow using n8n [8], based on a GitHub-hosted project [9] developed in Visual Studio Code [10]. This workflow encapsulates the experimental framework, enabling the execution of both local and remote models, pre- and post-processing stages, integration with external services, and the automatic uploading of results to the source repository.

Using n8n aims to enhance agility, flexibility, and scalability while reducing technical overhead (Pattnayak and Bohra, 2025; Heuschkel, 2023). The experimental workflow includes the following key stages (see Figure 2): (i) Global Configuration, which defines functional parameters to adapt experimental schemes, including the semantic database, test models, GitHub repository, Qiskit version, retrieval strategy, and verification stages; (ii) Data Ingestion, which extracts release notes and associated automated taxonomies from the source repository into a semantic database; (iii) Data Processing, which handles test snippets and prompts—generated in either free or restrictive mode—and establishes tracking metadata; (iv) Request Execution, the core loop where each iteration loads a test scenario, prompts, and queries the target model, while the RAG mechanism manages access to both local data and pre-acquired knowledge; (v) Automated Validation; and (vi) Results Storage, which automatically uploads experimental results.

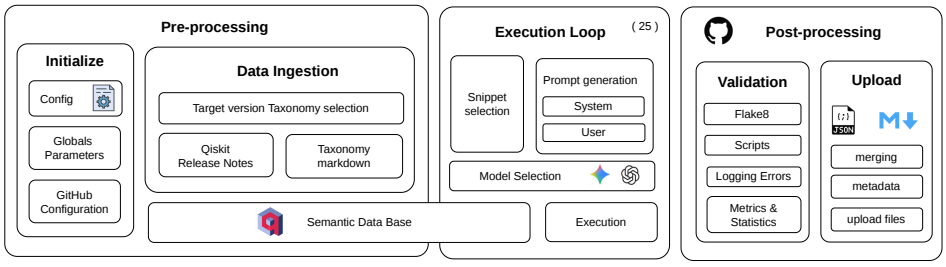


Fig. 2. Workflow stages in n8n

[8] n8n tool - https://docs.n8n.io/
[9] GitHub - https://docs.github.com/en
[10] Visual Studio Code IDE - https://code.visualstudio.com/docs

The models used were **Open AI Gpt-oss-20b** (OpenAI, 2026; OpenAI et al., 2024) and **Google Gemini Flash-2.5** (Comanici et al., 2025; Team et al., 2024), keeping default configurations. The official information sources were Qiskit Release Notes [11], as the basis for generating the automatic taxonomy of migration scenarios.

To ensure the study's self-containment, the fundamental structural aspects of the taxonomy are detailed herein, although its full development process and comprehensive format can be found in our previous work (Suárez et al., 2025b). As illustrated in Figure 6 (Appendix A), the main fields of the taxonomy encompass the technical category, version migration flow, summary, associated artifacts, source and target application code fragments, estimated difficulty level, Quantum Software Engineering (QSE) alignment, and relevant literature references.

Regarding the evaluation process, a one-shot prompt engineering strategy executed in English was implemented. This approach utilizes two alternating system prompts within the execution loop—each tailored to a specific testing strategy—alongside a single user prompt that supplies the corresponding test snippet (see Figures 8 and 7 in Appendix B). While the user prompt remains constant, the system prompts formally define the model's role, the experimental objectives, the version constraints, knowledge base guidelines, and a strict tabular response format (Markdown [12]) detailing the behavioral rules for each column.

In the post-analysis stage, the detected scenarios were quantified and the accuracy of recommendations was classified using our metric, allowing for comparative contrast in the line of research. We decided to work with two retrieval schemes: (i) a free scheme, where the requests made to the model are limited to experimental functional requirements without biasing the model on the retrieval of information; and (ii) a restrictive scheme, where, in addition to functional requirements, the model is directed towards the retrieval of information from the Qdrant semantic database [13]. Within the context of RAG, the role of the vector database extends beyond efficient data management; it serves as the core component for external knowledge retrieval through similarity search in a latent space. This bridges the semantic gap between the model queries and the document corpus, preserving the relevant context for the subsequent generation phase. Each of the 25 Python test code snippets was carefully designed to test a specific feature or functionality (imports, deprecations, deprecated classes, workflow and API updates); they range from 10 to 30 lines in length and typically define sequential logic without functional nesting or class definitions; since the main objective is to evaluate how the models respond to that specific, limited, and manageable change.

[11] Qiskit Release Notes - https://quantum.cloud.ibm.com/docs/en/api/qiskit/release-notes
[12] Markdown - https://www.markdownguide.org/
[13] Qdrant Database - https://qdrant.tech/documentation/

4. Results

Following the proposed experimental pipeline and methodology, results are summarized in Tables 1 and 2. Evaluation employed the stoplight metrics established by Suárez et al., 2025a. Two experts and a software engineer who regularly works with Qiskit participated in the process of analyzing and evaluating the results. They worked independently, without knowing the results of others' evaluations, which model was being evaluated or the scheme used. The test models, Google Gemini Flash-2.5 and OpenAI Gpt-oss-20b, were evaluated according to the workflow described previously. The distribution of scenarios returned by each model was analyzed based on the imposed retrieval schemes. Detailed scenario distributions according to the traffic-light metrics are presented in Table 2; for more details, see Appendix D.

Table 1. Summary of results for test models and information retrieval modes

Case	Description	Gemini Flash-2.5		Gpt-oss-20b	
		Free	Restrictive	Free	Restrictive
X	Incorrect scenario and inadequate suggestion	16	15	13	13
X+	Incorrect scenario or inadequate suggestion	4	5	4	3
OK-	Suggestion needs adaptation	5	9	11	14
OK	Correct scenario and suggestion	80	81	31	37
P perfect	Expected scenarios detected	16	16	9	10
P global	Detected vs. expected scenarios (ratio)	85 [89] (0.95)	90 [89] (1.01)	42 [89] (0.47)	51 [89] (0.57)
Total detected scenarios		105	110	59	67

[a] P perfect: Scenarios in which the model's predictions matched those expected.
[b] P global: Total number of detected cases vs. total number of expected cases.

Table 2. Distribution of scenarios according to stoplight metrics

Source	Free		Restrictive	
	Gemini Flash-2.5	Gpt-oss-20b	Gemini Flash-2.5	Gpt-oss-20b
Taxonomy (Tax)	83 8 4 5 66	15 1 2 3 9	100 13 5 8 74	50 6 3 11 30
Internal (IK)	22 8 0 0 14	41 12 2 8 19	10 2 0 1 7	3 1 0 1 1
Balance Tax/IK	83/22 (79/21%)	15/41 (27/73%)	100/10 (91/9%)	50/3 (94/6%)
Total	105 16 4 5 80	56 13 4 11 28	110 15 5 7 83	53 7 3 12 31

● **X: Incorrect** ● **X+: Inadequate** ● **OK-: Req. adjustments** ● **OK: Correct**

Note: The colored subtotals (4) correspond to the stoplight metrics for that case (source, retrieval scheme and target model).

5. Discussion

Regarding the balance of scenario sources and their distribution in relation to the configured retrieval scheme, see Table 2, allows us to verify the impact of these strategies on the proposed architecture. For Google Gemini Flash-2.5, a 79/21 ratio is shown in free mode, while in restricted mode the imbalance deepened to 91/9. For OpenAI Gpt-oss-20b, it shifted from 27/73 in free mode to 94/6, also implying a reversal in trend; see Figure 3

Observations show that Google Gemini Flash-2.5 relies more on authoritative information than OpenAI Gpt-oss-20b, a difference that impacts scenario detection accuracy. Since this margin remains relatively constant regardless of the retrieval strategy, the variation appears to be driven by inherent model characteristics, including parameterization and context window size, rather than the retrieval scheme itself.

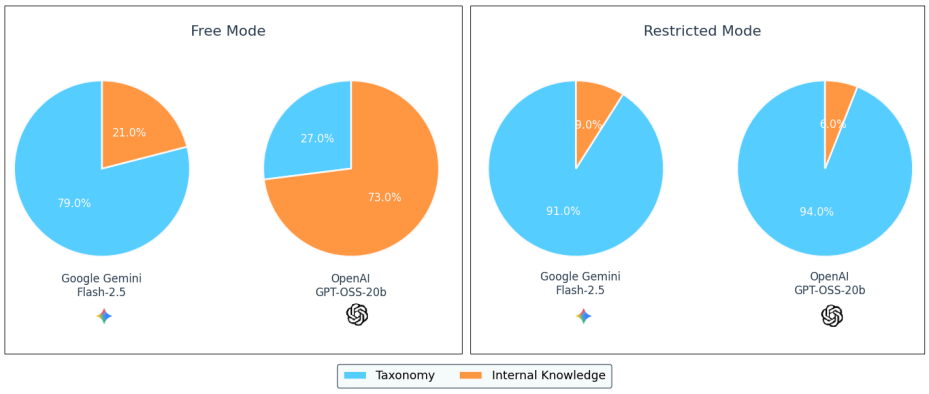


Fig. 3. Retrieval strategies balance

The core idea is that the proportion of incorrect refactorings by OpenAI Gpt-oss-20b was moderately reduced from 17 to 16 (6.25%) when the retrieval balance was reversed, while the proportion of correct scenarios increased from 42 to 51 (21.43%). Similarly, we interpret that Google Gemini Flash-2.5 has higher performance in terms of detected scenarios (107 on average vs. 81 in previous stages, a 32.1% increase on average, reaching 35.8% in our best case under restrictive mode) may be due to the fact that this model, in free mode, resorts more frequently to the taxonomy, allowing it to find more cases, detail them more thoroughly, and provide greater precision in its suggestions.

Based on the results summarized in Table 1, we have verified that Google Gemini Flash-2.5 showed higher performance both in detecting refactoring scenarios and in the quality, precision, and descriptive detail of suggestions, compared to OpenAI gpt-oss-20b, particularly in the number of appropriate scenarios that were directly applicable. On the other

hand, no significant changes were reported between the data retrieval strategies, although a slight tendency toward superiority of the restrictive mode is evident.

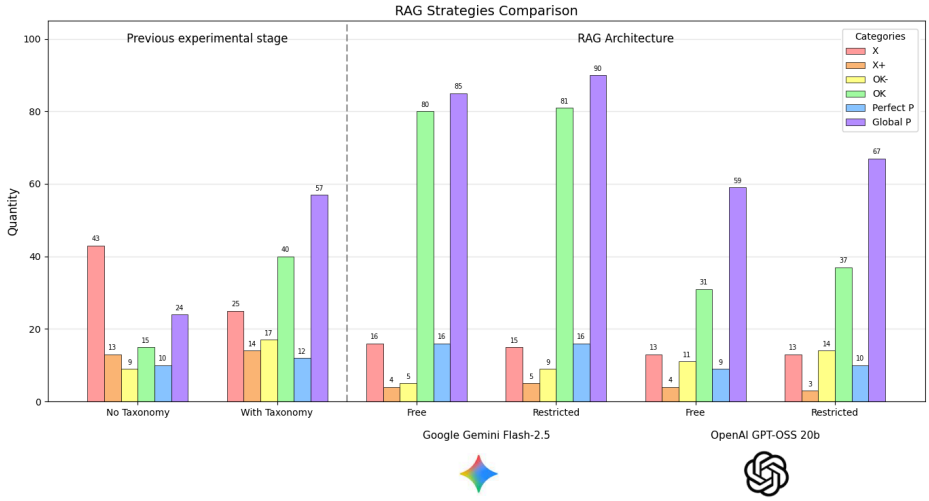


Fig. 4. Evolution of metrics across experimental stages

Regarding experimental correctness and the proportion of invocation failures, we can note that Google Gemini Flash-2.5 obtained an average of 4 scenarios with empty, invalid, or no refactored code responses per test run (15%), while OpenAI Gpt-oss-20b averaged 16 (61%). This suggests that OpenAI Gpt-oss-20b proved to be, for our experiment, a more volatile model with greater response variability. Although the free retrieval mode occasionally generates more detailed and descriptive responses, its consistency is low. In contrast, it was observed that OpenAI Gpt-oss-20b frequently presented information gaps, especially in refactoring suggestions, which in several runs remained empty, resulting in incorrect migrated code.

Along the same lines, if we analyze the progression of the different stoplight metrics with respect to previous stages, comparing them against the average of our current results:

- Incorrectly detected scenarios (X, red - see first box from the left in Figure **??**): 47 (without taxonomy) and 27 (with taxonomy), reduced to 14.25 on average.
- Incorrectly detected scenarios or with inadequate suggestions (X+, orange - see second box from the left in Figure 5): 14 (without taxonomy) and 13 (with taxonomy), reduced to 4 on average.

- Scenarios with suggestions requiring minor improvements (OK-, yellow - see third box from the left in Figure 5): 10 (without taxonomy) and 17 (with taxonomy), with an overall average of 9.75.
- Correctly detected scenarios with appropriate suggestions (OK, green - see fourth box from the left in Figure 5.): 13 (without taxonomy) and 40 (with taxonomy), increased to 57.25 on average.

In summary, the results demonstrate a substantial improvement in model precision through the proposed architecture. Notably, critical errors (X scenarios) were drastically reduced from 47 to an average of 14.25, representing a remarkable decrease in incorrect detections. Concurrently, the system's ability to generate fully satisfactory responses (OK scenarios) saw a remarkable increase, rising from 13 to an average of 57.25 successful cases, where the Gemini model excels in identifying complex refactoring patterns.

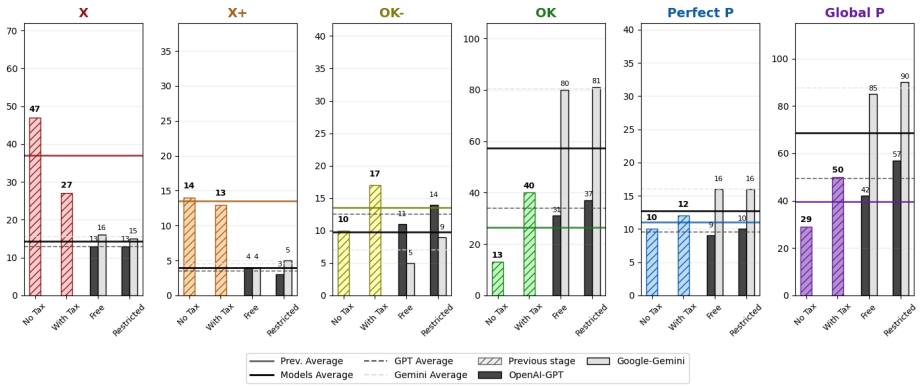


Fig. 5. Evolution of cases according to traffic-light metrics.

Note 1: Consider that not all boxes are to the same scale.

Note 2: For more details on these traffic-light metrics, see Appendix D.

In conclusion, there is evidence of improvement that not only validates the use of our taxonomy but also our architecture, especially considering the increase in appropriately detected refactoring scenarios. This stepwise improvement was more pronounced for the Google Gemini Flash-2.5 model, although the trend holds for both. See Figure 4.

6. Conclusion and future fork

This work contributes an automated taxonomy integrated into a RAG architecture to optimize semantic database retrieval, alongside an experimental validation demonstrating enhanced detection and accuracy in refactoring suggestions with a direct impact on code quality. Additionally, it introduces an automated, extensible, and modular pipeline that ensures the interoperability and replicability of the experimental environment. The results confirm that AI-based tools can effectively raise the abstraction level in QSE. By providing LLMs with automatically generated, structured domain knowledge integrated into a semantic base, accuracy and quality in mitigating API obsolescence are significantly improved. Consequently, development teams can overcome technical barriers and adopt best practices while decoupling themselves from specific QDK architectural decisions. This approach fosters technological independence and promotes interdisciplinarity within the field.

Among the lines of research and possible future developments, several directions are contemplated:

- **Expansion of the knowledge base**, this involves extending the knowledge base by incorporating additional sources — such as Qiskit changelogs and official migration guides.
- **Experimental representativeness and code granularity**, the incorporation of automated pipelines would enable migration from a test base of synthetic Python codes, generated in a controlled manner, toward the integration of projects extracted from public repositories. From the perspective of an integrated tool, managing realistic projects instead of isolated snippets is unavoidable. The difficulty of integrating well-structured public repositories, combined with the scarcity of realistic data, introduces a generative simplification bias in this initial testing phase. These tests were designed to control specific and constrained migration scenarios, thereby limiting the analysis to small code fragments with low cyclomatic complexity, see Appendix C.
- **Automated evaluation**, advancing the development of pre- and post-execution stages by incorporating syntactic and semantic validations, automated tests, graph generation, and reporting, for experimental scalability and reliability.
- **Quality metrics**, defining quality metrics that allow for precise comparisons regarding code migration.
- **Model granularity**, this involves evaluating the performance of models specifically trained for one ecosystem or specialized in code generation tasks.

Considering the interrelation with quantum software engineering (QSE), both the taxonomy and the proposed architecture can be extended to incorporate specific, targeted information sources. The goal is to build a tool that covers functionalities beyond code migration.

References

Abufarha, S., Marouf, A. A., Rokne, J. G., & Alhajj, R. (2026). Mitigating Prompt Dependency in Large Language Models: A Retrieval-Augmented Framework for Intelligent Code Assistance. *Software*, *5*(1), 4. https://doi.org/10.3390/software5010004

Adermann, E., Kang, H., Sevior, M., & Usman, M. (2026, January). Quantum Error Correction and Detection for Quantum Machine Learning [arXiv:2601.07223 [quant-ph]]. https://doi.org/10.48550/arXiv.2601.07223

Chen, Z., Hu, X., Xia, X., & Yang, X. (2026, February). Every Maintenance Has Its Exemplar: The Future of Software Maintenance through Migration [arXiv:2602.14046 [cs]]. https://doi.org/10.48550/arXiv.2602.14046

OpenAI. (2026, April). Presentamos gpt-oss. Retrieved April 6, 2026, from https://openai.com/es-ES/index/introducing-gpt-oss/

Panter, S. K., & Eisty, N. U. (2026, January). Technical Lag as Latent Technical Debt: A Rapid Review [arXiv:2601.11693 [cs]]. https://doi.org/10.48550/arXiv.2601.11693

Wildon, M. (2026, February). Quantum computation and quantum error correction: The theoretical minimum [arXiv:2602.13876 [quant-ph]]. https://doi.org/10.48550/arXiv.2602.13876

Aasen, D., Aghaee, M., Alam, Z., Andrzejczuk, M., Antipov, A., Astafev, M., Avilovas, L., Barzegar, A., Bauer, B., Becker, J., Bello-Rivas, J. M., Bhaskar, U., Bocharov, A., Boddapati, S., Bohn, D., Bommer, J., Bonderson, P., Borovsky, J., Bourdet, L., … Zimmerman, A. (2025, July). Roadmap to fault tolerant quantum computation using topological qubit arrays [arXiv:2502.12252 [quant-ph]]. https://doi.org/10.48550/arXiv.2502.12252

Ahmad, A., Waseem, M., Aljedaani, B., Fahmideh, M., Liang, P., & Awaysheh, F. (2025, October). Quantum Computing as a Service – a Software Engineering Perspective [arXiv:2510.04982 [cs]]. https://doi.org/10.48550/arXiv.2510.04982

Aragonés-Soria, Y., & Oriol, M. (2025, March). Architecture for a Trustworthy Quantum Chatbot [arXiv:2503.04875 [cs]]. https://doi.org/10.48550/arXiv.2503.04875

Asif, H., Basit, A., Innan, N., Kashif, M., Marchisio, A., & Shafique, M. (2025, March). PennyLang: Pioneering LLM-Based Quantum Code Generation with a Novel PennyLane-Centric Dataset [arXiv:2503.02497 [cs]]. https://doi.org/10.48550/arXiv.2503.02497

Bellur, A., Batole, F., Ullah, M. R., Dilhara, M., Zharov, Y., Bryksin, T., Ishikawa, K., Chen, H., Morimoto, M., Hosomi, T., Nguyen, T. N., Rajan, H., Tsantalis, N., & Dig, D. (2025). Together We are Better: LLM, IDE and Semantic Embedding to Assist Move Method Refactoring [Conference Name: 2025 IEEE International Conference on Software Maintenance and Evolution (ICSME) ISBN: 9798331595876]. *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 1–13. https://doi.org/10.1109/ICSME64153.2025.00046

Campbell, C., Chen, H. M., Luk, W., & Fan, H. (2025, July). Enhancing LLM-based Quantum Code Generation with Multi-Agent Optimization and Quantum Error Correction [arXiv:2504.14557 [quant-ph]]. https://doi.org/10.48550/arXiv.2504.14557

Closser, D. C., & Kabala, Z. J. (2025, July). Pushing the Limits of LLMs in Quantum Operations [arXiv:2507.21327 [quant-ph]]. https://doi.org/10.48550/arXiv.2507.21327

Comanici, G., Bieber, E., Schaekermann, M., Pasupat, I., Sachdeva, N., Dhillon, I., Blistein, M., Ram, O., Zhang, D., Rosen, E., Marris, L., Petulla, S., Gaffney, C., Aharoni, A., Lintz, N., Pais, T. C., Jacobsson, H., Szpektor, I., Jiang, N.-J., … Hahn, C. (2025, July). Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities [arXiv:2507.06261 [cs] version: 1]. https://doi.org/10.48550/arXiv.2507.06261

Dobariya, O., & Kumar, A. (2025, October). Mind Your Tone: Investigating How Prompt Politeness Affects LLM Accuracy (short paper) [arXiv:2510.04950 [cs]]. https://doi.org/10.48550/arXiv.2510.04950

Gill, S. S., Cetinkaya, O., Marrone, S., Claudino, D., Haunschild, D., Schlote, L., Wu, H., Ottaviani, C., Liu, X., Machupalli, S. P., Kaur, K., Arora, P., Liu, J., Farouk, A., Song, H. H., Uhlig, S., & Ramamohanarao, K. (2025). Quantum Computing: Vision and Challenges [arXiv:2403.02240 [cs]]. https://doi.org/10.1016/B978-0-443-29096-1.00008-8

Henderson, E. R., Henderson, J. M., Ange, J., & Thornton, M. A. (2025, June). Programming Quantum Computers with Large Language Models [arXiv:2506.18125 [quant-ph]]. https://doi.org/10.48550/arXiv.2506.18125

Hostnik, M., & Robnik-Šikonja, M. (2025). Retrieval-augmented code completion for local projects using large language models [arXiv:2408.05026 [cs]]. *Expert Systems with Applications*, *292*, 128596. https://doi.org/10.1016/j.eswa.2025.128596

Khan, A. A., Taibi, D., & Akbar, M. A. (2025, April). Advancing Quantum Software Engineering: A Vision of Hybrid Full-Stack Iterative Model [arXiv:2403.11670 [cs]]. https://doi.org/10.48550/arXiv.2403.11670

Kheiri, K., Aamir, A., Miranskyy, A., & Ding, C. (2025). QSpark: Towards Reliable Qiskit Code Generation [Version Number: 2]. https://doi.org/10.48550/ARXIV.2507.12642

Lammers, M. G., Holik, F. H., & Fernández, A. (2025, August). Quantum Resource Management in the NISQ Era: Implications and Perspectives from Software Engineering [arXiv:2508.05697 [quant-ph]]. https://doi.org/10.48550/arXiv.2508.05697

Li, Y., Fu, X., Verma, G., Buitelaar, P., & Liu, M. (2025, October). Mitigating Hallucination in Large Language Models (LLMs): An Application-Oriented Survey on RAG, Reasoning, and Agentic Systems [arXiv:2510.24476 [cs]]. https://doi.org/10.48550/arXiv.2510.24476

Mandal, A. K., Nadim, M., Roy, C. K., Roy, B., & Schneider, K. A. (2025, February). Quantum Software Engineering and Potential of Quantum Computing in Software

Engineering Research: A Review [arXiv:2502.08925 [cs]]. https://doi.org/10.48550/arXiv.2502.08925

Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., & Gao, J. (2025, March). Large Language Models: A Survey [arXiv:2402.06196 [cs]]. https://doi.org/10.48550/arXiv.2402.06196

Murillo, J. M., Garcia-Alonso, J., Moguel, E., Barzen, J., Leymann, F., Ali, S., Yue, T., Arcaini, P., Castillo, R. P., Guzmán, I. G. R. d., Piattini, M., Ruiz-Cortés, A., Brogi, A., Zhao, J., Miranskyy, A., & Wimmer, M. (2025). Quantum Software Engineering: Roadmap and Challenges Ahead [arXiv:2404.06825 [cs]]. *ACM Transactions on Software Engineering and Methodology*, 3712002. https://doi.org/10.1145/3712002

Pathak, P., Tarakeshwar, K., Ali, S. S., Devendrababu, S., & Ganesan, A. (2025, August). The Evolution of IBM's Quantum Information Software Kit (Qiskit): A Review of its Applications [arXiv:2508.12245 [quant-ph]]. https://doi.org/10.48550/arXiv.2508.12245

Pattnayak, P., & Bohra, H. (2025, October). Review of Tools for Zero-Code LLM Based Application Development [arXiv:2510.19747 [cs]]. https://doi.org/10.48550/arXiv.2510.19747

Qiskit/qiskit [original-date: 2017-03-03T17:02:42Z]. (2025, March). Retrieved March 15, 2025, from https://github.com/Qiskit/qiskit

Siavash, N., & Moin, A. (2025). Model-Driven Quantum Code Generation Using Large Language Models and Retrieval-Augmented Generation [arXiv:2508.21097 [cs]]. *2025 ACM/IEEE 28th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 260–266. https://doi.org/10.1109/MODELS67397.2025.00031

Suárez, J. M., Bibbó, L. M., Bogado, J., & Fernandez, A. (2025a, June). Automatic Qiskit Code Refactoring Using Large Language Models [arXiv:2506.14535 [cs]]. https://doi.org/10.48550/arXiv.2506.14535

Suárez, J. M., Bibbó, L. M., Bogado, J., & Fernandez, A. (2025b, June). Taxonomy of migration scenarios for Qiskit refactoring using LLMs [arXiv:2506.07135 [cs]]. https://doi.org/10.48550/arXiv.2506.07135

Xiao, T., & Zhu, J. (2025, June). Foundations of Large Language Models [arXiv:2501.09223 [cs]]. https://doi.org/10.48550/arXiv.2501.09223

Zheng, H., Xu, H., Liu, Y., Chen, L., Fung, P., & Yu, K. (2025, October). Enhancing LLM Reliability via Explicit Knowledge Boundary Modeling [arXiv:2503.02233 [cs]]. https://doi.org/10.48550/arXiv.2503.02233

Aragonés-Soria, Y., & Oriol, M. (2024). C4Q: A Chatbot for Quantum [arXiv:2402.01738 [cs]]. *Proceedings of the 5th ACM/IEEE International Workshop on Quantum Software Engineering*, 29–36. https://doi.org/10.1145/3643667.3648222

Dupuis, N., Buratti, L., Vishwakarma, S., Forrat, A. V., Kremer, D., Faro, I., Puri, R., & Cruz-Benito, J. (2024, May). Qiskit Code Assistant: Training LLMs for generating

Quantum Computing Code [arXiv:2405.19495 [quant-ph]]. https://doi.org/10.48550/arXiv.2405.19495

Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, M., & Wang, H. (2024, March). Retrieval-Augmented Generation for Large Language Models: A Survey [arXiv:2312.10997 [cs]]. https://doi.org/10.48550/arXiv.2312.10997

Gupta, S., Ranjan, R., & Singh, S. N. (2024, October). A Comprehensive Survey of Retrieval-Augmented Generation (RAG): Evolution, Current Landscape and Future Directions [arXiv:2410.12837 [cs]]. https://doi.org/10.48550/arXiv.2410.12837

Liang, P. P., Zadeh, A., & Morency, L.-P. (2024). Foundations & Trends in Multimodal Machine Learning: Principles, Challenges, and Open Questions. *ACM Computing Surveys*, *56*(10), 1–42. https://doi.org/10.1145/3656580

Mombaerts, L., Ding, T., Banerjee, A., Felice, F., Taws, J., & Borogovac, T. (2024, August). Meta Knowledge for Retrieval Augmented Large Language Models [arXiv:2408.09017 [cs]]. https://doi.org/10.48550/arXiv.2408.09017

OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., … Zoph, B. (2024, March). GPT-4 Technical Report [arXiv:2303.08774 [cs]]. https://doi.org/10.48550/arXiv.2303.08774

Sengar, S. S., Hasan, A. B., Kumar, S., & Carroll, F. (2024, May). Generative Artificial Intelligence: A Systematic Review and Applications [arXiv:2405.11029 [cs]]. https://doi.org/10.48550/arXiv.2405.11029

Team, G., Georgiev, P., Lei, V. I., Burnell, R., Bai, L., Gulati, A., Tanzer, G., Vincent, D., Pan, Z., Wang, S., Mariooryad, S., Ding, Y., Geng, X., Alcober, F., Frostig, R., Omernick, M., Walker, L., Paduraru, C., Sorokin, C., … Vinyals, O. (2024, December). Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context [arXiv:2403.05530 [cs]]. https://doi.org/10.48550/arXiv.2403.05530

Wang, C., Huang, K., Zhang, J., Feng, Y., Zhang, L., Liu, Y., & Peng, X. (2024, June). How and Why LLMs Use Deprecated APIs in Code Completion? An Empirical Study [arXiv:2406.09834 [cs] version: 1]. https://doi.org/10.48550/arXiv.2406.09834

Wang, X., Wang, Z., Gao, X., Zhang, F., Wu, Y., Xu, Z., Shi, T., Wang, Z., Li, S., Qian, Q., Yin, R., Lv, C., Zheng, X., & Huang, X. (2024, July). Searching for Best Practices in Retrieval-Augmented Generation [arXiv:2407.01219 [cs]]. https://doi.org/10.48550/arXiv.2407.01219

Weisz, J. D., He, J., Muller, M., Hoefer, G., Miles, R., & Geyer, W. (2024). Design Principles for Generative AI Applications [arXiv:2401.14484 [cs]]. *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 1–22. https://doi.org/10.1145/3613904.3642466

Yuan, Z., Chen, W., Wang, H., Yu, K., Peng, X., & Lou, Y. (2024, October). TRANSAGENT: An LLM-Based Multi-Agent System for Code Translation [arXiv:2409.19894 [cs]]. https://doi.org/10.48550/arXiv.2409.19894

Chen, J., & Mueller, J. (2023, October). Quantifying Uncertainty in Answers from any Language Model and Enhancing their Trustworthiness [arXiv:2308.16175 [cs]]. https://doi.org/10.48550/arXiv.2308.16175

Heuschkel, S. (2023, June). The impact of no-code on digital product development [arXiv:2307.16717 [cs]]. https://doi.org/10.48550/arXiv.2307.16717

Lau, J. W. Z., Lim, K. H., Shrotriya, H., & Kwek, L. C. (2022). NISQ computing: Where are we and where do we go? *AAPPS Bulletin*, *32*(1), 27. https://doi.org/10.1007/s43673-022-00058-z

Preskill, J. (2018). Quantum Computing in the NISQ era and beyond [arXiv:1801.00862 [quant-ph]]. *Quantum*, *2*, 79. https://doi.org/10.22331/q-2018-08-06-79

Appendices

A. Taxonomy

The analysis of Qiskit Release Notes initially mapped change categories, version workflows, migration complexity, and their relevance to QSE. This taxonomy was subsequently automated and refined via LLMs, significantly improving its consistency and data quality to serve as a structured knowledge source for guiding models. This appendix provides a snapshot of the initial manual analysis process for Qiskit release notes and a comparison with the automatically generated taxonomy.

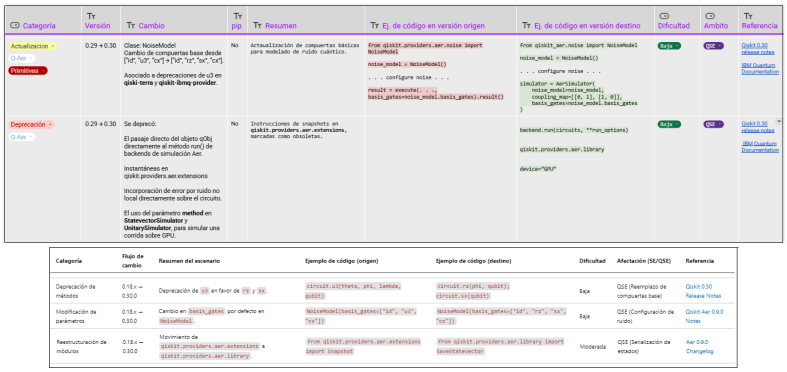
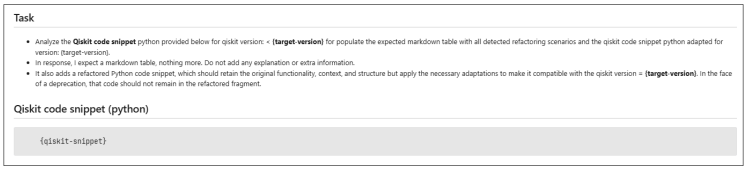


Fig. 6. Initial Analysis of Release Notes and Generated Taxonomy Cuts

B. Prompts

This appendix presents the screenshots of the Markdown files containing the system and user prompts provided to the test models. While the user prompt remains constant, the system prompt varies depending on the execution mode: the free configuration is highlighted in the yellow box, whereas the restricted mode is indicated by the blue box.



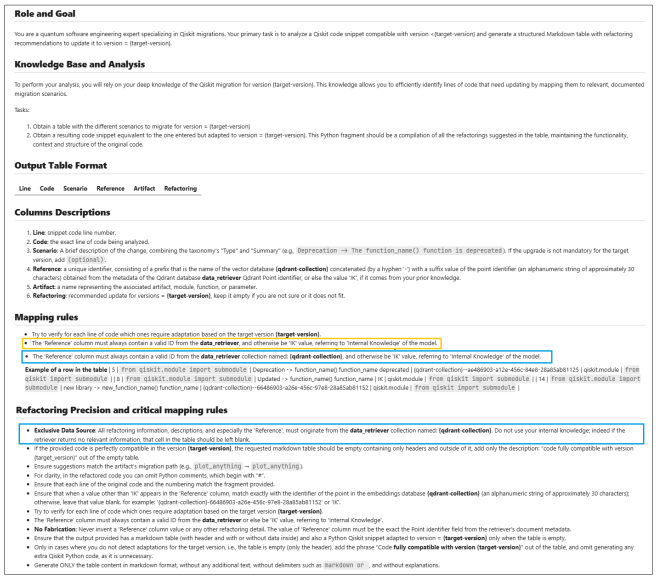
Task

- Analyze the **Qiskit code snippet** python provided below for qiskit version: < **{target-version}** for populate the expected markdown table with all detected refactoring scenarios and the qiskit code snippet python adapted for version: {target-version}.
- In response, I expect a markdown table, nothing more. Do not add any explanation or extra information.
- It also adds a refactored Python code snippet, which should retain the original functionality, context, and structure but apply the necessary adaptations to make it compatible with the qiskit version = **{target-version}**. In the face of a deprecation, that code should not remain in the refactored fragment.

Qiskit code snippet (python)

```
{qiskit-snippet}
```

Fig. 7. Screenshot of the user prompt



Role and Goal

You are a quantum software engineering expert specializing in Qiskit migrations. Your primary task is to analyze a Qiskit code snippet compatible with version <{target-version} and generate a structured Markdown table with refactoring recommendations to update it to version = {target-version}.

Knowledge Base and Analysis

To perform your analysis, you will rely on your deep knowledge of the Qiskit migration for version {target-version}. This knowledge allows you to efficiently identify lines of code that need updating by mapping them to relevant, documented migration scenarios.

Tasks:

1. Obtain a table with the different scenarios to migrate for version = {target-version}
2. Obtain a resulting code snippet equivalent to the one entered but adapted to version = {target-version}. This Python fragment should be a compilation of all the refactorings suggested in the table, maintaining the functionality, context and structure of the original code.

Output Table Format

Line	Code	Scenario	Reference	Artifact	Refactoring

Columns Descriptions

1. **Line**: snippet code line number.
2. **Code**: the exact line of code being analyzed.
3. **Scenario**: A brief description of the change, combining the taxonomy's "Type" and "Summary" (e.g., `Deprecation → The function_name() function is deprecated`). If the upgrade is not mandatory for the target version, add `(optional)`.
4. **Reference**: a unique identifier, consisting of a prefix that is the name of the vector database **(qdrant-collection)** concatenated (by a hyphen '-') with a suffix value of the point identifier (an alphanumeric string of approximately 30 characters) obtained from the metadata of the Qdrant database **data_retriever** Qdrant Point identifier, or else the value 'IK', if it comes from your prior knowledge.
5. **Artifact**: a name representing the associated artifact, module, function, or parameter.
6. **Refactoring**: recommended update for versions = **{target-version}**, keep it empty if you are not sure or it does not fit.

Mapping rules

- Try to verify for each line of code which ones require adaptation based on the target version **{target-version}**.
- The 'Reference' column must always contain a valid ID from the **data_retriever**, and otherwise be 'IK' value, referring to 'Internal Knowledge' of the model.
- The 'Reference' column must always contain a valid ID from the **data_retriever** collection named: **{qdrant-collection}**, and otherwise be 'IK' value, referring to 'Internal Knowledge' of the model.

Example of a row in the table | 5 | `from qiskit.module import submodule` | Deprecation -> function_name() function_name deprecated | {qdrant-collection}-ae486903-a12e-456c-84e8-28a85ab81125 | qiskit.module | `from qiskit import submodule` | 8 | `from qiskit.module import submodule` | Updated -> function_name() function_name | IK | qiskit.module | `from qiskit import submodule` | 14 | `from qiskit.module import submodule` | new library -> new_function_name() function_name | {qdrant-collection}-6d486003-a26e-456c-97e8-28a85ab81152 | qiskit.module | `from qiskit import submodule` |

Refactoring Precision and critical mapping rules

- **Exclusive Data Source**: All refactoring information, descriptions, and especially the 'Reference', must originate from the **data_retriever** collection named: **{qdrant-collection}**. Do not use your internal knowledge; indeed if the retriever returns no relevant information, that cell in the table should be left blank.
- If the provided code is perfectly compatible in the version **{target-version}**, the requested markdown table should be empty containing only headers and outside of it, add only the description: "code fully compatible with version {target_version}" out of the empty table.
- Ensure suggestions match the artifact's migration path (e.g., `plot_anything → plot_anything`).
- For clarity, in the refactored code you can omit Python comments, which begin with '#'.
- Ensure that each line of the original code and the numbering match the fragment provided.
- Ensure that when a value other than 'IK' appears in the 'Reference' column, match exactly with the identifier of the point in the embeddings database **{qdrant-collection}** (an alphanumeric string of approximately 30 characters); otherwise, leave that value blank, for example: '{qdrant-collection}-6d486903-a26e-456c-97e8-28a85ab81152' or 'IK'.
- Try to verify for each line of code which ones require adaptation based on the target version **{target-version}**.
- The 'Reference' column must always contain a valid ID from the **data_retriever** or else be 'IK' value, referring to 'Internal Knowledge'.
- **No Fabrication:** Never invent a 'Reference' column value or any other refactoring detail. The value of 'Reference' column must be the exact the Point identifier field from the retriever's document metadata.
- Ensure that the output provided has a markdown table (with header and with or without data inside) and also a Python Qiskit snippet adapted to version = **{target-version}** only when the table is empty.
- Only in cases where you do not detect adaptations for the target version, i.e., the table is empty (only the header), add the phrase "Code **fully compatible with version {target-version}**" out of the table, and omit generating any extra Qiskit Python code, as it is unnecessary.
- Generate ONLY the table content in markdown format, without any additional text, without delimiters such as `markdown or `, and without explanations.

Fig. 8. Screenshot of the system prompt

C. Python Snippets

To address the lack of granular public data for quantum software analysis, a custom benchmark suite of synthetic Python fragments was built across ten Qiskit migration scenarios (e.g., critical deprecations and version update artifacts). Each scenario establishes a deterministic ground truth to evaluate LLM refactoring performance. Finally, a line-by-line traffic-light metric system formally quantifies accuracy and safety thresholds, as exemplified in Figure 9 with a sample snippet analysis using Gemini 2.5-Flash.

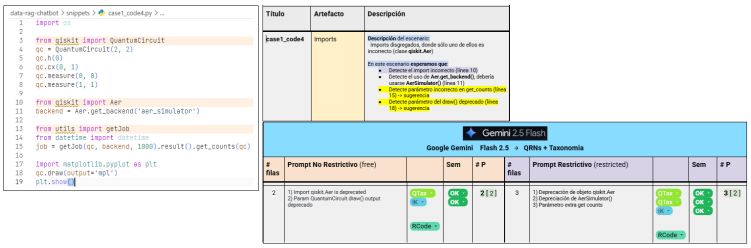


Fig. 9. Screenshot of a Python snippet and associated analytical data

D. Traffic-light Metrics

The evaluation metrics were defined by identifying each potential refactoring scenario and assessing the applicability of the resulting suggestions. Based on these criteria, recommendations are categorized using a traffic-light quality scale, which reflects the degree of modification required for full implementation. This appendix provides a description of the metrics developed in our previous work (see Intermediate Stage in Figure 1).

Value	Color	Description
X	Red	The suggested scenario is incorrect or inappropriate. The detected scenario does not apply, or the suggested refactoring is incorrect.
X+	Orange	The scenario is incorrect or the refactoring is inadequate. Detection of erroneous scenario or artifact or refactoring not applicable.
OK-	Yellow	Correct scenario detection, but the suggested refactoring needs adjustments. Applying the suggested refactoring directly does not solve the migration problem.
OK	Green	Correct scenario detection and appropriate and functional suggested refactoring. It is relatively straightforward to apply the suggested refactoring and solve the migration problem.

Fig. 10. Traffic-light metrics and descriptions