UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Aylton Bernardino de Almeida Junior

On the Use of Large Language Models for Software Library Migration

Belo Horizonte
2026

Aylton Bernardino de Almeida Junior

On the Use of Large Language Models for Software Library Migration

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Marco Túlio de Oliveira Valente
Co-Advisor: José Laerte Pires Xavier Júnior

Belo Horizonte
2026

2026, Aylton Bernardino de Almeida Junior.
Todos os direitos reservados

Almeida Junior, Aylton Bernardino de.

A447o On the use of Large Language Models for software library migration [recurso eletrônico] / Aylton Bernardino de Almeida Junior – 2026.
1 recurso online (91 f. il., color.) : pdf.

Orientador: Marco Túlio de Oliveira Valente
Coorientador: José Laerte Pires Xavier Júnior.

Dissertação (Mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f. 84-89.

1. Computação – Teses. 2. Engenharia de software - Teses. 3. Python (Linguagem de programação de computador) – Teses. 4. SQL (Linguagem de programação de computador) – Teses. 5. Inteligência artificial – Processamento de dados – Teses. 6. Sistemas de consultas e respostas – Teses. I. Valente, Marco Túlio de Oliveira. II. Xavier Júnior, José Laerte Pires. III. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. IV. Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Irénquer Vismeg Lucas Cruz
CRB 6/819 - Universidade Federal de Minas Gerais – ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS

ON THE USE OF LARGE LANGUAGE MODELS FOR SOFTWARE LIBRARY MIGRATION

AYLTON BERNARDINO DE ALMEIDA JUNIOR

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. Marco Túlio de Oliveira Valente - Orientador
Departamento de Ciência da Computação - UFMG

Prof. José Laerte Pires Xavier Júnior - Coorientador
Instituto de Ciências Exatas e Informática - PUC Minas

Prof. Guilherme Amaral Avelino
Departamento de Computação - UFPI

Prof. Eduardo Magno Lages Figueiredo
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 10 de abril de 2026

Documento assinado eletronicamente por **Marco Tulio de Oliveira Valente**, **Professor do Magistério Superior**, em 16/04/2026, às 10:43, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Eduardo Magno Lages Figueiredo**, **Professor do Magistério Superior**, em 16/04/2026, às 11:27, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **José Laerte Pires Xavier Júnior**, **Usuário Externo**, em 23/04/2026, às 10:03, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Guilherme Amaral Avelino**, **Usuário Externo**, em 06/05/2026, às 16:09, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufmg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **5095717** e o código CRC **475071C4**.

Referência: Processo nº 23072.220407/2026-51 SEI nº 5095717

Acknowledgments

A conclusão desta dissertação representa um marco na minha trajetória acadêmica e pessoal. Entretanto, este trabalho não foi realizado de forma solitária, mas contou com o apoio de muitas pessoas. Por isso, deixo aqui o meu mais sincero agradecimento a todos que contribuíram para essa conquista. Em especial, agradeço:

Aos meus pais, Aylton e Jacqueline, e meu irmão Bernardo, vocês me deram oportunidades, me ensinaram a importância do estudo e me incentivaram durante todo esse percurso. Sem vocês, eu não teria chegado tão longe; espero que saibam como sou grato por ter vocês como família.

À minha namorada, Brenda, que durante todo esse tempo me ajudou a ser paciente e a enfrentar as dificuldades encontradas neste processo. Você me ajudou a celebrar vitórias como passar em uma prova difícil ou ter um *paper* aceito. Sou muito grato pelo seu companheirismo e espero sempre conseguir fazer o mesmo por você.

Aos meus amigos de longa data, que, mesmo sem conseguirmos nos encontrar com a frequência que costumávamos, tornaram cada reunião, aniversário ou celebração especial da forma que só vocês sabem fazer.

Ao meu orientador, Prof. Dr. Marco Tulio Valente, e ao meu coorientador, Prof. Dr. Laerte Xavier, pela atenção e paciência durante toda essa jornada. Nossas discussões e suas orientações foram essenciais para aprimorar ideias e estruturar as tarefas que possibilitaram a finalização desta dissertação.

Aos professores da PUC Minas e membros da ASERG, agradeço todo o apoio na minha jornada desde o curso de Engenharia de Software.

Aos membros da banca, Prof. Dr. Guilherme Avelino e Prof. Dr. Eduardo Figueiredo, pela disponibilidade e críticas que possibilitam a finalização deste trabalho.

Ao DCC/UFMG e ao PPGCC, pelo auxílio financeiro, logístico e profissional.

"*All we have to decide is what to do with the time that is given us*"
(J.R.R Tolkien)

Resumo

A manutenção de software e a atualização de dependências evitam a obsolescência de sistemas. Quando bibliotecas introduzem breaking changes, desenvolvedores enfrentam refatorações custosas e propensas a erros. Em linguagens dinâmicas como Python, ferramentas tradicionais baseadas em Árvores de Sintaxe Abstrata (AST) falham devido à flexibilidade da linguagem em tempo de execução. Nesse cenário, os Large Language Models (LLMs) surgem como uma alternativa promissora, mas sua eficácia em migrações complexas demanda validação empírica. Nesta dissertação, investigamos o uso de LLMs na migração de bibliotecas em Python, utilizando as transições para a versão 2 do SQLAlchemy e Pydantic como objeto de estudo. Introduzimos a métrica de Migration Coverage para quantificar a proporção de código legado efetivamente transformado e utilizamos a porcentagem de testes que passam para avaliar o funcionamento da aplicação após a migração. A pesquisa avaliou duas abordagens: uma estática (baseada em Prompts) e uma autônoma (GitHub Copilot Agent Mode). O primeiro estudo revelou que a abordagem orientada por prompts (Zero-Shot, One-Shot e Chain-of-Thoughts) atingiu altas taxas de Migration Coverage. Contudo, apresentou pouco percentual de testes passando. Em contrapartida, o segundo estudo demonstrou que o uso do modo Agente muda esse cenário. Ao interagir com o ambiente, executar testes e analisar erros, o agente manteve alta Migration Coverage (medianas de 100% e 60% para cada biblioteca) e manteve o funcionamento. No SQLAlchemy, a mediana de testes foi de quase 40%, com 5 repositórios alcançando uma migração quase perfeita (100% de cobertura e mais de 80% de testes passando). Conclui-se que o uso de LLMs para migração de bibliotecas é viável, porém é necessário um desenvolvedor que possa auxiliar com ajustes menores e garantir o funcionamento do sistema. Além disso, é relevante mencionar que um código padronizado, bem documentado e com uma suíte de teste robusta pode auxiliar o modo Agente da LLM, possibilitando melhores resultados.

Palavras-chave: migração de API; large language models; ChatGPT; Python; SQLAlchemy; Pydantic; Copilot; Visual Studio Code; LLM; agentes LLM; engenharia de prompts.

Abstract

Software maintenance and dependency update avoid system obsolescence. When libraries introduce breaking changes, developers face costly and error-prone refactorings. In dynamic languages like Python, traditional tools based on Abstract Syntax Trees (AST) fail due to the language's flexibility at runtime. In this scenario, Large Language Models (LLMs) emerge as a promising alternative, but their effectiveness in complex migrations requires empirical validation. In this dissertation, we investigate the use of LLMs for Python library migration, using the transitions to version 2 of SQLAlchemy and Pydantic as subjects of study. We introduce the Migration Coverage metric to quantify the proportion of legacy code effectively transformed, and we use the percentage of passing tests to evaluate the application's functionality after migration. The research evaluated two approaches: a static one (Prompt-based) and an autonomous one (Agent-based). The first study revealed that the prompt-oriented approach (Zero-Shot, One-Shot, and Chain-of-Thoughts) achieved high Migration Coverage rates. However, it yielded a low percentage of passing tests. In contrast, the second study demonstrated that using the Agent mode changes this scenario. By interacting with the environment, executing tests, and analyzing errors, the agent maintained high Migration Coverage (medians of 100% and 60% for each library) and preserved functionality. For SQLAlchemy, the median of passing tests was nearly 40%, with 5 repositories achieving an almost perfect migration (100% coverage and over 80% passing tests). We conclude that using LLMs for library migration is viable; however, a developer is required to assist with minor adjustments and ensure system functionality. Furthermore, it is relevant to note that standardized, well-documented code with a robust test suite can assist the LLM's Agent mode, enabling better results.

Keywords: API migration; large language models; ChatGPT; Python; SQLAlchemy; Pydantic; Copilot; Visual Studio Code; LLM; LLM agent; prompt engineering

List of Figures

2.1 Example of API breaking change . . . 20
2.2 Limitation of traditional migration approaches in the `bump-pydantic` repository. 22
2.3 Synchronous (a) and asynchronous (b) code using the `asyncio` module . . . 23
2.4 Python code without (a) and with (b) type hints . . . 24
2.5 SQLAlchemy migration from version 1 (a) to version 2 (b) . . . 25
2.6 Comparison between Pydantic V1 (a) and V2 (b) class declarations. . . . 26
2.7 Comparison of prompting strategies applied to a hypothetical migration task. 29

4.1 Base command used for all prompts . . . 49
4.2 One-Shot prompt . . . 50
4.3 Chain-of-Thoughts prompt . . . 51

5.1 Prompt used to generate the Copilot Instructions file . . . 64
5.2 Prompt to migrate the repositories . . . 65
5.3 Copilot's generated TODOs list . . . 66
5.4 Agent reasoning on the `casbin` issue with `asyncio` for the `sqlalchemy-adapter` repository. . . . 67
5.5 Agent asking the user to install the `poetry` library in the `lnurl` repository. . . . 70

List of Tables

3.1 Summary of repository selection and filtering process 41
3.2 Characterization of the final dataset repositories as of 2026 42
3.3 Migration Coverage Example 43

4.1 SQLAlchemy Prompt-Based Migration Results 52
4.2 Pydantic Prompt-Based Migration Results 57

5.1 SQLAlchemy Agent-Based Migration Results 67
5.2 Pydantic Agent-Based Migration Results 69

A.1 Migration Coverage per repository and migration strategy. 90
A.2 Passing Tests per repository and migration strategy. 91

Contents

1 Introduction **14**

1.1 Motivation . . . 14
1.2 Proposed Work . . . 15
1.3 Contributions . . . 15
1.4 Publications . . . 16
1.5 Outline of the Dissertation . . . 17

2 Background and Related Work **18**

2.1 Application Programming Interfaces . . . 18
2.1.1 API Evolution and Compatibility . . . 19
2.1.2 API Migration . . . 20
2.2 Modern Python Development and Target Libraries . . . 21
2.2.1 Asynchronous I/O in Python . . . 22
2.2.2 Static Typing in Python . . . 24
2.2.3 Target Library: SQLAlchemy . . . 25
2.2.4 Target Library: Pydantic . . . 26
2.3 Large Language Models . . . 27
2.3.1 Foundations and Adoption . . . 27
2.3.2 Prompt Engineering . . . 28
2.3.3 Applications in Software Engineering . . . 29
2.3.4 Chatbos vs. Agentic Systems . . . 30
2.4 Evaluation of Automated API Migration . . . 31
2.4.1 Functional Correctness . . . 32
2.4.2 Static Analysis and Typing Metrics . . . 32
2.4.3 Evaluating Migration Completeness . . . 33
2.5 Related Work . . . 33
2.5.1 API Migration in Software Engineering . . . 34
2.5.2 Large Language Models in Software Engineering . . . 35
2.5.3 Code Migration using Large Language Models . . . 36
2.6 Final Remarks . . . 37

3 Study Design **39**

3.1 Target APIs . . . 39

3.2 Dataset Creation . . . 40
3.3 Evaluation Metrics . . . 43
3.3.1 Migration Coverage . . . 43
3.3.2 Percentage of Passing Tests . . . 44
3.3.3 Application Compiles . . . 44
3.3.4 Quality Metrics . . . 45
3.4 Threats to Validity . . . 46
3.5 Final Remarks . . . 47

4 Prompt-Based Migration **48**
4.1 Methodology . . . 48
4.2 Results . . . 51
4.2.1 SQLAlchemy Results . . . 52
4.2.2 Pydantic Results . . . 57
4.3 Threats to Validity . . . 61
4.4 Final Remarks . . . 62

5 Agent-Based Migration **63**
5.1 Methodology . . . 63
5.2 Results . . . 66
5.2.1 SQLAlchemy Results . . . 66
5.2.2 Pydantic Results . . . 69
5.3 Threats to Validity . . . 72
5.4 Final Remarks . . . 73

6 Conclusion **75**
6.1 Overview . . . 75
6.2 Summary of Findings . . . 76
6.2.1 Prompt-Based Strategy . . . 76
6.2.2 Agent-Based Strategy . . . 77
6.3 Comparative Analysis . . . 78
6.3.1 Functional Correctness and Autonomy . . . 78
6.3.2 The Impact of Breaking Change Complexity . . . 79
6.3.3 Environmental vs. Contextual Limitations . . . 80
6.4 Practical Guidelines for LLM-Assisted Migration . . . 80
6.4.1 Stabilizing the Execution Environment . . . 81
6.4.2 Ensuring Test Suite Robustness . . . 81
6.4.3 Choosing the Right strategy . . . 82
6.5 Future Work . . . 82

References 84

Appendix A Detailed Results by Repository 90

Chapter 1

Introduction

1.1 Motivation

Software maintenance remains a critical concern in Software Engineering. As systems evolve, the need to keep applications up to date is essential to prevent them from becoming rigid legacy systems that are costly and difficult to maintain. A significant portion of this effort is driven by the continuous evolution of third-party libraries and frameworks, which frequently introduce new features, performance improvements, and security patches [5, 15, 19, 37]. However, this evolution often comes at the cost of introducing breaking changes. Libraries frequently modify their public interface by renaming core methods, altering function signatures, or removing deprecated features. Consequently, developers are forced to refactor portions of their codebase to maintain compatibility.

In statically typed ecosystems, automated refactoring tools based on Abstract Syntax Trees (AST) have proven effective for such tasks [11, 50]. However, in dynamic languages like Python, these traditional tools often fall short due to the language's flexibility and runtime behavior [38, 54]. Consequently, developers are frequently left with manual refactoring as their only option, a process that is time-consuming and error-prone. This gap creates the need for more intelligent automation solutions capable of understanding the semantic context of code changes rather than just their syntactic structure.

Recently, Large Language Models (LLMs) have emerged as a promising avenue for addressing this challenge. With their ability to reason and generate code based on natural language instructions, LLMs offer a potential alternative to rule-based migration tools [16, 25, 57]. However, despite their popularity, there is limited empirical evidence regarding their effectiveness in handling complex, real-world library migrations. It remains unclear whether simple prompting strategies are sufficient for these tasks or if more autonomous, agentic workflows are required to ensure functional correctness.

1.2 Proposed Work

This dissertation proposes an empirical study investigating the effectiveness of Large Language Models (LLMs), specifically GPT-4o (for prompt-based migration) and GitHub Copilot with GPT-4o (for agent-based migration), in automating Python library migrations across major versions. To capture a diverse range of challenges, we focus on the transition of two widely adopted libraries: SQLAlchemy (migrating from version 1.4 to 2.0) and Pydantic (migrating from version 1 to 2). These libraries were selected because they represent distinct migration scenarios: SQLAlchemy involves a paradigm change toward asynchronous execution, while Pydantic imposes many breaking changes to its API in order to improve its performance.

To empirically assess current LLM technology in this context, we designed two distinct studies. The first study explores a Prompt-Based approach, assessing how different levels of context—Zero-Shot, One-Shot, and Chain-of-Thoughts—affect the model's ability to correctly migrate deprecated code. This study aims to establish a baseline for what can be achieved through static generation without execution feedback.

The second study investigates an Agent-Based approach, evaluating the autonomous capabilities of GitHub Copilot's Agent Mode. Unlike static prompting, this approach allows the LLM to interact with the local development environment, execute tests, read error logs, and iteratively refine its output. By contrasting these two strategies, this work aims to identify the trade-offs between the amount of code migrated and whether the code retains its functionality after the migration.

It is worth noting that during the agent-based study, the model occasionally attempted to resolve dependency conflicts (e.g., with `casbin` and `poetry` configurations) that were not part of the migration target, but whose incompatibility with the new library version was identified by the agent as a blocker. These cases are discussed in detail in Chapter 5.

1.3 Contributions

The main contributions of this dissertation to the field of Software Engineering are as follows.

- **An empirical comparison of Prompting Strategies:** We provide a detailed analysis of Zero-Shot, One-Shot, and Chain-of-Thoughts prompts for code migra-

tion, demonstrating that providing a single relevant example (One-Shot) is often more effective for migration tasks than complex reasoning chains.

- **An evaluation of Agentic Workflows in Maintenance:** We show that while autonomous agents significantly improve functional correctness by leveraging test feedback (achieving up to 100% passing tests in specific cases), they are limited by environmental complexities, such as dependency management and configuration loops.
- **The proposal of Migration Coverage as a metric:** We introduce Migration Coverage, a metric designed to quantify the amount of code migrated to the newer library version. By measuring the proportion of deprecated usages successfully replaced or removed, this metric allows researchers to evaluate the model's ability to identify and transform code excerpts, distinguishing between the effort to migrate and the ability to preserve behavior.
- **Practical Guidelines for LLM-Assisted Migration:** Based on our findings, we propose a set of actionable guidelines for developers, emphasizing the need for standardized execution environments and robust test suites as requirements for deploying autonomous coding agents.

1.4 Publications

The research conducted in this dissertation resulted in the following publications:

- Aylton Almeida, Laerte Xavier, and Marco Tulio Valente. Automatic Library Migration Using Large Language Models: First Results. *18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Emerging Results, Vision and Reflection Track*. pages 427–433, 2024. https://doi.org/10.1145/3674805.3690746
- Aylton Almeida, Laerte Xavier, Marco Tulio Valente. Using Copilot Agent Mode to Automate Library Migration: A Quantitative Assessment. *1st International Workshop on Agentic Engineering (AGENT). Co-located with 48th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 1–5, 2026. https://doi.org/10.48550/arXiv.2510.26699

1.5 Outline of the Dissertation

The remainder of this dissertation is organized as follows.

- **Chapter 2** provides the theoretical background necessary to understand the challenges of software migration and the architecture of Large Language Models. It also reviews related work in automated refactoring and code generation.
- **Chapter 3** details the methodology used in this research, describing the selection process for the dataset of open-source repositories, the definition of the target libraries (SQLAlchemy and Pydantic), and the metrics established for evaluation.
- **Chapter 4** presents the first empirical study, focusing on the Prompt-Based strategy. It analyzes the effectiveness of Zero-Shot, One-Shot, and Chain-of-Thoughts techniques in migrating the selected libraries.
- **Chapter 5** describes the second study, which evaluates the Agent-Based strategy. It explores the impact of autonomous interaction with the development environment on migration coverage and functional correctness.
- **Chapter 6** concludes the dissertation by synthesizing the findings from both studies and presenting a comparative analysis of the strategies. It provides practical guidelines for LLM-assisted migration, focusing on environmental standardization and testing strategies, before outlining potential directions for future research.

Chapter 2

Background and Related Work

This chapter presents the theoretical foundation and context related to this dissertation. In Section 2.1, we discuss the concept of Application Programming Interfaces (APIs). Section 2.2 contextualizes these challenges within the Python ecosystem and our target libraries. Section 2.3 introduces Large Language Models (LLMs). Section 2.4 addresses the complexity of evaluating automated code migration. Finally, Section 2.5 reviews related work and Section 2.6 presents final remarks.

2.1 Application Programming Interfaces

Modern software development relies on the reuse of third-party libraries and frameworks [37]. These components are typically accessed through Application Programming Interfaces (APIs), which define a contract for interaction between different software components. By leveraging APIs, developers can integrate complex functionalities, such as database management, network communication, or data validation, avoiding their complete implementation. This practice can reduce development time and the number of errors in an application. However, the widespread adoption of APIs introduces a critical dependency management challenge: as these libraries evolve, client applications must adapt to maintain compatibility and security [18].

The remainder of this section explores the challenges associated with this evolution. Section 2.1.1 discusses API evolution. In Section 2.1.2, we present strategies used for API migration.

2.1.1 API Evolution and Compatibility

APIs are not static artifacts: they evolve continuously to meet new requirements and improve software quality. This need for change is driven by various factors, including the addition of new features, performance optimizations, improving of internal design, and the correction of bugs or security vulnerabilities [15, 19]. Although evolution is necessary for the health of a library, it poses risks to client systems. Changes in APIs are generally categorized into two types: Non-breaking Changes and Breaking Changes. Non-breaking Changes typically involve adding new API elements without compromising backward-compatibility. This allows client applications to function without modification. In contrast, Breaking Changes alter the existing contract, causing client applications to fail during compilation or execution [4].

Breaking Changes can be categorized based on the type of modification applied to the API elements. According to the taxonomy established by Dig and Johnson [11], these changes primarily fall into four categories:

1. **Renaming:** The identifier of a class, method, or field changes;
2. **Moving:** Involves the relocation of elements between packages or modules;
3. **Signature Changes:** Includes adding, removing, or reordering method parameters, as well as altering return types;
4. **Visibility Changes:** Restricting access to previously public members.

In the context of this work, we also consider the introduction of asynchronous definitions (`async/await`) as a distinct category of breaking change, as it requires a structural propagation affecting both the function signature and its invocation sites.

To illustrate these concepts, Figure 2.1 presents two versions of a simple utility library designed to calculate product discounts. In version 1.0 (Figure 2.1a), the function `calculate_discount` accepts a single argument: the price. The discount rate is hard-coded internally. In version 2.0 (Figure 2.1b), the developers changed the function signature such that the discount rate needs to be passed as a second argument. This modification constitutes a breaking change because the function signature has been altered. Client code developed for version 1.0, which invokes the function with only one argument (Line 6), will fail immediately upon updating to version 2.0 due to a missing parameter error.

```python
# Library Definition (V1)
def calculate_discount(price):
    return price * 0.1

# Client Usage
final = calculate_discount(100)
# Result: 10.0
```

(a)

```python
# Library Definition (V2)
def calculate_discount(price,
    rate):
     return price * rate

# Client Usage (Unchanged)
final = calculate_discount(100)
# Error: missing argument 'rate'
```

(b)

Figure 2.1: Example of API breaking change

2.1.2 API Migration

API migration defines the process of adapting a client application to work either with a different library or a newer version of a library or framework already in use [19]. Keeping dependencies up to date is a critical maintenance activity [10]. Failure to migrate can lead to the accumulation of technical debt, turning applications into rigid legacy systems that are costly to maintain. Furthermore, outdated libraries often expose systems to unpatched security vulnerabilities and compatibility issues [18].

Despite its importance, API migration remains a predominantly manual, time-consuming, and error-prone task. Developers often struggle to keep up with frequent updates, leading to either a delay in adoption or simply ignoring updates due to the complexity involved in applying them [14, 49]. The manual migration process typically follows a general workflow [19]:

1. **Mapping/Discovery:** Developers must identify which parts of their codebase rely on the deprecated or changed API elements.
2. **Usage Analysis:** They must consult documentation, changelogs, or migration guides to understand the semantics of the new API version. This step is often hindered by incomplete or ambiguous documentation [9].
3. **Transformation:** The code must be rewritten to accommodate the changes. This often goes beyond simple syntax replacement, requiring structural changes to support new paradigms (e.g., transition from global functions to class methods).
4. **Correctness Checking:** Finally, the migrated code must be tested to ensure that the application's behavior is preserved and that no regressions were introduced.

Given the complexity and repetitive nature of this workflow, relying solely on manual strategies is often inefficient and error-prone [23]. Developers typically seek automated solutions to reduce the effort of refactoring, minimize human error, and ensure

consistency across the codebase. Initially, the first tasks related to automated code migration relied on rule-based transformations or the manipulation of the Abstract Syntax Tree (AST) [1]. Pioneered by Opdyke [27], these techniques enable tools to parse code into a structured format, apply specific transformation rules, and regenerate the source text. In the Python ecosystem, an initial use for this strategy was in the context of programming language modernization. The release of Python 3 represented a considerable change in the language, and tools such as `2to3`[1] were widely used to automate the transition from Python V2 to V3. More recently, library maintainers have released specialized tools, such as `bump-pydantic`[2], which use similar AST patterns to automate the migration between Pydantic library versions. This library is responsible for allowing usage of static typing validation during runtime in Python code.

While effective for standard patterns, these deterministic tools often suffer from syntactic rigidity, failing to recognize valid code variations that deviate slightly from expected rules. Figure 2.2 contains a concrete example of this limitation. In it, we can see Issue #134 [3], from the official `bump-pydantic` repository. The issue describes a problem in which the tool relies on specific AST structures to identify decorators but fails to detect the `@root_validator` usage when it is applied without parentheses (e.g. `@root_validator` vs. `@root_validator()`). Although both forms are valid and semantically equivalent in Python, the rule-based engine ignores the former, leaving the code unmigrated.

This limitation exemplifies why purely syntactic approaches are insufficient for comprehensive migration. Maintaining rules for every possible syntactic variation imposes a burden on tool developers [57], and often, robust migration requires understanding the code's intent rather than just its structure [8].

2.2 Modern Python Development and Target Libraries

While API evolution is a general phenomenon, its impact is particularly frequent in dynamic ecosystems like Python. Over the past decade, the language has undergone significant changes in its development paradigms, transitioning from simple, synchronous scripting to the construction of robust, statically-typed, and asynchronous distributed systems [7]. These updates have forced major libraries to rewrite their code, often triggering breaking changes for downstream users. In this dissertation we use two libraries

[1] https://docs.python.org/3.9/library/2to3.html
[2] https://github.com/pydantic/bump-pydantic
[3] https://github.com/pydantic/bump-pydantic/issues/134

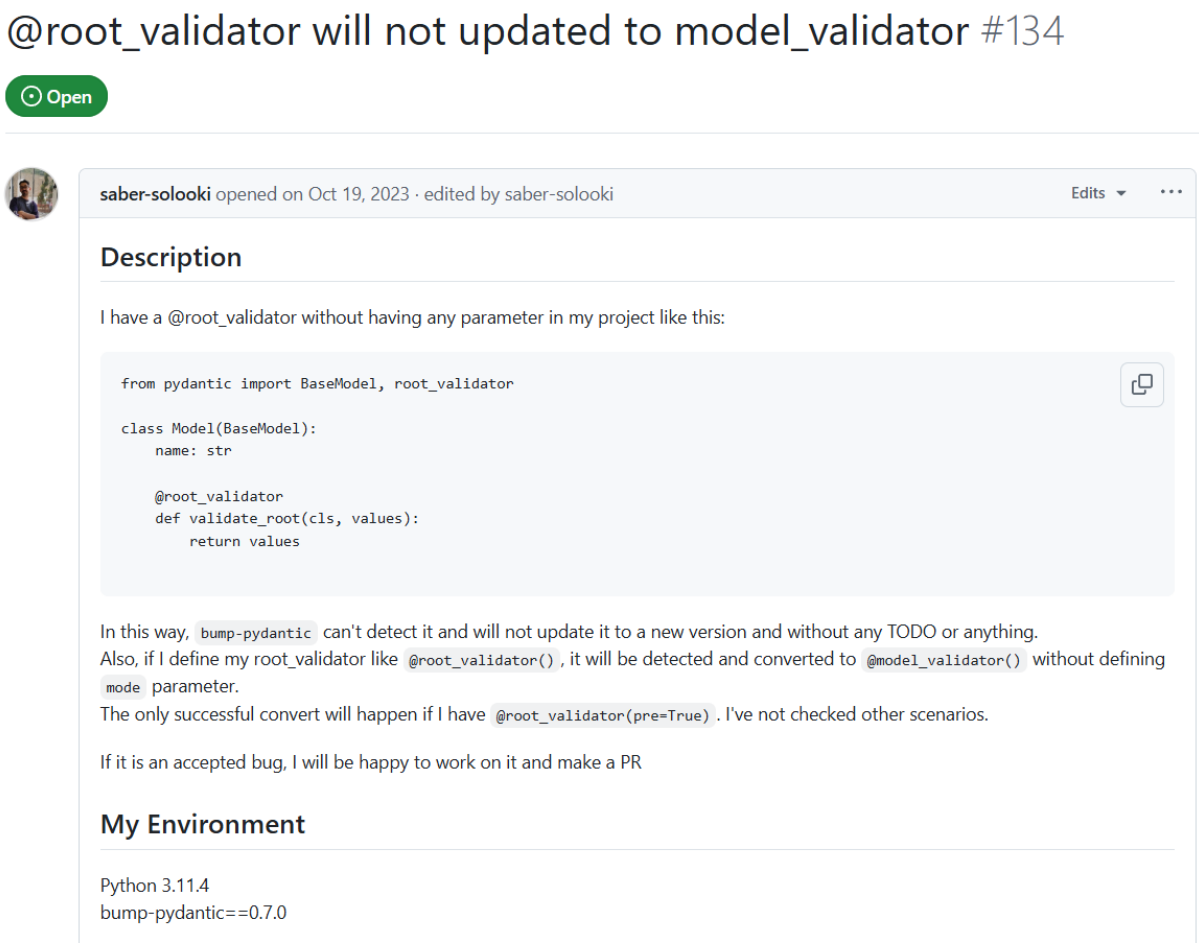

Figure 2.2: Limitation of traditional migration approaches in the `bump-pydantic` repository.

that went through this migration process for our studies: SQLAlchemy and Pydantic.

The remainder of this section details these changes and how they impact the specific libraries selected. First, Section 2.2.1 outlines the transition to Asynchronous I/O and the challenges of the "colored function" problem in Python development. Section 2.2.2 discusses the implementation of Static Typing for the Python language. Building on this context, Section 2.2.3 presents the architectural evolution of SQLAlchemy. Finally, Section 2.2.4 describes changes of Pydantic V2.

2.2.1 Asynchronous I/O in Python

Traditionally, Python applications relied on synchronous execution, where I/O operations (such as database queries or API requests) blocked the main execution thread until completion. While this strategy makes code easier to be written, this synchronous

(thread-based) model struggles to scale under high concurrency caused by the overhead of context switching and memory usage [28, 52]. To address this, Python 3.4 introduced the `asyncio` module, which implements a multitasking model based on the `async` and `await` keywords. In practice, the `async` keyword is used to define functions that can operate concurrently, without blocking the main thread. Within these functions, the `await` keyword marks the specific point where execution should pause to wait for an external result, allowing the system to handle other tasks effectively in the meantime.

The motivation for this change is performance: by suspending execution while waiting for I/O, a single thread can handle multiple concurrent connections. However, its adoption is hindered by a semantic segregation often referred to as the "colored function" problem [26]. In this paradigm, functions are divided into two categories: synchronous and asynchronous. A synchronous function cannot natively call and await an asynchronous one without blocking the event loop. Consequently, migrating a low-level function (e.g., a database query) to be asynchronous triggers a ripple effect: every function in the call stack that invokes the modified function must also be redefined as `async` and updated to use `await`.

This change requires not only syntactic updates but a structural reorganization of the control flow, as shown in Figure 2.3. As illustrated, simply adding the `async` keyword to the `fetch_data` function (Line 3) is insufficient. It triggers a refactoring chain that forces the developer to also update: (1) the `time.sleep` function (Line 5), which must be replaced by `asyncio.sleep`; (2) the `main` function (Line 8), which must become asynchronous; (3) the `fetch_data` call (Line 10), which requires the `await` keyword; and (4) the script execution code (Line 14), which must leverage the `asyncio.run` method.

```python
import time

def fetch_data():
    # Blocks execution for 1s
    time.sleep(1)
    return "Data"

def main():
    # Runs synchronously,
    needing no keyword
    data = fetch_data()
    print(data)

if __name__ == "__main__":
    main()
```

(a)

```python
import asyncio

async def fetch_data():
    # Suspends execution
    await asyncio.sleep(1)
    return "Data"

async def main():
    # Must await the result
    data = await fetch_data()
    print(data)

if __name__ == "__main__":
    asyncio.run(main())
```

(b)

Figure 2.3: Synchronous (a) and asynchronous (b) code using the `asyncio` module

2.2.2 Static Typing in Python

Python is dynamically typed by design, meaning variable types are checked at runtime. While this offers flexibility, it can lead to runtime errors that are hard to identify in large codebases. To mitigate this, Python version 3.5 introduced Type Hints, a standard syntax for annotating variable types (e.g., `name: str`).

The primary motivation is code quality and tooling support. Type hints enable static analysis tools (like Mypy and Pyright) to catch type-related bugs before execution, effectively bringing compile-time safety to an interpreted language. Figure 2.4 illustrates how validation during development may prevent bugs. The dynamic approach (Figure 2.4a) defines the function's parameter without a type hint (Line 1). Since the parameter name (`code`) can be interpreted as either a `string` or a `number`, the developer in the example passed the number 200. However, inside the function, we use the `upper()` function (Line 4), which is part of the `str` class. This would result in the application breaking during runtime due to an invalid method access error. On the other hand, by using static types (Figure 2.4b), we can explicitly define that the `code` parameter must be of type `str` (Line 1). This ensures that, if the developer passes a variable of any other type, a warning can be raised during development time by an IDE or a type checker, preventing the code from being deployed to production and avoiding runtime errors.

Modern Python libraries take this a step further by using these hints for runtime data validation, transforming Python's optional typing into a strict contract mechanism. Therefore, to use the library, the developer must upgrade their code to use static typing, otherwise it may not work as expected.

```
1 def get_status(code):
2     # Risk: 'code' is unknown.
3     # If int is passed, this
    breaks at runtime with
    AttributeError
4     return code.upper()
5
6 # No warning during development
7 get_status(200)
8 # -> ERROR
```

(a)

```
1 def get_status(code: str) -> str
    :
2     # Explicit contract
3     return code.upper()
4
5 # Static Analysis Tool Error:
6 # "Argument 1 to 'get_status'
    has
7 # incompatible type 'int';
    expected 'str'"
8 get_status(200)
```

(b)

Figure 2.4: Python code without (a) and with (b) type hints

2.2.3 Target Library: SQLAlchemy

SQLAlchemy is a widely used Object-Relational Mapper (ORM) for Python [42]. Its version 2.0 represents a modernization that enforces the two major changes described previously: Asynchronous I/O and Static Typing. To take advantage of these new features, a refactoring effort is required on two fronts. First, to support asynchronous I/O (Section 2.2.1), queries created using the synchronous `Session.query()` API must be migrated to use the asynchronous `select()` construct. This modern approach uses a syntax similar to SQL to create queries, along with the `async/await` keywords. Second, to satisfy the static typing requirement (Section 2.2.2), the library introduced a new declarative system where model attributes are explicitly annotated with generic types (e.g., `Mapped[int]`). This allows IDEs to infer the types of database columns without running the code and allows the library to validate types during runtime for simple conversions.

Figure 2.5 shows an example of such changes. In version 1.4 (Figure 2.5a), a table is declared with the `Column` object (Lines 5 and 9), which receives the type of the column and possible parameters it should have (e.g., `primary_key`). In version 2.0 (Figure 2.5b), the library uses static typing to infer the type of each column, based on the type passed to the `Mapped` class (Lines 5 and 6). Regarding the way queries are written, version 1.4 uses synchronous calls and the `Session.query()` API (Line 12), while version 2.0 uses the newer `select()` construct syntax with the `await` keyword (Line 9).

```
1  # 1. Model Definition
2  class User(Base):
3      __tablename__ = 'user'
4      # Type is hidden inside
       Column
5      id = Column(
6          Integer,
7          primary_key=True
8      )
9      name = Column(String)
10
11 # 2. Query Execution
12 users = session.query(User).all
       ()
```

(a)

```
1  # 1. Model Definition
2  class User(Base):
3      __tablename__ = 'user'
4      # Explicit Type Hinting
5      id: Mapped[int] =
       mapped_column(primary_key=
       True)
6      name: Mapped[str] =
       mapped_column()
7
8  # 2. Query Execution
9  res = await session.execute(
       select(User))
10 users = res.scalars().all()
```

(b)

Figure 2.5: SQLAlchemy migration from version 1 (a) to version 2 (b)

2.2.4 Target Library: Pydantic

Pydantic is a data validation library that relies on the static typing shift described in Section 2.2.2. It uses Python type hints to define schema and validation rules, widely used in modern web frameworks (like FastAPI).

The release of Pydantic V2 introduced a complete rewrite of the core logic to improve performance. While this brought speed improvements, it introduced numerous breaking changes. Figure 2.6 illustrates two of these changes. The first change is related to the `Config` class. In Pydantic V1 (Figure 2.6a), this class could be added to a model in order to customize its behavior, such as adding automatic ORM serialization. This was done by passing an `orm_mode` attribute that enables this behavior (Line 8). However, in Pydantic V2 (Figure 2.6b), not only does the configuration now take place via a dictionary, but many parameters were also renamed or removed. The `orm_mode`, for example, is now called `from_attributes` (Line 8). The second change is related to the attribute validators. In V1, there was a single decorator called `@validator` (Line 10), but in V2 it was broken into two new decorators: `@field_validator` and `@model_validator`.

```
1  from pydantic import BaseModel,
       validator
2
3  class User(BaseModel):
4      id: int
5      name: str
6
7      class Config:
8          orm_mode = True
9
10     @validator('name')
11     def name_must_be_capitalized
       (
12         cls, v
13     ):
14         return v.title()
```

(a)

```
1  from pydantic import BaseModel,
       field_validator, ConfigDict
2
3  class User(BaseModel):
4      id: int
5      name: str
6
7      model_config = ConfigDict(
8          from_attributes=True
9      )
10
11     @field_validator('name')
12     @classmethod
13     def name_must_be_capitalized
       (
14         cls, v: str
15     ) -> str:
16         return v.title()
```

(b)

Figure 2.6: Comparison between Pydantic V1 (a) and V2 (b) class declarations.

2.3 Large Language Models

Large Language Models (LLMs) represent a significant advancement in artificial intelligence, demonstrating capabilities in understanding and generating both natural language and source code. Built upon the Transformer architecture [47], these models are pre-trained on massive datasets to capture patterns across diverse domains [36]. Their fundamental operating principle involves the probabilistic prediction of the next token in a sequence based on the context provided by previous tokens. This mechanism allows LLMs to perform complex reasoning and generative tasks, ranging from text summarization to creative writing, and automated code synthesis.

The remainder of this section details the technical and practical aspects of LLMs relevant to this study. First, Section 2.3.1 provides an overview of the evolution and adoption of these models. Next, Section 2.3.2 explains the mechanics of *Prompt Engineering*, describing the strategies used to improve model output. Building on these concepts, Section 2.3.3 reviews current applications in Software Engineering. Finally, Section 2.3.4 introduces the concept of Agentic Systems.

2.3.1 Foundations and Adoption

The field of Natural Language Processing (NLP) experienced a paradigm change with the introduction of the Transformer architecture by Vaswani et al. [47]. Unlike previous recurrent architectures (e.g., RNNs) that processed data sequentially, Transformers rely on a Self-Attention mechanism to weigh the relevance of different words simultaneously. Originally, this architecture was designed as an Encoder-Decoder structure: the encoder processed the input for understanding, while the decoder generated the output. In the context of Software Engineering, the self-attention mechanism is particularly useful, as it enables the model to capture long-range dependencies inherent in source code more effectively than sequential models. An example is linking a variable usage to its declaration, even if they are in different files.

Modern LLMs, such as the GPT series [6], diverge from this original dual-block design by utilizing a decoder-only architecture. This structural choice optimizes the model specifically for autoregressive tasks, where the primary goal is to synthesize coherent sequences of text and code rather than classify inputs. Trained using a self-supervised objective known as Causal Language Modeling, these models learn to predict the next token in a sequence given the preceding context. Through training on massive corpora,

this architecture enables the acquisition of a deep statistical understanding of syntax and semantics, leading to emergent behaviors like "in-context learning" [36]. Which is the ability to perform unseen tasks solely based on natural language instructions without parameter updates.

However, the output of these models is inherently probabilistic. During inference, the text generation process is controlled by hyperparameters, most notably Temperature. This parameter modifies the probability distribution of potential next tokens, directly influencing the randomness of the output. High temperatures flatten the probability distribution, increasing diversity, whereas lower temperatures sharpen the distribution, forcing the model to select high-probability tokens [56]. For deterministic Software Engineering tasks, such as code generation, low temperatures are typically preferred to ensure syntactic correctness and reproducibility [35]. We adopt this deterministic setting for our migration tasks to minimize hallucinations.

2.3.2 Prompt Engineering

The performance of Large Language Models is dependent on the quality of the input instructions, a constraint that led to the development of *Prompt Engineering*. This field focuses on optimizing inputs to guide the model toward specific goals and outputs [36]. In the context of code manipulation, distinct prompting strategies may alter the model's accuracy and output format. Figure 2.7 illustrates three common strategies applied to a hypothetical migration task. The strategies shown in Figure 2.7 differ in the guidance provided to the model. A migration context is included in the figure to help understanding the task, but is not part of any prompt. Particularly, the figure compares three prompting strategies

- **Zero-Shot (Panel A):** The model receives only the task description, with no examples, reasoning hints, or API documentation. This tests the model's intrinsic knowledge, but often yields variable results for complex tasks where the target API pattern is not well-known to the model or when the library version is very recent.
- **Few-Shot (Panel B):** The prompt includes one or more examples. Sahoo et al. [36] highlight that providing even a single example (One-Shot) helps the model recognize the expected pattern and output format, such as including specific arguments that were implied in the example.

Migration Context
Deprecated API and its replacement in the new version.

```
Deprecated (v1.x): db.connect()
New API    (v2.x): db.open(timeout=30)
```

A. Zero-Shot Prompting
Relies only on the model's pre-trained knowledge.

```
Prompt:  Migrate the code 'db.connect()' to the new API.
Output:  db.create_connection()
```

B. Few-Shot Prompting (One-Shot Example)
Provides examples to guide the expected format.

```
Prompt:  Migrate the code 'db.connect()' to API v2.  Follow the example bellow:
Example:'db.get_user(active=True)'
Output:  db.create_connection(active=True)
```

C. Chain-of-Thought Prompting
Guides the model with explicit reasoning steps before the final answer.

```
Prompt:  Migrate the code 'db.connect()' to API v2.  Think step-by-step:
  Step 1:  The API in version 2 requires a timeout parameter
  Step 2:  The Method 'connect' is now called 'open'.
Output:  db.open(timeout=30)
```

Figure 2.7: Comparison of prompting strategies applied to a hypothetical migration task.

- **Chain-of-Thought (Panel C):** This approach instructs the model to use a list of steps for its reasoning before producing a final answer. This leads the LLM to think "step-by-step" when working on complex tasks. This approach may yield better results [51].

2.3.3 Applications in Software Engineering

In the domain of Software Engineering, LLMs have evolved from experimental tools to essential components of the development workflow. Recent studies demonstrate their effectiveness in automating critical tasks such as unit test generation [40], automated bug fixing [41], and code review automation [44, 45]. These applications typically leverage the pattern-matching capabilities of models to identify common coding structures and suggest improvements.

Beyond these daily tasks, LLMs are increasingly applied to large-scale maintenance activities, specifically library migration. Unlike simple code completion, migration tasks require the model to understand the semantics of deprecated APIs and map them to their

modern equivalents. This process often involves context that spans multiple files and requires logical deductions about how data flows through the application. Recent works explore the use of LLMs to automate these updates, aiming to reduce the manual effort required to keep software systems up-to-date and secure against vulnerabilities [2].

However, using LLMs with migration tasks is not without challenges. Models can suffer from hallucinations, generating syntactically correct but semantically non-existent code. They are also constrained by context window limits. These limitations often hinder the migration of large codebases when using simple interaction modes, requiring more autonomous and context-aware approaches.

2.3.4 Chatbos vs. Agentic Systems

Most developers interact with LLMs through conversational interfaces, commonly referred to as *Chat Mode*. In this paradigm, the interaction is passive and linear: the user provides a natural language query (which we refer as Prompt), and the model generates a text-based response. While effective for isolated tasks—such as explaining a code snippet or generating a single function—this mode places the burden of execution entirely on the user. The developer must manually copy the generated code, integrate it into the project, run tests, and iteratively feed any error messages back to the model. This "human-in-the-loop" dependency becomes a bottleneck for complex maintenance tasks like library migration, which often require modifying multiple files and verifying changes against a test suite.

To address these limitations, the field is shifting toward Agentic Systems. Unlike a passive chatbot, an AI Agent is designed to pursue a high-level goal autonomously. According to Plaat et al. [30], an agentic system distinguishes itself through three core capabilities:

- **Perception:** The ability to gather context from the environment (e.g., reading the file system or scanning a repository structure) without explicit user pasting.
- **Reasoning and Planning:** The capacity to decompose a complex goal into a sequence of logical steps before execution.
- **Tool Use:** The authority to execute actions, such as running terminal commands, writing to files, or invoking compilers.

Developers typically access these capabilities through two primary interfaces. The first is via programmable APIs, such as OpenAI's API. This method allows the creation of

custom scripts to interact with models in a structured request-response pattern, serving as the foundation for automated but non-autonomous workflows. The second interface involves integrated environments like GitHub Copilot and Cursor. These tools have evolved to incorporate fully agentic capabilities, such as Copilot's Agent Mode, which allows developers to delegate multi-step tasks directly within the editor.

This architecture enables self-correction loops. For instance, in a migration scenario, an agent can modify a file, execute the project's tests, observe a failure in the terminal output, and autonomously trigger a new reasoning step to fix the bug without human intervention. This shift from *predicting text* to *executing workflows* represents a significant evolution in AI-assisted development.

In this dissertation, we investigate both paradigms to assess this technological advancement. Specifically, our first study explores the capabilities of standard LLMs guided by external scripts (representative of the passive approach), while our second study evaluates the efficacy of fully integrated Agentic Systems in handling the end-to-end migration workflow.

2.4 Evaluation of Automated API Migration

Evaluating the quality of automatically generated code is a significant challenge, particularly in the context of library migration. Unlike simple code completion tasks—where success can often be measured by textual similarity to a reference solution—migration involves structural and semantic transformations that render textual metrics (such as BLEU or CodeBLEU) insufficient [29, 34]. Therefore, to assess an LLM's performance in updating APIs, three complementary dimensions are typically considered: functional correctness, static consistency, and the extent of the migration coverage.

The remainder of this section details the specific dimensions selected to assess migration quality in the context of API migration. First, Section 2.4.1 addresses functional correctness, highlighting the reliance on regression testing to ensure functionality preservation. Complementing this, Section 2.4.2 explores the role of static analysis in detecting non-existent APIs and type violations typical of LLM hallucinations. Finally, Section 2.4.3 introduces the concept of Migration Coverage, a metric designed to evaluate the completeness of the migration by measuring the ratio of deprecated elements successfully refactored.

2.4.1 Functional Correctness

A good measure of a successful migration is the preservation of software behavior such that the change is transparent to the client. In Software Engineering, this is traditionally evaluated through regression testing. If the migrated code passes the existing test suite that validated the original version, it is assumed that the external behavior remains consistent [55].

However, relying solely on test execution has limitations in the context of LLM-generated code. First, the test suite itself may have low coverage, allowing subtle bugs introduced during migration to pass undetected. Second, LLMs can generate code that is syntactically correct and passes loose tests but fails in edge cases or introduces performance regressions—such as blocking the event loop in an asynchronous context (as discussed in Section 2.2.1). Therefore, while functional correctness is the "gold standard," it is often necessary to combine it with static analysis.

2.4.2 Static Analysis and Typing Metrics

With the adoption of static typing in languages like Python (Section 2.2.2), static analysis tools have become powerful evaluators of code quality. Tools like Mypy and Pyright can verify type consistency without executing the code, serving as an objective filter for hallucinations.

If an LLM creates a call to a non-existent method (e.g., `session.execute_async()` instead of `session.execute()`), a static analyzer will flag this as a "method not found" error immediately. In migration studies, the reduction of static analysis errors is a strong proxy for code correctness, as empirical research indicates that strict type checking can prevent a significant ratio of runtime defects in dynamic languages [12]. A migration is considered syntactically successful if the resulting code contains no type violations, ensuring that the new API usage adheres to the library's strict contract.

2.4.3 Evaluating Migration Completeness

While functional correctness ensures the code still runs, and static analysis ensures types are valid, neither metric guarantees the completeness of the migration. A partially migrated codebase might pass all tests (especially if the legacy API is deprecated but not yet removed) and satisfy type checkers. However, there is no guarantee that the necessary changes have been applied to the code, ensuring usage of the newer APIs provided by the library.

Standard Software Engineering metrics, such as code coverage (from testing), measure the percentage of lines executed during verification. However, broadly applied code coverage is insufficient for migration tasks because it does not distinguish between modernized lines and legacy lines. A test suite might achieve 100% coverage by executing old, deprecated functions that still function at runtime (emitting only warnings), giving a false sense of security regarding the migration status.

To address this, it is necessary to quantify the extent to which specific API usages have been transformed. Recent research by Islam et al. [17] provides a taxonomy for characterizing Python library migrations, mapping how specific elements (e.g., classes, functions, parameters) evolve between versions. While their work focuses on classifying these changes, evaluating an automated migration using an LLM requires quantifying its success rate in applying them across a project.

This creates a gap in standard evaluation methodologies: there is a need for a metric that functions similarly to "Recall" in Information Retrieval [22]—calculating the ratio of deprecated API instances that were successfully identified and refactored. In Chapter 3, we formally define Migration Coverage to bridge this gap, proposing a systematic way to score the completeness of the transformation based on the taxonomies of API evolution.

2.5 Related Work

This section reviews the existing literature to contextualize our contributions within the field of software maintenance and automated engineering. The discussion is organized into three main areas. First, Section 2.5.1 examines the challenges found when migrating code. Section 2.5.2 explores the application of Large Language Models in Software engineering. Lastly, Section 2.5.3 narrows the focus to code migration using Large Language

Models.

2.5.1 API Migration in Software Engineering

The evolution of software libraries imposes a continuous maintenance burden on client applications, a phenomenon that has been extensively characterized in empirical Software Engineering research. Studies typically highlight the tension between the necessity of evolution (to fix bugs or improve performance) and the cost of breaking changes, which often leads to technical debt and dependency lag [3, 18]. In this section, we review specific works that analyze the limitations of manual migration and traditional automation techniques, contrasting them with our proposed LLM-based workflows.

Brito et al. [5] investigated the quality of deprecation mechanisms in large-scale software ecosystems. Through an empirical analysis of over 600 Java systems, they found that, on the median, approximately 33% of deprecated API elements lacked any "replacement message" to guide developers toward the new solution. This gap forces developers to manually search for migration strategies or deduce them from source code. In contrast, our work investigates the potential of Large Language Models to overcome this lack of explicit documentation, assessing their ability to infer semantic replacements for deprecated APIs even when the library maintainers fail to provide migration guides.

Lamothe et al. [19] conducted a systematic literature review on API evolution and compatibility, analyzing over 300 primary studies to map the landscape of existing tools. Their review revealed that the majority of research focuses on detecting breaking changes or assessing their impact, rather than automating the repair of the client code. Unlike these detection-centric approaches, our research shifts the focus to the execution of these repairs, evaluating how Generative AI can be used to synthesize the necessary code transformations and reduce the manual effort involved in the process.

In the specific context of Python, Malloy and Power [21] empirically analyzed the ecosystem-wide transition from version 2 to 3, highlighting the limitations of rule-based automation. They observed that standard utilities like `2to3` (An AST-based tool) were often insufficient to guarantee a successful migration, leading many projects to delay updates. While these tools handled simple syntactic mappings effectively, they failed to address semantic changes or code that relied on specific runtime behaviors. In contrast, our work adopts LLMs that are not bound by rigid syntactic rules, enabling the handling of complex structural refactorings which AST-based tools cannot safely perform, such as the conversion of synchronous functions to asynchronous coroutines.

Addressing the specific challenge of asynchronous migration, Turcotte et al. [46]

proposed a technique to automatically refactor synchronous functions into more modern `async/await` patterns in Javascript. Their approach relied on static analysis to construct call graphs and propagate asynchronous constraints, aiming to identify the ripple effect of changing a blocking API to a non-blocking one. However, acknowledging that static analysis struggles with the dynamic features of interpreted languages, our work explores the use of LLMs to overcome this rigidity, leveraging probabilistic generation to infer asynchronous transformations in Python that traditional tools often miss.

Finally, regarding the reliability of verification methods in dynamic languages, Lyu et al. [20] addressed the challenge of detecting incompatibility issues between Python library versions. They highlighted that Python's dynamic nature makes standard static analysis insufficient for capturing runtime behaviors and type mismatches, often requiring actual code execution to guarantee compatibility. Addressing this limitation, our research integrates dynamic verification as a core component of the migration workflow. Unlike static tools that rely solely on syntax checking, our approach validates the LLM-generated code against the project's test suite, using runtime feedback to ensure the correctness of the transformed code in a way that static analysis cannot.

2.5.2 Large Language Models in Software Engineering

The application of Large Language Models (LLMs) in Software Engineering has shifted from simple code completion to handling complex maintenance tasks. Recent literature demonstrates that these models possess a semantic understanding of code that allows them to perform activities previously requiring human reasoning, such as automated debugging, code explanation, and test generation. In this section, we review key studies that assess the capabilities of LLMs in these domains and contrast their general-purpose applications with our specific focus on library migration.

Regarding the general capability of LLMs as coding assistants, Tian et al. [43] conducted a comprehensive empirical study to evaluate models like ChatGPT across diverse tasks, including code generation and documentation. They assessed performance on standard benchmarks and found that while LLMs can generate syntactically correct code for standalone algorithmic problems, they frequently struggle with complex algorithmic logic and subtle edge cases. While their study provides a broad baseline, our work narrows the scope to a specific, high-complexity maintenance task in library migration. This was done to investigate whether the model can go beyond generating isolated snippets to correctly transforming existing codebases while preserving application behavior.

In the domain of interactive repair, Xia and Zhang [53] introduced a conversational

approach to bug fixing, demonstrating that providing iterative feedback significantly improves results. Their study showed that when an LLM is presented with the output of a failed test, it can self-correct its previous attempt, fixing 162 out of 337 bugs in the Defects4J benchmark. Our work contrasts with this interactive paradigm by focusing on fully autonomous execution. Instead of relying on a developer to guide the correction process through conversation, we restrict our evaluation to a hands-off setup, presenting the initial prompt and assessing the LLM's ability to migrate code and resolve issues independently, without human interaction.

Finally, concerning quality assurance, Schäfer et al. [39] explored the use of LLMs for generating unit tests. Their research highlighted that while models can generate plausible test cases, they often produce code with incorrect assertions or references to non-existent variables, requiring iterative repair. Our approach differs by grounding the verification in the project's existing test specifications rather than generating them from scratch. However, recognizing that major architectural shifts such as asynchronous migration inevitably break synchronous tests, we do not treat the test suite as immutable. Instead, we grant the LLM the autonomy to refactor the test code alongside the application logic, treating them as part of the migration process.

2.5.3 Code Migration using Large Language Models

While general software maintenance tasks like bug fixing and test generation have been widely explored, automated code migration represents a distinct challenge due to its requirement for consistency across multiple files and the preservation of complex runtime behaviors. In this section, we examine the state-of-the-art in applying LLMs to large-scale migration, distinguishing between human-in-the-loop workflows and emerging autonomous agentic approaches.

At an industrial scale, Ziftci et al. [58] reported on Google's internal use of LLMs to migrate massive C++ and Java codebases. Their approach integrated LLMs into a build system equipped with change location discovery, relying on a rigorous human-in-the-loop review process to ensure correctness. They found that while LLMs could automate approximately 75% of the code changes, they often struggled with subtle semantic nuances, requiring developer intervention to fix build and test failures. This highlights a key limitation in current industrial practices: the reliance on expensive human review to compensate for the model's nondeterminism. In contrast, our work investigates the capacity of the LLM on working entirely by itself. We assess whether current models have reached a level of maturity where they can perform these migrations autonomously, evaluating

their performance without the interactive guidance or manual corrections relied upon in industrial workflows.

Focusing specifically on the reliability of these models in the Python ecosystem, Islam et al. [16] conducted a large-scale empirical study on library migration using state-of-the-art models like GPT-4 and Llama 3. Their analysis of over 300 real-world migration scenarios revealed that, despite high syntactic accuracy, models frequently fail to capture the semantic nuances of API changes, leading to "silent failures" where code runs but behaves incorrectly. Their findings underscore that without a robust verification mechanism, raw LLM output is insufficient for production-grade migration. This empirical evidence supports our methodological choice to implement Migration Coverage, a metric designed to ensure that the migrated code is not just syntactically valid, but actually exercised and verified by the test suite.

To overcome these reliability issues, recent research has pivoted towards multi-agent architectures. Pucho et al. [31] proposed a specialized multi-agent framework for refactoring Python code in machine learning projects. By decomposing the refactoring task into specialized roles—such as strategy planning and code transformation—they demonstrated that agentic systems significantly outperform single-prompt approaches in handling complex structural changes. However, they also reported that the absence of a dedicated validation step led to a significant number of syntactically invalid outputs. Our research directly tests this limitation in the context of asynchronous migration. By conducting a comparative analysis between a scripted strategy and an agentic workflow we aim to determine whether the internal reasoning capabilities of an autonomous agent are sufficient to produce valid code, or if they suffer from the same hallucination issues reported in the literature.

2.6 Final Remarks

This chapter reviewed the challenges of software maintenance in the context of library migration, emphasizing that API evolution often introduces breaking changes that are costly to resolve manually. We discussed how traditional automated techniques, such as static analysis and refactoring tools, frequently fall short in dynamic languages like Python, where resolving dependencies requires semantic understanding rather than just syntactic transformation.

The emergence of Large Language Models has introduced a new paradigm for addressing these migration tasks. As our review of the state-of-the-art indicates, while LLMs demonstrate significant potential for code generation and repair, their application

to large-scale migration is still maturing. Current industrial approaches rely heavily on human-in-the-loop workflows to ensure correctness, while academic benchmarks reveal that autonomous models frequently produce plausible but functionally incorrect code. Our research addresses this gap by investigating the limits of autonomous LLM-based migration. In the following chapters, we detail our two studies, one focused in a Prompt-Based strategy and the other in an Agent-Based method. These aim to analyze the capacity of an LLM in correctly migrating code between library versions without the need for human intervention.

Chapter 3

Study Design

In this chapter, we introduce the aspects of the methodology that are shared across both studies described in this dissertation. In Section 3.1, we start by describing the libraries chosen for these studies, as well as a brief description of the main changes expected to when migrating them from version 1 to version 2. Next, Section 3.2 details the construction of the two datasets used in the studies, one for each library. Section 3.3 describes the metrics adopted in both studies, including a novel metric called Migration Coverage, which measures the percentage of API usages correctly migrated in a repository. Lastly, Section 3.4 describes threats to validity, while Section 3.5 concludes this chapter.

3.1 Target APIs

SQLAlchemy is an Object-Relational Mapping (ORM) tool that facilitates communication between Python applications and relational databases. It simplifies development by abstracting complex database connection and data manipulation tasks. It supports the connection to various databases, such as PostgreSQL, MySQL, and Oracle, using a simple API that resembles pure SQL queries. The API is well-known and widely adopted in the open source community, with more than 11.1K stars, 1.6K forks, and 1.1m uses among other GitHub repositories [42].

In this dissertation, we address the migration from SQLAlchemy version 1 to version 2. While version 1 is widely-adopted, version 2 was introduced to leverage recent advancements in the Python ecosystem. Among the most significant changes in version 2, the compatibility with Python's static typing stands out, improving error detection during development and facilitating the maintenance of large applications. Furthermore, version 2.0 introduces major performance optimizations for asynchronous operations. Lastly, SQLAlchemy 2 has improved the query syntax, making it more intuitive and easier to use.

Pydantic, on the other hand, is a data validation and settings management library that leverages Python's native type hints to ensure structures are validated at runtime. It enables developers to define models containing specific rules for attribute validation, promoting type safety and data integrity. Pydantic's simple API also makes it easy to parse and validate complex data types while providing clear error messages. This library has gained significant popularity due to its performance and ease of use, with more than 25.7K stars, 2.3K forks, and usage in over 887K dependent repositories on GitHub [32].

Our focus is on migrating Pydantic from version 1 to version 2. Unlike SQLAlchemy, Pydantic version 2 introduces several breaking changes, requiring users to adapt their code to the new version. Most adaptations involve renaming elements or adding and removing functions and parameters. For instance, the `Config` subclass was removed and replaced by the `model_config` attribute, while both `@validator` and `@root_validator` were replaced by `@field_validator` and `@model_validator`, respectively. Finally, the `BaseSettings` class was removed from the original package and relocated to a new library called `pydantic-settings`.

3.2 Dataset Creation

To gather a diverse set of client applications, we built two distinct datasets: one for SQLAlchemy repositories and another for Pydantic repositories. Both datasets were constructed using a crawling script that interacted with GitHub's GraphQL API. The script used the query detailed in Listing 3.1 to fetch repositories that explicitly mentioned each library. Only repositories with at least 50 stars were included, ensuring a minimum level of community adoption and maintenance activity. For SQLAlchemy, we included repositories created since 2018, coinciding with the library's major version release. Similarly, for Pydantic, we included repositories established since 2017, the year the Pydantic library was first pushed to GitHub. This approach ensured that our datasets included a wide range of client applications that have likely adopted and evolved with these libraries over time. The data collection was performed in June, 2024.

Following the automated collection, a manual curation process was applied to select repositories suitable for the experiment. Repositories without passing tests were removed, as well as repositories that already used SQLAlchemy or Pydantic in version 2. Lastly, only projects where the application and tests could be executed successfully were retained.

```
query getRepos {
    search(
      type: REPOSITORY
      first: %(first)i
      query: "stars:>=50 language:python created:%(period)s %(library)s
   in:file"
      after: %(after)s
    ) {
      pageInfo {
        hasNextPage
        endCursor
      }
      edges {
        cursor
        node {
          ... on Repository {
            nameWithOwner
            url
            star_count: stargazerCount
            createdAt
            updatedAt
          }
        }
      }
    }
  }
```

Listing 3.1: Graphql query used to extract repositories from Github

A summary of the selection and filtering process for both libraries is presented in Table 3.1. The initial query yielded 135 candidate repositories for SQLAlchemy. From this total, 44 were discarded for lacking a working test suite, 49 for already using the library in its version 2, and 32 due to execution failures. This cleaning process resulted in a final dataset of 10 repositories for this library. Similarly, the initial query fetched 189 repositories for the Pydantic library. Out of these, 25 had no passing tests, 139 were already using Pydantic v2 and 14 failed to be executed. This resulted in 11 suitable repositories.

Table 3.1: Summary of repository selection and filtering process

Library	**Initial**	**No Tests**	**Already v2**	**Execution Failures**	**Final**
SQLAlchemy	135	44	49	32	10
Pydantic	189	25	139	14	11

Table 3.2 provides a characterization of the 21 repositories that constitute the final datasets. For the SQLAlchemy dataset, star counts range from 81 (`sqlalchemy-adapter`) to 2,771 (`fastapi-template`), while the number of Python source files varies from 7 to 139, reflecting projects of different scales and complexities. The Pydantic dataset presents a similar diversity, with star counts ranging from 54 (`pydantic-vault`) to 1,404 (`pydantic-sqlalchemy`), and Python file counts from 4 to 69. In both datasets, the number of test cases also varies substantially, from as few as 4 to as many as 87 and 63 tests for SQLAlchemy and Pydantic repositories, respectively. This allows the studies to evaluate LLM migration performance across a broad spectrum of real-world project characteristics.

Table 3.2: Characterization of the final dataset repositories as of 2026

Repository	**Stars**	**Python Files**	**Tests**
SQLAlchemy Repositories (Study 1)			
`sqlalchemy-adapter`	81	7	17
`db-to-sqlite`	490	12	9
`alembic_utils`	295	45	87
`Fast-Api-example`	177	15	16
`fastapi-debug-toolbar`	167	58	18
`FastApi-Strawberry[...]-BoilerPlate`	150	30	12
`fastapi-template`	2,771	11	8
`graphene-sqlalchemy-filter`	119	50	48
`nebulo`	92	139	49
`paracelsus`	172	22	20
Pydantic Repositories (Study 2)			
`djantic`	471	25	32
`FastAPI-Pydantic-[...]_CRUD_API`	172	28	21
`finstruments`	198	69	57
`mongox`	123	18	28
`ormdantic`	150	25	39
`pydantic-sqlalchemy`	1,404	4	4
`pydantic-vault`	54	6	45
`pydastic`	123	9	31
`pydbantic`	240	42	26
`lnurl`	67	15	63
`pydantic-numpy`	58	6	12

3.3 Evaluation Metrics

To assess the effectiveness of the migration and determine whether the migrated applications function as intended, we defined four categories of metrics, detailed in the following subsections. These metrics enable us to evaluate the migration from three complementary perspectives: (1) the accuracy of the library migration itself; (2) the preservation of code functionality; (3) the overall quality of the produced code.

3.3.1 Migration Coverage

To measure the agent's effectiveness at the code-transformation level, we defined a Migration Coverage metric. The methodology builds upon existing research on characterizing API changes in Python [17], which analyzed patterns of library evolution. We apply these principles to quantify the extent to which the LLM correctly migrated API usages in a given client application.

To calculate this metric, we first manually identify all library usages requiring an update. We then use a table (see an example in Table 3.3) to systematically track each required transformation. The first two columns define the change (e.g., renaming `Column` to `mapped_column`). The third column records the total number of instances where this change is needed, and the fourth column records the number of times the LLM successfully performed it. Migration Coverage is calculated as the ratio between the sum of the values in the fourth column and the sum of the values in the third column.

Table 3.3: Migration Coverage Example

Before	After	Instances	Score
`Column`	`mapped_column`	3	2
`create_engine`	`create_async_engine`	5	2
`session.query`	`select`	3	1
Total		11	5

An example is shown in Table 3.3, referring to an API with three changes. We can see that the class `Column` (first column) was renamed to `mapped_column` (second column) in the new version of the API. In the system under analysis, this class is used in three source code locations (third column), of which two were correctly migrated by the LLM (fourth column). We can also see the old API was referenced in 11 code location in total,

of which 5 were correctly migrated by the LLM. Thus, the Migration Coverage for this example is $5/11 \approx 45\%$.

This metric provides a direct measure of how many of the necessary transformations, required by the library version change, have been successfully implemented.

3.3.2 Percentage of Passing Tests

To evaluate whether the migrated applications preserved their original functionality, we defined the Percentage of Passing Tests metric. This metric measures the proportion of automated tests that continue to pass after migration, providing a direct indication of whether the system's functional behavior was maintained.

All repositories selected for this study contained automated test suites. For each migration, the tests were executed both before and after the process, and the metric was computed as the ratio between the number of tests passing after migration and the total number of tests in the repository. However, since not all repositories had 100% of their tests passing before migration, the analysis focuses on whether the proportion of passing tests remained consistent before and after migration. In other words, a migration is considered functionally successful if the same subset of tests continues to pass afterward, even if the total percentage of passing tests is below 100%.

This metric serves as an empirical measure of whether the application continues to behave as expected after migration, allowing us to identify potential regressions introduced during the process.

3.3.3 Application Compiles

While the previous metric evaluates whether the migrated applications maintain their functional behavior, this one assesses whether the application successfully compiles after migration. It ensures that the migrated code is at least syntactically correct and that all necessary dependencies and imports are properly resolved before execution.

Although Python is an interpreted language, there is still an intermediate compilation phase in which the source code is translated into bytecode before execution. Therefore, we define a compilation failure as any error that prevents the application from being executed, typically occurring before the runtime phase. These include, for instance,

`SyntaxError` exceptions caused by invalid syntax introduced during the migration, and `ImportError` or `ModuleNotFoundError` exceptions caused by incorrect or missing import statements.

This metric is binary, indicating whether the migrated application can be successfully executed without errors at startup. In practice, it serves as an early indicator of whether the migration process produced runnable code, even before functional or quality analysis is performed.

3.3.4 Quality Metrics

The last group of metrics focuses on assessing the quality of the code produced after migration. These metrics aim to determine whether the migrated code maintains a similar level of quality compared to the version before migration. To evaluate this dimension, we used two well-established tools in the Python ecosystem: Pylint and Pyright. Both are static analysis tools but focus on different aspects of code quality.

Pylint is one of the most widely adopted linters in the Python community, with over 5.6K stars on GitHub [33]. It checks for a wide range of issues, including syntax errors, code style violations, unused imports or variables, and potential programming mistakes. After each migration, Pylint was executed using its default configuration to assess whether the migration introduced new problems or reduced the overall code quality. The resulting Pylint score—which ranges from 0 to 10—provides a quantitative measure of code quality, where higher scores indicate cleaner and more maintainable code.

Pyright, in contrast, is a static type checker designed to analyze Python codebases that use type annotations. It has also gained significant adoption in the developer community, with around 15K stars on GitHub [24]. Since both SQLAlchemy and Pydantic v2 rely heavily on Python's static typing features, Pyright was used to detect potential typing inconsistencies introduced during the migration. The tool reports the number of typing-related errors in the code. This allows us to assess whether the LLM-generated code uses modern Python typing conventions and whether the migration improved or worsened type safety.

Together, these two tools provide complementary perspectives on post-migration quality. While Pylint evaluates general code quality and adherence to best practices, Pyright focuses on correctness and consistency from a typing perspective. This combination allows us to capture both stylistic and semantic aspects of code quality, providing a more comprehensive understanding of how well the migration preserved or improved the software's maintainability.

3.4 Threats to Validity

This section presents the main threats to validity related to the methodological decisions previously described. These are shared between both studies discussed in this dissertation.

Dataset: There are thousands of projects that use each library, but our datasets include only a fraction of them. Both datasets were constructed using Github's GraphQL API with filters such as minimum star count, creation date and explicit mentions of the target libraries. These filters may have excluded representative real-world projects, such as industrial applications, internal organizational systems, or less popular but technically diverse repositories. Furthermore, the manual filtering process—where we kept only repositories that can be run and that have passing tests—likely filtered out less mature or more complex projects, which might present different migration challenges. Lastly, the usage of Github's API also means we didn't take into account other repository hosting sites, such as Gitlab or Bitbucket, potentially limiting the diversity and generalizability of the dataset.

Migration Coverage Computation: The Migration Coverage metric relies on manually identifying all relevant API usages that require migration. This introduces a threat to construct validity: human error or subjective interpretation may influence which API elements are considered "relevant" or "requiring change". Errors in classification—especially for more complex or implicit usages—could lead to overestimation or underestimation of the LLM's performance. Additionally, differences in complexity across repositories (e.g., custom wrappers around the target libraries) might cause variation in what is counted as an API usage, affecting cross-repository comparability.

Execution Environment: All repositories were executed in a controlled environment to ensure compatibility, but these conditions may not perfectly reflect the diversity of execution environments used in real-world systems. Differences in operating systems, dependency versions, database availability, and environment configuration can affect both test outcomes and compilation behavior.

3.5 Final Remarks

In this chapter, we defined the methodological aspects common to both studies discussed in this dissertation. Specifically, we described the libraries selected for migration, detailed the process used to build the datasets on which the studies were conducted, and outlined the metrics used to evaluate the migration results.

The choice of SQLAlchemy and Pydantic as migration targets was motivated by their distinct characteristics: SQLAlchemy introduces a paradigm shift toward asynchronous execution, while Pydantic imposes extensive breaking API changes aimed at improving performance. Together, they represent complementary migration challenges, enabling a broader assessment of LLM capabilities.

The dataset construction process prioritized reproducibility and diversity, selecting repositories with active test suites and a minimum level of community adoption. The filtering criteria ensured that only projects with verifiable pre-migration behavior were included, which is a prerequisite for meaningful evaluation using the functional correctness metrics described in Section 3.3.

Chapter 4

Prompt-Based Migration

This chapter describes the first study presented in this dissertation. We use a Prompt-Based migration strategy, using Open AI's Python API, to migrate the repositories in both datasets. Section 4.1 describes the three prompts used during this study: Zero-Shot, One-Shot, and Chain-of-Thoughts. Next, in Section 4.2 we report the metric results for each prompt and library. Lastly, Section 4.3 presents threats to validity and Section 4.4 concludes the chapter.

4.1 Methodology

To evaluate the effectiveness of the prompt-based migration strategy, we implemented an automated script that executes different prompting strategies over each client application. The script relied on OpenAI's Python API (v1.14) and the GPT-4o model, configured with a temperature of zero and a fixed system instruction ("You are a developer with expertise in Python"). This aimed to produce more deterministic outputs. For each repository, the migration was executed once per prompting strategy, and only the first generated output was collected for analysis. The script automatically migrated both application and test files, upgrading all SQLAlchemy or Pydantic usages from version 1 to version 2 while ignoring auto-generated or irrelevant directories such as `.git`, `__pycache__`, and virtual-environment folders.

In this first study, we explored three different prompting methods. The first is a Zero-Shot approach, where the model receives a task description in the prompt but no example is provided to illustrate the expected output. The second prompt uses a One-Shot approach, providing the model with an output example to help it understand the given task. Lastly, a Chain-of-Thoughts approach was used. In this case, the prompt contains a step-by-step guide that lead to the final output [51]. The same base command was used for the three prompts, as shown in Figure 4.1. In this command, we instruct the model to return only code, with no explanations. This is relevant to better automate the

task. Additionally, we ask it to use both Python's `asyncio` and `typing` features in order to ensure it adopts the newest features and syntax for both SQLAlchemy and Pydantic v2.

Base Prompt

The Python code bellow uses the library ¡sqlalchemy—pydantic¿ with version 1. Migrate it so that it works with version 2 of ¡sqlalchemy—pydantic¿. Make the code compatible with python's `asyncio`. Use python's typing module to add type hints to the code. Your answer must only contain code. Do not explain it. Do not add markdown backticks for code. Do not add extra functionality to the code. Do not remove code that is not being changed. If there's no need to change the code, answer only with the code itself. The first line of code must have a comment "### START CODE ###". The last line of code must have a comment "### END CODE ###".

Here is the code to migrate:

START CODE
...

Figure 4.1: Base command used for all prompts

Zero-Shot Prompting: In this setting, the model receives only the task description and the code to be migrated, with no examples demonstrating the expected output format or transformation steps. This approach provides the LLM full freedom to interpret the task, and is the simplest and most direct prompting strategy evaluated in this work [36]. The Zero-Shot prompt consists solely of the base command shown in Figure 4.1 followed by the code that should be migrated.

One-Shot Prompting: In this second approach, the model is provided with an example of correctly migrated code to serve as a reference. For the SQLAlchemy repositories, the example was retrieved from the examples folder in the original library repository. It defines some tables and operations, using the newest library version, together with `asyncio` and `typing`. For the Pydantic migration, a base migration case was extracted from the examples section in the official documentation of this library. It contains usages of most of the APIs that needed migration between versions. This additional context helps guide the model by showing the expected structure of migrated code and reducing the ambiguity present in the Zero-Shot setup. The addition made to the original prompt can be seen in Figure 4.2.

One-Shot Prompt

Use the following code block as an example of migrated code, follow the same patterns used in it:

START CODE
...

Figure 4.2: One-Shot prompt

Chain-of-Thoughts: In this case, the model receives not only the example used in the One-Shot prompt, but also a structured, step-by-step guide describing the actions required for a correct migration. This prompt is intended to simulate the internal reasoning steps involved in performing the transformation, guiding the model through essential migration operations such as replacing deprecated classes, adopting typed column declarations, switching to asynchronous operations, and resolving imports. Figure 4.3 shows the changes made to the prompt for each library, as well as commons steps used in both of them.

Chain-of-Thoughts Prompt

Use the steps bellow as a guide for the migration. You don't need to follow them exactly as described, but they should be able to help with the migration:

// SQLAlchemy's specific steps
1. Update the used database engine, if any, so that you're using 'create_async_engine' instead of 'create_engine'.
2. If any tables and their columns are declared, update their declarations so that they use 'mapped_columns' instead of 'schema.Column' and ensure they are correctly typed with the Mapped annotation, making sure to import the correct types from the library.
3. Ensure that all queries, if any, are updated to use the new 2.0 style of querying, such as using 'select()' instead of 'query()'.
4. Update functions that use 'sessionmaker' to use 'session' instead.
5. Update the code to use async functions and await calls where necessary.

// Pydantic's specific steps ...
1. Update all Pydantic models to use the new syntax.
2. The 'Config' class must be replaced with the 'model_config' attribute.
3. '@validator' must be replaced by '@field_validator'.
4. '@root_validator' must be replaced by '@model_validator'.
5. You must not add a '@dataclass' decorator to your Pydantic models.
6. Add default values to the 'Fields' as they are not 'None' by default.
7. 'BaseSettings' must be imported from 'pydantic_settings' now.

// Instructions used for both packages
1. Implement type hinting for all functions and variables and update old type hinting to ensure they are correct.
2. Ensure there are no missing import statements.
3. Remove any unused imports or variable declarations.
4. Make sure the code works.

Figure 4.3: Chain-of-Thoughts prompt

4.2 Results

In this section, we present and analyze the results achieved by ChatGPT using the three previously defined prompts. First, we present the results for the SQLAlchemy library (Section 4.2.1) and then present the results for the Pydantic library (Section 4.2.2).

4.2.1 SQLAlchemy Results

Table 4.1 presents the results for the migration process of the SQLAlchemy clients using the methodology discussed in Section 4.1. We also present measures for the code before the migration, as a baseline for comparison. In particular, both Migration Coverage and Passing Tests metrics are presented in two forms: *aggregate* and *median*. The aggregate results are obtained by analyzing all projects collectively, as if they constituted a single repository. In contrast, the median results are calculated across the individual outcomes of each of the ten studied repositories. Per-repository values are provided in Appendix A (Tables A.1 and A.2).

Table 4.1: SQLAlchemy Prompt-Based Migration Results

Metrics	**Before**	**Zero-Shot**	**One-Shot**	**Chain-of-Thoughts**
Migration Coverage				
Aggregate (%)	-	9.40	22.30	85.80
Median (%)	-	29.88	87.13	100
Passing Tests				
Aggregate (%)	95.16	0.32	15.08	0.77
Median (%)	100	0	0	0
Compiling Repos (#)	10	7	7	4
Avg. Pylint Score	6.16	5.55	5.94	5.53
Avg. Pyright Errors	45.8	71.2	63.3	69.8

Zero-Shot Prompt: When analyzing the results for the Zero-Shot approach, we can see in Table 4.1 that it did not perform well. Regarding the migration coverage metric, it achieved an aggregate migration coverage of only 9.40% when analyzing all repositories. An example of a migration issue can be seen in Listing 4.1. In it, the `Column` object (referenced in lines 4-7) should have been migrated to the `mapped_column` object, but this change was not performed by ChatGPT.

```
1 class TodoInDB(SQL_BASE):  # type: ignore
2     __tablename__ = "todo"
3
4     id = Column(Integer, primary_key=True, autoincrement=True)
5     key = Column(String(length=128), nullable=False, unique=True)
6     value = Column(String(length=128), nullable=False)
7     done = Column(Boolean, default=False)
```

Listing 4.1: `fastapi-template` non migrated class definition for the Zero-Shot migration

After the migration with the Zero-Shot prompt, 7 out of 10 repositories compiled. Of the 3 repositories that didn't compile, the most recurring error was Python's

`ImportError`. An example can be seen in line 2 of Listing 4.2, where an attempt was made to import the `create_async_engine` method from the wrong module. This method is implemented in `sqlalchemy.ext.asyncio`, but the automatically migrated code attempts to import it from the `sqlalchemy` module. Additionally, in line 4, ChatGPT tried importing a method called `create_async_session`, but this method does not exist. Both cases prevent the application from correctly compiling.

```
1 from pydantic import BaseModel
2 from sqlalchemy import Boolean, Column, Integer, String,
     create_async_engine
3 from sqlalchemy.exc import DatabaseError
4 from sqlalchemy.ext.asyncio import AsyncSession, create_async_session
5 from sqlalchemy.orm import declarative_base, sessionmaker
6 from sqlalchemy.pool import NullPool
```

Listing 4.2: `fastapi-template` import errors for the Zero-Shot migration

When examining the test execution results, Table 4.1 reveals that only an aggregate of 0.32% passed after the migration. This result is worsened when looking at the median of 0%. For the Zero-Shot approach, most failures were due to Python's `AttributeError`, which occurs when an invalid object attribute is accessed. Listing 4.3 provides an example: the `e` variable was accessed on line 5 before the `coroutine` returned by the `get_enforcer` function had finished processing. This issue was introduced by the migration, in which the new version of `get_enforcer` returns a `coroutine`, thus requiring an `await` statement on line 2 to ensure the returned object was ready before access. Consequently, attempting to access an enforce attribute on a `coroutine` results in the following error: `AttributeError: 'coroutine' object has no attribute 'enforce'`.

```
1 def test_update_policy(self):
2     e = self.get_enforcer()
3     example_p = ["mike", "cookie", "eat"]
4
5     self.assertTrue(e.enforce("alice", "data1", "read"))
```

Listing 4.3: `sqlalchemy-adapter` attribute error for the Zero-Shot migration

Ultimately, the Zero-Shot prompt yielded lower quality metrics compared to the original applications. Pylint averaged a score of 5.55, compared to 6.16 before the migration. This decline was mainly a consequence of the errors mentioned previously. Another factor contributing to the poor outcome of the Zero-Shot method was its inability to use Python's typing feature. An example can be seen in Listing 4.4, in which ChatGPT used an `Iterator` instead of an `AsyncIterator` as the return type for the `create_todo_repository` function. Due to such issues, the average number of errors raised by Pyright increased from 45.8 (before the migration) to 71.2 (after the migration).

```python
@asynccontextmanager
async def create_todo_repository() -> Iterator[TodoRepository]:
    async with create_async_session(get_engine(os.getenv("DB_STRING")))
    as session:
        todo_repository = SQLTodoRepository(session)

        try:
            yield todo_repository
        except Exception:
            await session.rollback()
            raise
        finally:
            await session.close()
```

Listing 4.4: `fastapi-template` typing errors for the Zero-Shot migration

One-Shot Prompt: Regarding the results for the One-shot prompt, there was improvement when compared to the Zero-Shot approach. Table 4.1 shows that this method yielded an aggregate migration coverage of 22.30%. With a larger improvement when looking at the median (87.13% vs 29.88%). This indicates that the provided example was enough to considerably raise the amount of code upgraded to the newer library version.

```python
# Before
...
app = FastAPI(debug=True)
templates = Jinja2Templates(directory="templates")


app.add_middleware(
    DebugToolbarMiddleware,
    panels=["panels.ExamplePanel"],
    jinja_loaders=[templates.env.loader],
)

# After
...
class Base(AsyncAttrs, DeclarativeBase):
    pass

class A(Base):
    __tablename__ = "a"

    id: Mapped[int] = mapped_column(primary_key=True)
    data: Mapped[Optional[str]]
    create_date: Mapped[datetime.datetime] = mapped_column(
    server_default=func.now())
    bs: Mapped[List[B]] = relationship()
...
```

Listing 4.5: `fastapi-debug-toolbar` quickstart.py file before and after the One-Shot migration

Of the 10 repositories, 3 didn't compile after the migration using the One-Shot approach, mirroring the results for the Zero-Shot strategy, though different ones didn't compile in this instance. A notable migration issue occurred in both `fastapi-debug-toolbar` and `fastaapi-template` repositories, which resulted in compilation errors for both cases. The LLM deleted the original content of one of the files in the repositories, replacing it with example code provided in the prompt. An example can be seen in Listing 4.5. In it the code seen in lines 3-11 was replaced with the one in lines 15-24. This ended up breaking the application and leaving it in a non-compilable state. This behavior might be attributed to the file not requiring any actual changes for the migration.

The One-Shot migration method outperformed the Zero-Shot approach regarding the aggregate percentage of passing tests (15.08% vs 0.30% respectively). However, the median shows that the results are still not great, preserving a 0% median of passing tests. A significant error observed in two migrations involved the insertion of a code snippet from an example into several repository files, similar to the issue seen in Listing 4.5. This snippet, designed to interact with a local database requiring a specific connection string, led to application execution failures, and consequently, failing tests. Listing 4.6 provides an illustrative example. In all cases, since no local database was running with those specific credentials, the application ended up raising a `DBConectionError`, which stopped it's execution.

```
1 async def main() -> None:
2     engine = create_async_engine("postgresql+asyncpg://scott:
    tiger@localhost/test", echo=True)
3     await test_create_and_drop(engine)
4     await engine.dispose()
5
6
7 asyncio.run(main())
```

Listing 4.6: `alembic_utils` inserted example snippet after the One-Shot migration

The One-Shot method, while outperforming the Zero-Shot prompt in quality metrics, still showed some degradation compared to the original applications. After migration, Pylint scores averaged 5.94, down from 6.16, largely due to unused imports and variable declarations. Pyright errors also increased significantly, averaging 63.3 compared to 45.8 before the migration.

Chain-of-Thoughts Prompt: Looking at the this strategy results, we can see in Table 4.1 that it managed to get the best result for the migration coverage metric. The prompt got an aggregate migration coverage of 85.80%, and a median of 100%. Both values outperform the other two prompts by a considerable margin. In general, it migrated most of the necessary code to consider an application compatible with SQLAlchemy version 2, but was not fully consistent. An example is the `nebulo` repository, in which it did

not correctly migrate the `Column` API to the newer `mapped_column` one, something that it did correctly in all other applications.

While the results for the migration coverage metric were good, this approach produced the worst compilation results of the three, with only 4 applications still compiling after the migration. Among the compilation errors, the `MappedAnnotationError` is a notable one, it is a specific SQLAlchemy v2 error that occurs when an invalid parameter is passed to the `Mapped` type annotation. This error can be seen in Listing 4.7 (line 6), in which a `datetime.datetime` type was passed to the annotation without being previously imported in the file. It is also worth mentioning that the file where this error occurred had its content replaced with the provided example. The same error appeared with the One-Shot prompt for this repository as well, but in a different file.

```
1 class A(Base):
2     __tablename__ = "a"
3
4
5     id: Mapped[int] = mapped_column(primary_key=True)
6     data: Mapped[Optional[str]]
7     create_date: Mapped[datetime.datetime] = mapped_column(
    server_default=func.now())
8     bs: Mapped[List[B]] = relationship()
```

Listing 4.7: `fastapi-debug-toolbar` MappedAnnotationError example after the Chain-of-Thoughts migration

Regarding the percentage of passing tests, the Chain-of-Thoughts prompt performed similarly to the Zero-Shot approach, with only an aggregate of 0.70% of tests passing after the migration, and keeping a median of 0%. The Chain-of-Thoughts prompt experienced instances of database connection issues, where the connection `string` provided in the example was copied to the end of the file, similar to the issue seen in Listing 4.6.

Lastly, the Chain-of-Thoughts strategy yielded results similar to the other prompts when evaluating quality metrics. It averaged a Pylint score of 5.53 (compared to 6.16 before the migration), the lowest among the three strategies. Additionally, it averaged 69.8 errors in the Pyright analysis (compared to 45.8 before the migration), performing better than the Zero-Shot prompt but worse than the One-Shot prompt.

Summary: Relative to the amount code actually migrated, the Chain-of-Thoughts prompt yielded the best results. With an aggregate migration coverage of 85.80%, it outperformed both other prompts by a large margin (Zero-Shot with 9.40% and One-Shot with 22.30%). Both Zero-Shot and One-Shot strategies tied for the number of applications that compiled, with 7 each. Lastly, the One-Shot prompt had the highest aggregate percentage of passing tests, 14.35%, compared to 0.30% and 0.73%, for the Zero-Shot and Chain-of-Thoughts approaches respectively.

4.2.2 Pydantic Results

Table 4.2 presents the results for the migration process of the Pydantic clients using the methodology discussed in Section 4.1. Similarly to how we presented the SQLAlchemy repositories migration results, we also present measures for the code before the migration, as a baseline for comparison. We also present values for aggregate and median measures for both migration coverage and passing tests metrics. Per-repository values are provided in Appendix A (Tables A.1 and A.2).

Table 4.2: Pydantic Prompt-Based Migration Results

Metrics	**Before**	**Zero-Shot**	**One-Shot**	**Chain-of-Thoughts**
Migration Coverage				
Aggregate (%)	-	44.96	55.81	52.34
Median (%)	-	54.55	50	33.33
Passing Tests				
Aggregate (%)	96.14	0	0	2
Median (%)	100	0	0	0
Compiling Repos (#)	10	7	8	6
Avg. Pylint Score	7.31	7.68	7.89	8.13
Avg. Pyright Errors	48.27	52.72	63.3	49.36

Zero-Shot Prompt: The Zero-Shot approach for Pydantic migration achieved an aggregate of 44.96% migration coverage, the lowest among the three strategies for this dataset. However, it achieved a median of 54.55%, being the highest among all prompts. In general, there was no single essential change the LLM consistently failed to make, but some examples can be seen in Listing 4.8. In this example, the `__fields__` attribute should have been migrated to the `model_field` attribute (line 6), and the `update_forward_refs` function was renamed to `model_rebuild` (line 9).

In terms of the number of compiling repositories, 7 out of the 11 repositories

```python
@classmethod
def update_backward_refs(cls):
    """
    force update forward refs of all backward referncing DataBaseModels
    """
    for _, model_field in cls.__fields__.items():
        data_base_model = cls.check_if_subtype({"type": model_field.
    type_})
        if data_base_model:
            data_base_model.update_forward_refs()
```

Listing 4.8: `pydbantic` migration issues for the Zero-Shot Prompt

compiled after migration. Most of the repositories that did not compile encountered migration-specific errors. An example can be seen in Listing 4.9, in which the LLM tried to use a nonexistent "`PydanticFieldInfo`" class (line 1), resulting in a "`Name not defined`" error.

```python
class Field(PydanticFieldInfo):
    def __init__(self, **kwargs: Any):
        supported_config = {"default_factory"}
        field_info_config = {k: kwargs[k] for k in supported_config if k
    in kwargs}
        super().__init__(**field_info_config)
        for k, v in kwargs.items():
            setattr(self, k, v)
```

Listing 4.9: `pydantic-numpy` PydanticFieldError for the Zero-Shot Prompt

No tests passed after the migration with the Zero-Shot prompt for any of the repositories. Most test failures were due to Pydantic-specific issues. For example, in Listing 4.10, a `PydanticUserError` was raised. This error occurs when Pydantic is used in a way not intended by its developers. In this case, a `__modify_schema__` function was declared (line 3), which was deprecated in Pydantic v2. This issue caused all tests in the `pydantic-numpy` repository to fail.

```python
class _BaseDType:
    @classmethod
    def __modify_schema__(cls, field_schema: dict[str, Any]) -> None:
        field_schema.update({"type": cls.__name__})
```

Listing 4.10: `pydantic-numpy` PydanticUserError for the Zero-Shot Prompt

Regarding the quality metrics, the average Pylint score improved slightly, 7.68 from 7.31, suggesting the LLM was able to preserve and improve code quality for this dataset. However, the average number of Pyright errors increased to 52.54 (compared to 48.2 before the migration), indicating the model struggled to correctly apply the new

typing rules of Pydantic v2.

One-Shot Prompt: The One-Shot approach showed similar migration coverage results to Zero-Shot, with an aggregate of 55.81% and a median of 50%. This shows that even though the median is smaller, the One-Shot approach was able to migrate more instances of code that needed migration when compared to the results for the Zero-Shot prompt. However. this contrasts with the SQLAlchemy migration results, where the One-Shot prompt significantly outperformed Zero-Shot. Most of the unmigrated code was similar to that seen with the Zero-Shot strategy, suggesting that the single code example provided for Pydantic was not effective in guiding the model's migration process.

```
1 from typing import Optional, Union
2
3 from pydantic import FieldInfo
4
5 from ormdantic.types import AnyNumber, default_max_length
```

Listing 4.11: `ormdantic` `ImportError` for the One-Shot Prompt

Regarding the number of repositories that successfully compiled after the migration, the One-Shot strategy showed a slight improvement over Zero-Shot, with one additional repository compiling (8 versus 7). Two of the three remaining repositories failed to compile due to errors such as the one illustrated in Listing 4.11 (line 3). This error occurred because the `FieldInfo` class, previously imported from the `pydantic` module, was moved to `pydantic.fields` in Pydantic v2, leading to an `ImportError`.

```
1 @__dataclass_transform__(kw_only_default=True, field_descriptors=(Field
    ,))
2 class ESModelMeta(type):
3     """Abstract ESModel Metaclass
4     This Metaclass ensures that any concrete implementations of ESModel
5     include all necessary definitions, ex. Meta internal class
6     """
7
8 ...
9
10 class ESModel(BaseModel, metaclass=ESModelMeta):
11     id: Optional[str] = Field(default=None)
```

Listing 4.12: `pydastic` `TypeError` for the One-Shot Prompt

Similarly to the Zero-Shot approach, all tests failed after the migration, with Python's `TypeError` being the most frequent error in the One-Shot approach. Listing 4.12 illustrates this issue in the `pydastic` repository. Updating `ESModelMeta` to inherit from the type class instead of `ModelMetaclass` broke Pydantic's internal logic for the `ESModel` class (line 10). This resulted in a `TypeError`, as `ESModelMeta` was no longer compatible as a metaclass for `ESModel`.

The One-Shot and Zero-Shot results for quality metrics were similar. While the average Pylint score increased modestly from 7.68 to 7.89, the number of Pyright errors also increased, from 52.54 to 63.3. This suggests that including an example also did not significantly affect these metrics.

Chain-of-Thoughts (CoT) Prompt: Looking at the results for the Chain-of-Thoughts strategy in Table 4.2, we observe that it yielded comparatively weaker results than the One-Shot approach. Its aggregate score was 52.34% (vs. 55.81%), and its median coverage was also lower, at only 33.33%. Although some repositories showed improvements relative to the previous approaches—such as the `FastAPI-Pydantic-Mongo_Sample_CRUD_API`, which achieved 63.6% migration coverage under the One-Shot approach and 100% under Chain-of-Thoughts—the lower median suggests that this strategy produced a higher number of repositories with limited migration success.

In terms of compilation results, CoT performed worst among the prompting methods. Only 6 out of the 11 applications compiled successfully after the migration. Most failures were caused by `ImportError` exceptions, typically due to incorrect or missing imports. These issues appeared in patterns similar to those observed in the One-Shot approach, but with a higher frequency.

Regarding test outcomes, Chain-of-Thoughts was the only prompting method that achieved non-zero passing test results for the Pydantic migrations. However, performance remained poor: the aggregate percentage of passing tests was only 2%, and the median continued to be 0. The issues were similar to those found in the other prompting methods, such as inconsistencies introduced during the API usage updates.

Finally, CoT produced the strongest results in terms of code quality. The average Pylint score increased to 8.13 (from 7.31 prior to migration), and the number of Pyright errors increased only slightly, with an average of 49.36 (vs. 48.2 before migration). These results indicate that, even though the CoT prompt did not perform well in terms of functional migration, it either preserved or improved the overall maintainability and quality of the code.

Summary: The results for the Pydantic library migration do not clearly present a superior prompt among the three evaluated. In terms of migration coverage, all approaches produced similar outcomes, with the One-Shot method achieving the highest aggregate score (55.81%, compared to 44.96% and 52.34% for Zero-Shot and Chain-of-Thoughts, respectively). The results for the passing tests metric show that the LLM was unable to maintain a fully functional application after the migration, as all strategies yielded a median of 0% passing tests. Despite these limitations, slight improvements were observed regarding the quality metrics. All prompts managed to improve the average Pylint score, with the Chain-of-Thoughts achieving the highest result (8.13 vs 7.31 prior the migration), although the average number of Pyright errors increased in all cases. Finally, the One-Shot approach resulted in the largest number of applications that successfully compiled after migration, with eight repositories executing without errors.

4.3 Threats to Validity

In addition to the threats described in Section 3.4, this first study is also subject to the following validity concerns:

Prompt Design: The results observed in this chapter may be influenced by the specific design of the three prompts evaluated (Zero-Shot, One-Shot, and Chain-of-Thoughts). Although they were selected to represent commonly used prompting strategies, other alternatives—such as Few-Shots or iterative refinements prompts—could have led to different outcomes. Additionally, the example code used in both One-Shot and Chain-of-Thoughts prompts, as well as the step-by-step instructions designed for the latter, may not fully represent the diversity of usage patterns found in real-world applications. Consequently, variations in the structure, quality, or completeness of these parameters could alter the migration results.

Determinism and Single run execution: Another threat relates to the inherent non-deterministic nature of LLMs, which can produce different outputs even when the same prompt is executed multiple times. To mitigate this risk, we set the `temperature` parameter to zero, aiming to increase determinism, and relied exclusively on the first response returned for each migration task, as described in Chapter 3. Even so, running each prompt multiple times or aggregating results across executions might have yielded different migration outcomes, potentially changing the interpretation of the findings.

4.4 Final Remarks

In this chapter, we presented the three prompting strategies used in our first study and discussed the results obtained during the migration of applications that rely on the SQLAlchemy and Pydantic libraries. We described how Zero-Shot, One-Shot, and Chain-of-Thoughts prompts were constructed, executed, and evaluated across all selected repositories. Our results indicate that, although the LLM was able to correctly apply many required migration changes, its performance varied considerably depending on the prompting method, the target repository, and the library being upgraded.

Overall, the One-Shot prompt was the most consistently effective, frequently producing either the best or second-best outcomes across all evaluated metrics. This suggests that enriching or refining the example provided to the model may further enhance migration accuracy. Nevertheless, none of the prompting strategies achieved a flawless migration across all repositories and metrics.

These findings show that prompt-based migration can support developers by accelerating the upgrade process and reducing manual effort, but it is not sufficient to fully automate the activity. Human oversight—particularly for verification, testing, and post-migration corrections—remains necessary to ensure that applications remain functional and reliable after the changes performed by the model.

Chapter 5

Agent-Based Migration

This chapter describes the second study presented in this dissertation. In it, we use an Agent-Based migration strategy, using Github's Copilot Agent Mode [48], to migrate the repositories in both datasets. Section 5.1 describes the prompts and instruction files used in this study, as well as how they were generated. Next, in Section 5.2 we report the metric results for each library. Lastly, Section 5.3 presents threats to validity and Section 5.4 concludes the chapter.

5.1 Methodology

In this study, the migration for each repository was performed in two steps. The first was the creation of a GitHub Copilot instructions file [13]. This file provides the LLM specific instructions, outlining the necessary code modifications for upgrading SQLAlchemy or Pydantic from version 1.x to 2.x. The second step was the migration itself, performed by issuing a prompt to Copilot's Agent Mode, which utilizes the GPT-4o model.

To mitigate threats to validity, we also made two key decisions. First, the migration prompt for each project was executed only once to avoid variability in the LLM's output. Second, whenever the agent requested input or clarification, we provided the standardized response "keep going" to minimize human influence. A migration process was terminated under one of three conditions: the agent confirmed the migration was complete; the agent encountered an unrecoverable error; or the agent entered an infinite loop.

In order to perform the migration, we engineered two distinct prompts. The first, shown in Figure 5.1, was designed to generate the Copilot Instructions file that would guide the subsequent migration. Drawing insights from previous research [2], we provided the model with an example of an already migrated code snippet to improve the quality of the output. The prompt also included explicit directives, such as specifying the target library version and instructing the model to avoid introducing new functionality.

Copilot Instructions Creation Prompt

The Python code in this repository uses the library [SQLAlchemy—Pydantic] with version 1. We will migrate it so that it works with version 2 of [SQLAlchemy—Pydantic]. We must also make the code compatible with python's `asyncio` and use python's `typing` module to add type hints to the code. We must not add extra functionality to the code. Create a `migrate-[sqlalchemy|pydantic].instructions.md` file detailing the migration process and what should be done to achieve this migration. Use the example provided in the `[sqlalchemy|pydantic].py` file and the official documentation as reference to create the instructions file. The instructions file should only reference the [SQLAlchemy—Pydantic] migration, focusing on details needed to upgrade it between versions.

Figure 5.1: Prompt used to generate the Copilot Instructions file

A snippet of an example output of this prompt can be seen in Listing 5.1. In it we can see that the LLM added an initial paragraph describing the task to be done (line 3). In addition, it defined key changes between Pydantic versions and a migration steps section, in which it describes minimal steps necessary to perform the migration. Both these sections are similar to what was done in the Chain-of-Thoughts prompt strategy described in Chapter 4. Although the generated instructions file was slightly different for each repository, most of them had at least these three sections, showing that the LLM consider them important to be used as a reference for the migration process.

The second prompt, presented in Figure 5.2, was used to execute the actual migration. In this prompt, we directed the LLM agent to use the newly created `migrate-[sqlalchemy|pydantic].instructions.md` file as its primary guide for upgrading the repository to either SQLAlchemy or Pydantic v2. This prompt was executed once for each repositories in our datasets. To facilitate a realistic development workflow, we included instructions for managing the environment, such as using `uv` as the package manager and a local Python virtual environment. We also instructed Copilot to generate a TODO list to approach the task systematically. Finally, we provided credentials for a locally running PostgreSQL Docker container for repositories that required a database connection during test execution.

```
# Migration Instructions: Pydantic 1.x to 2.x

This document provides detailed instructions for migrating code from
    Pydantic version 1.x to 2.x. The migration process focuses on
    upgrading the code to be compatible with Pydantic 2.x, ensuring
    compatibility with Python's asyncio, and adding type hints using
    Python's `typing` module. The instructions are based on the example
    provided in the `examples/pydantic.py` file and the official Pydantic
     documentation.

## Key Changes in Pydantic 2.x

1. **Model Validators**:
    - Replace `@validator` with `@field_validator` for field-specific
    validation.
    - Replace `@root_validator` with `@model_validator` for model-wide
    validation.
    - Use the `mode` parameter in `@model_validator` to specify
    validation timing (`"before"` or `"after"`).

...

## Migration Steps

### 1. Update Dependencies
- Ensure that Pydantic 2.x is installed in your environment.
- Update the `requirements.txt` file to specify the new version:
  ```
  pydantic>=2.0
  ```

...
```

Listing 5.1: `djantic` Pydantic Migration Instructions file snippet

Migration Prompt

Using the `migrate-sqlalchemy.instructions.md` instructions file, upgrade this repository so that it works with SQLAlchemy v2.
Use `uv` as the python package and project manager
Use Python's virtual environment for development and testing
Before starting, define a TODO list with what you need to do and then systematically perform each task, without asking me for help.
If you need to use a database connection, use the postgres docker container running locally with the following credentials
username: postgres
password: postgres
database: [REPOSITORY_NAME]

Figure 5.2: Prompt to migrate the repositories

The agent execution followed mostly similar steps as the ones listed in Figure 5.3 for all repositories. Among them, we can see that the LLM cover the basis on what is needed to correctly migrate the application: (1) Set up the local environment, installing

necessary dependencies; (2) Making the actual requested changes; (3) Testing the migration.

Copilot Agent TODOs list

- ✓ Set up Pydantic v2 environment
- ✓ Update validators to Pydantic v2
- ✓ Update model configuration
- ✓ Update JSON handling methods
- ✓ Add modern type hints
- ✓ Test the migration

Figure 5.3: Copilot's generated TODOs list

5.2 Results

In this section, we present and analyze the migration of both library datasets using Github's Copilot Agent Mode powered by GPT-4o. The results for the SQLAlchemy dataset migration are detailed in Section 5.2.1, while the results for the Pydantic migration are detailed in Section 5.2.2.

5.2.1 SQLAlchemy Results

Table 5.1 summarizes the results obtained for the SQLAlchemy library migration using the methodology described in Section 5.1. For comparison purposes, we also report baseline measures collected from the code prior to migration. Both the Migration Coverage and Passing Tests metrics are presented in two forms: *aggregate* and *median*. The aggregate value reflects the results obtained when all repositories are analyzed collectively, as a single combined codebase. Conversely, the median represents the central tendency of the results when considering the individual outcomes of each of the ten evaluated repositories. Per-repository values are provided in Appendix A (Tables A.1 and A.2).

Table 5.1: SQLAlchemy Agent-Based Migration Results

Metrics	**Before Migration**	**After Migration**
Migration Coverage		
Aggregate (%)	-	45.48
Median (%)	-	100
Passing Tests		
Aggregate (%)	95.16	39.75
Median (%)	100	70.83
Compiling Repos (#)	10	8
Avg. Pylint Score	6.16	6.48
Avg. Pyright Errors	45.8	35.6

Migration Coverage: The results for this metric, as presented in Table 5.1, show complementary insights. While the median has a value of 100%, the aggregate measure shows that only 45.48% of the code was actually migrated in the whole dataset. This was mostly due to the performance achieved in the `sqlalchemy-adapter`. This repository contains the highest number of SQLAlchemy usages in the dataset, with 124 instances that require migration. However, Copilot was able to migrate only 9.67% of them. A key factor that hindered this process was a conflict with the `casbin` library. The `sqlalchemy-adapter` project uses version 0.8.1 of `casbin`, which lacks compatibility with Python's `asyncio`—a core requirement to the migration to SQLAlchemy version 2.

Copilot Chat Thought Process

The `casbin` package is installed, but it seems the `async_adapter` module is not available in this version. This could be due to a version mismatch or a missing feature in the installed version of `casbin`.

I will check the `casbin` documentation or source code to confirm the availability of the `async_adapter` module.

The `async_adapter` module is not part of the `casbin` package itself but is provided by external adapters, such as the `casbin_async_sqlalchemy_adapter`. This adapter is specifically designed for `async` operations with SQLAlchemy.

I will install the `casbin_async_sqlalchemy_adapter` package to resolve the missing import issue.

Figure 5.4: Agent reasoning on the `casbin` issue with `asyncio` for the `sqlalchemy-adapter` repository.

As shown in Figure 5.4, the agent indeed identified this compatibility issue. However, it attempted to install the `casbin_async_sqlalchemy_adapter` package to fix the issue, but this created a new problem. The application under migration uses a class called `AsyncAdapter`, which is not present in the installed package. This resulted in a

loop where the agent would repeatedly try to consult the library documentation, fail, and request manual intervention. In line with our methodology, we answered with "keep going" three times, and after observing the agent was in loop, we decided to abort the process. Interestingly, a simpler fix existed: updating the `casbin` library to version 1.23.0, which has `asyncio` support, and would have resolved the issue without the need for a new library.

Passing Tests: Analyzing the Passing Tests results, we observe that it achieved 39.75% of success for the aggregate measures and a higher value for the median 70,83%). Out of the 10 migrated repositories, only two repositories had 100% of passing tests after the migration. Two other repositories experienced test failures due to assertion errors. For instance, the `paracelsus` repository includes a test that verifies whether converting a `Mermaid` type to a `string` produces the output "`True if post is published,nullable`". However, the output was inverted after the migration, resulting in "`nullable,True if post is published`". This indicates that the agent did, in fact, change the code's behavior during the migration.

The remaining repositories showed failing tests due to syntax errors during the migration process. For instance, the `db_to_sqlite` repository faced migration issues related to the way the agent attempted to fix its tests. We observed that 18 (out of 19) failing test cases were actually skipped using the `skipif` decorator. This represented an attempt by the agent to skip tests when a specific library was not installed. However, after manually installing the library, removing the decorator and rerunning the tests, they still failed.

Compiling Repositories: Out of the ten migrated repositories, eight successfully compiled after the migration. The two that failed were `nebulo` and `alembic_utils`. The first failure occurred due to an incomplete migration: the agent stopped midway because of a local virtual environment issue, leaving the application in an unrecoverable state. The second failure was caused by a `CircularImport` error, which occurs when a module attempts to import another module that is only partially loaded.

Quality Metrics: Interestingly, the static analysis metrics suggest an improvement in code quality after the automated migration. The average Pylint score increased from 6.16 to 6.48, indicating that the LLM-generated code adhered more closely to standard Python style and conventions. More significantly, the average number of Pyright type errors decreased from 45.8 to 35.6. This indicates that the agent was able to correctly apply Python's `typing` module in the code, resulting in better type safety, which may reduce errors in future development.

Summary: An analysis of the migration results highlights an interesting pattern. First, a group of five repositories emerged, representing half of our dataset. These projects achieved perfect migration coverage (100%) and high test pass rates (over 80%, with two repositories having 100%). One project, `FastApi-Strawberry-GraphQL-SqlAlchemy-BoilerPlate`, stood out as a perfect migration case: it achieved both 100% migration coverage and a 100% test pass rate, along with an improved Pylint score (from 0 to 3.1) and reduced Pyright errors (from 39 to 32). This shows the agent's capability to successfully complete migrations in particular cases. The other five repositories had test pass rates below 80%. However, this functional failure was not necessarily due to a "lack" in code migration, since three of these five projects had migration coverage scores of 80% or higher. Thus, in these three projects, the agent understood and correctly identified the code requiring update but failed to update the codebase without breaking the application, leading to test failures.

5.2.2 Pydantic Results

Table 5.2 reports the results obtained for the Pydantic library dataset migration. Similarly to what was presented in the previous section, we also report baseline measures before the migration. In addition, both Migration Coverage and Passing Tests metrics are presented as an aggregate measure and as a median. Per-repository values are provided in Appendix A (Tables A.1 and A.2).

Table 5.2: Pydantic Agent-Based Migration Results

Metrics	**Before Migration**	**After Migration**
Migration Coverage		
Aggregate (%)	-	66.41
Median (%)	-	60
Passing Tests		
Aggregate (%)	96.14	22.59
Median (%)	100	0
Compiling Repos (#)	11	9
Avg. Pylint Score	7.31	8.59
Avg. Pyright Errors	48.27	38.45

Migration Coverage: The results in Table 5.2 show substantial variability across the eleven repositories. The aggregate coverage after the migration was 66.41%, while the median was 60%. These values indicates that the agent performed some of the transfor-

mations needed to achieve a complete migration; however, its level of success varied from project to project.

Three repositories achieved 100% migration coverage, and two others exceeded 80%. In contrats, two repositories had a migration coverage of 0%, and three others remained below 60%. Notably, the number of required transformations does not appear to be the determining factor behind these differences. For example, one of the repositories with 0% coverage, `pydantic-sqlalchemy`, required only two changes, but the LLM failed to apply either of them. On the other hand, Copilot's agent mode successfully migrated 95.83% of the repository with the highest number of required updates—the `pydantic-vault` project, which contained 24 migration-relevant instances.

Compared to the SQLAlchemy migrations, the Pydantic dataset revealed a larger number of failures related to local environment setup. Out of the eleven projects in the dataset, four encountered problems setting up the `poetry` library locally. One such example can be seen in Figure 5.5. In it, the LLM asks the user to manually install the library to continue with the migration. Even after answering with "keep going" three times, the agent continued issuing the same request. What aggravates this situation is that `poetry` was already installed and available within the local Python virtual environment—an environment originally created by the agent itself. These repeated cycles eventually caused the agent to stop the migration, resulting in low migration coverage for the affected repositories.

Copilot asking to install Poetry

To proceed, we need to resolve the issue with `poetry` not being available. Please confirm if you have installed `poetry` or if you'd like me to guide you through the installation process. Once `poetry` is set up, we can re-run the tests to validate the changes.

Figure 5.5: Agent asking the user to install the `poetry` library in the `lnurl` repository.

Passing Tests: The results for the Passing Tests metric show that, although the agent was able to apply several of the required transformations, most repositories did not retain functional behavior after the migration. Table 5.2 shows that the aggregate percentage of passing tests after the migration was 22.59%, while the median remained at 0%. This indicates that the majority of repositories experienced failures severe enough to prevent their test suites from executing successfully.

Out of the eleven repositories in the dataset, eight had 0% passing tests after migration, meaning that none of their original tests executed successfully. The remaining three repositories showed partial success, with one achieving 92% passing tests, another 47.5%, and a third 37%. This variability suggests that the agent was able to preserve functionality in certain projects but encountered blocking errors in others.

A closer look at the failed cases shows that many test failures are due to incorrectly migrated API usages. One such example is shown in Figure 5.2. In this case, the agent attempted to migrate the deprecated `class Config` block (line 1)—which, in Pydantic v2, must be replaced by a `model_config` class attribute—but instead simply renamed it to `ModelConfig`. This transformation is syntactically valid Python but semantically incorrect, as Pydantic no longer interprets it as configuration. As a result, this repository produced multiple `PydanticSchemaGenerationError` exceptions, which occur when model definitions are malformed. These errors propagated through the test suite, leading to failures across all tests that relied on Pydantic model validation.

```
1 class ModelConfig:
2     orm_mode = True
3     allow_population_by_field_name = True
4
5     @staticmethod
6     def alias_generator(string: str) -> str:
7         return string[0].lower() + string.title().replace("_", "")[1:]
8
9 PydanticUser = sqlalchemy_to_pydantic(User)
```

Listing 5.2: Incorrect migration of `class Config` in `pydantic-sqlalchemy` during Agent-Based Pydantic migration

Compiling Repositories: The compilation results for the Pydantic migration show that, despite several challenges observed in earlier metrics, the agent was able to produce syntactically valid code for most repositories. Out of the eleven projects in the dataset, nine compiled successfully after the migration. This suggests that, in general, the agent avoided introducing syntax errors or unresolved imports during the transformation process.

The two repositories that failed to compile highlight distinct types of migration-related issues. The first one occurred in a project that relied on Pydantic's `BaseSettings` class. In Pydantic v2, settings management has been extracted into a separate package, `pydantic-settings`, and must be imported explicitly. However, the agent did not apply this transformation: it retained the `BaseSettings` import from the main Pydantic package, resulting in a `ModuleNotFoundError` during startup. This error prevented the application from executing, even though the rest of the migration steps were partially correct.

The second compilation failure was not caused by the code migration itself but rather by the agent's difficulty in setting up the local development environment. In this case, Copilot incorrectly concluded that the repository had been fully migrated, even though the environment configuration step had not been completed successfully. As a

result, the project remained in a non-compilable state.

Quality Metrics: The quality metrics suggest that the agent was generally able to improve both code quality and type correctness after the migration. As shown in Table 5.2, the average Pylint score increased from 7.31 to 8.59, indicating that many of the changes performed by the agent—such as removing unused imports, simplifying code structure, or improving general style—aligned with linting best practices. In parallel, the average number of Pyright errors decreased from 48.27 to 38.45, suggesting that the agent was often able to adjust type hints and model definitions in a way that better conforms to Pydantic v2's typing expectations. Although some repositories still accumulated new issues due to partially applied transformations, the overall trend indicates that, when the migration progresses far enough, Copilot's agent mode tends not only to update the API usage but also to produce code that is statically cleaner and more consistent from a typing perspective.

Summary: The Pydantic migration results reveal a distinct pattern compared to SQLAlchemy. Although the agent applied several transformations correctly, Pydantic migrations proved significantly more fragile: five repositories exceeded 80% migration coverage—including three with 100%—yet functional behavior was rarely preserved, as reflected by a median of 0% passing tests and an aggregate of only 22.59%. High coverage did not guarantee correctness: four of the high-coverage repositories still had 0% passing tests, including two fully migrated ones. This highlights Pydantic's stricter structural requirements, where small mistakes in validators, configurations, or model definitions often lead to schema-generation failures that prevent test execution entirely. Migration was further hindered by environment-setup issues, particularly involving the `poetry` package manager. In four repositories, the agent repeatedly attempted to reinstall `poetry`—even though it was already available—and ultimately aborted the migration prematurely.

5.3 Threats to Validity

In addition to the threats described in Section 3.4, this second study is also subject to the following validity concerns.

Repository Structure Dependence: The agent's performance is sensitive to the repository's structure, tooling choices, and configuration conventions. Projects using custom

environment managers, monorepos, or non-standard directory structures can show different results from ones using standard patterns and tools. Because the methodology did not normalize or modify repository structure prior to migration, variation in layout may have influenced outcomes. This poses a threat to external validity, as the results may not generalize to repositories following different organizational patterns.

Lack of interaction with the agent: The study employed a controlled protocol in which the user responded "keep going" whenever the agent requested clarification. Migration was terminated once the user determined the agent had either finished the migration or reached a state in which it was unable to proceed. Although this constraint was necessary for reproducibility, it introduces a validity threat: an important aspect of real-world usage of LLM-based development agents is the interactive dialogue between the agent and the developer. Restricting this interaction through a static response mechanism may produce results that differ from those obtained in scenarios where the developer actively answers the agent's questions or provides contextual guidance.

5.4 Final Remarks

In this chapter, we presented the Agent-Based migration strategy used in our second study and discussed the results obtained for both datasets—the SQLAlchemy and Pydantic libraries. We described how the instruction files were generated and how they guided Copilot's Agent Mode throughout the migration process. Our findings indicate that, particularly for the SQLAlchemy library, the Agent-Based strategy can achieve near-perfect results in certain repositories. In those cases, the agent successfully executed all required steps: configuring the local environment, updating dependencies, refactoring the code, and ensuring that the test suite remained functional after the migration.

However, the results for the Pydantic library demonstrate that migrations involving libraries with more extensive breaking changes or repositories with more complex environments—such as those relying on custom package managers (e.g., `poetry`)—pose substantial challenges for the agent. These difficulties contributed to the Pydantic dataset yielding a median of 0% passing tests, meaning that most repositories did not retain their original functionality after the migration.

Despite these limitations, the quality metrics show that when the agent is able to progress through the migration workflow, it often generates code that is stylistically cleaner and more type-consistent. Improvements in Pylint scores and reductions in Pyright errors suggest that the agent is capable not only of updating the API usage

but also of modernizing aspects of the codebase.

Overall, these findings reinforce the observations from our first study. Although more powerful and structured than a Prompt-Based approach, an Agent-Based strategy is still not capable of independently completing all migrations. Nonetheless, it meaningfully enhances the developer experience by accelerating the identification and resolution of migration issues that would be more time-consuming and error-prone without such a tool.

Chapter 6

Conclusion

This chapter concludes the dissertation, synthesizing the insights gained from the two empirical studies on automated library migration. We begin in Section 6.1 with an overview of the research context and the dual-strategy approach adopted. Section 6.2 summarizes the key findings obtained from the Prompt-Based and Agent-Based experiments. Building on these results, Section 6.3 presents a comparative analysis. It contrasts the strengths and weaknesses of each methodology regarding migration coverage and functional correctness. Section 6.4 outlines practical guidelines for developers, focusing on environmental standardization and testing strategies. Finally, Section 6.5 suggests potential directions for future research.

6.1 Overview

Software maintenance remains one of the most resource-intensive phases in the software development lifecycle. A significant portion of this effort is driven by the need to keep dependencies up to date, a task often complicated by API evolution. When libraries introduce breaking changes, such as the transition from synchronous to asynchronous paradigms or the enforcement of static typing, developers face the challenge of migrating legacy codebases that are no longer compatible with modern standards. As discussed in previous chapters, traditional automated tools often fall short in dynamic languages like Python, leaving manual refactoring as the predominant solution, regardless of how error-prone it is.

This dissertation investigated the potential of Large Language Models (LLMs) to automate this migration process. We focused on the transition of two Python libraries, SQLAlchemy and Pydantic, from their version 1 to version 2. These libraries were selected not only for their popularity but also because their upgrades represent two distinct classes of migration challenges: SQLAlchemy involves a paradigm change toward asynchronous execution while still maintaining deprecated APIs to avoid breaking changes. On the other

hand, Pydantic imposes strict structural changes in data validation and configuration to its models, resulting in many breaking changes.

To evaluate the limits of current LLM capabilities, we designed and implemented two distinct migration studies. The first, detailed in Chapter 4, explored a Prompt-Based approach, assessing how varying prompts (Zero-Shot, One-Shot, and Chain-of-Thoughts) affect the model's ability to correctly migrate the library usages. The second, detailed in Chapter 5, investigated an Agent-Based approach using GitHub Copilot's Agent Mode. This strategy introduced a higher level of autonomy, allowing the LLM to interact with the local development environment, execute tests, and autonomously refine its output.

Our findings indicate that while LLMs can effectively automate the transformation of deprecated code, achieving high migration coverage across different strategies, the preservation of software functionality remains a significant challenge. We observed that Prompt-Based approaches, despite achieving high coverage in specific contexts, consistently failed to maintain application behavior, resulting in a median of 0% passing tests across both datasets. In contrast, the Agent-Based strategy demonstrated a superior potential for functional correctness, achieving instances of 100% passing tests and migration coverage in the SQLAlchemy dataset. However, this success was not uniform; the migration of the Pydantic library, characterized by strict structural breaking changes and environment complexities, proved resistant to both strategies, highlighting that the nature of the API evolution influences the effectiveness of LLM-based migration.

6.2 Summary of Findings

This section synthesizes the empirical results obtained from the two studies conducted in this dissertation. Section 6.2.1 presents the findings for the Prompt-Based strategy, while Section 6.2.2 presents findings for the Agent-Based strategy.

6.2.1 Prompt-Based Strategy

Our first study revealed that while LLMs can effectively perform syntactic transformations, their success is highly sensitive to the prompting strategy employed. In the context of SQLAlchemy, the Chain-of-Thoughts prompt achieved the highest Aggregate Migration Coverage (85.80%), significantly outperforming the Zero-Shot approach (9.40%).

However, this increase in coverage came at the cost of stability: the Chain-of-Thoughts strategy produced the lowest number of compiling repositories (4 out of 10), often introducing hallucinations such as non-existent methods (e.g., `create_async_session`) or incorrect import statements that prevented execution.

Despite the ability to generate syntactically plausible code, the Prompt-Based approach consistently failed to preserve the functional behavior of the applications. Across both libraries and all three prompting strategies, the median Percentage of Passing Tests remained at 0%. This failure can be justified by the model's inability to manage the global application context or verify its own output. Without execution feedback, the model frequently introduced logical errors, such as accessing attributes on unawaited coroutines. While syntactically correct, these changes ended up causing runtime failures in the test suites.

Comparatively, the One-Shot prompting strategy emerged as the most balanced approach in this study. This observation holds true for the SQLAlchemy dataset, where the One-Shot strategy was the only approach to achieve an Aggregate Percentage of Passing Tests above 0% (15.08%), compared to less than 1% for both Zero-Shot and Chain-of-Thoughts. Similarly, for the Pydantic library, it achieved the highest Aggregate Migration Coverage (55.81%) and the highest number of compiling repositories (8 out of 11). This suggests that providing a single, clear example is often more effective for structural migrations than complex step-by-step instructions.

6.2.2 Agent-Based Strategy

The introduction of an Agent-Based workflow in our second study marked a significant improvement in the functional correctness of migrations, particularly for the SQLAlchemy dataset. Unlike the Prompt-Based approach, the agent achieved a median Percentage of Passing Tests of 70.83%, with two repositories maintaining 100% of their passing tests after migration. This result validates that the ability to interact with the environment, such as executing tests and reading error logs, allows the LLM to iteratively correct the logical faults that previously led to zero percent pass rates.

However, the efficacy of the agent proved to be severely limited by the complexity of the execution environment. We observed that the agent frequently struggled with package management tasks, getting trapped in loops while attempting to install dependencies or configure tools like `poetry`. In the case of the `sqlalchemy-adapter` repository, despite correctly identifying a compatibility issue with the `casbin` library, the agent failed to resolve it due to a lack of understanding of the local environment context. These oper-

ational failures acted as a bottleneck, preventing the agent from attempting to migrate the actual code in several instances.

Finally, the results from the Pydantic migration highlight a critical distinction between code coverage and semantic correctness. While the agent achieved a median Migration Coverage of 60%, with three repositories reaching 100%, the median Percentage of Passing Tests remained at 0%. This discrepancy indicates that for libraries with strict structural constraints, high coverage metrics can be misleading. The agent successfully applied syntax changes (e.g., renaming classes) but often failed to grasp the semantic implications of breaking changes, such as the new strict validation logic required by Pydantic version 2.

6.3 Comparative Analysis

This section contrasts the findings from the two studies to identify the determining factors for migration success. In Section 6.3.1, we analyze how the level of autonomy (Prompt vs. Agent) influences functional correctness. Section 6.3.2 discusses the impact of the nature of API changes (SQLAlchemy vs. Pydantic), while Section 6.3.3 examines the trade-offs between environmental context and stability.

6.3.1 Functional Correctness and Autonomy

A defining distinction between the two strategies lies in the gap between generating syntactically valid code and ensuring functional correctness. Our results from Chapter 4 demonstrate that Prompt-Based strategies can often produce code that resembles the target version. For instance, the Chain-of-Thoughts prompt achieved a high Migration Coverage of 85.80% for SQLAlchemy. However, these strategies consistently fail to preserve application behavior. Across both libraries and all three prompting techniques, the median Percentage of Passing Tests remained at 0%. This highlights a fundamental limitation of static generation. Without execution feedback, the model cannot verify its assumptions, which leads to logical errors that cause runtime failures, even if the code appears syntactically correct.

In contrast, the Agent-Based strategy introduced in Chapter 5 demonstrated a superior potential for recovering functionality. By using an iterative feedback loop, the

agent performs tests, reads error logs, and attempts to correct them. Using this approach, it achieved a median of 70.83% passing tests for the SQLAlchemy dataset, with two repositories reaching 100% functional correctness. While these results represent an improvement over the static approach, they also highlight that giving full autonomy to the model is not a silver bullet. Even with the ability to self-correct, the agent was unable to achieve a median greater than 0% for the Pydantic dataset. This indicates that the complexity of certain migrations exceeds the current capabilities of the model.

6.3.2 The Impact of Breaking Change Complexity

The difference in results between the two target libraries suggests that the nature of the breaking change is a critical variable in automated migration. The transition from SQLAlchemy 1 to 2 involves a paradigm change, primarily moving from synchronous to asynchronous execution. This is coupled with direct syntactic replacements, such as changing `Column` to `mapped_column`. Our data shows that LLMs are relatively effective at handling these patterns. Consequently, the Agent-Based approach achieved instances of 100% Migration Coverage and passing tests for this library, effectively translating the code from one paradigm to another.

However, the migration of Pydantic proved significantly more resistant to automation. The migration from version 1 to 2 involves many changes in the API syntax, with methods and classes having been renamed or removed. Examples include the replacement of the `Config` class with `model_config` attributes and changes in the `@validator` decorator. These modifications require a semantic understanding of the library's internal logic rather than just pattern matching. As evidenced in Chapter 5, although the agent achieved high coverage (median of 60%), the median percentage of passing tests remained at 0%. Errors such as `PydanticSchemaGenerationError` indicate that the model struggled to construct valid internal schemas, even if it updated the syntax. This suggests that structural breaking changes pose a higher barrier to current LLM-based migration tools than paradigm changes.

6.3.3 Environmental vs. Contextual Limitations

Finally, our comparative analysis reveals a trade-off between the LLM's ability to understand application context and its ability to maintain a stable development environment. Prompt-Based approaches operate in isolation. They lack knowledge of the broader project structure, which leads to "hallucinated" imports or issues caused by the lack of context of the application as a whole. An example is the misplacement of `create_async_engine` in Chapter 4. However, this isolation also prevents them from having environmental failures. The prompt simply generates text and does not interact with the system.

The Agent-Based approach gains the ability to understand the global context. This is reflected in the reduction of Pyright errors, which dropped from 45.8 to 35.6 in the SQLAlchemy dataset. However, this approach introduces a new point of failure: the execution environment. We observed multiple instances where the agent abandoned the migration due to loops in package management, such as repeatedly trying to install `poetry` or resolve `casbin` dependencies. This failure was not due to an inability to write code, but rather an inability to manage the development environment. This suggests that as migration tools become more autonomous, their robustness becomes increasingly dependent on their ability to accurately interpret and manipulate the local development environment, as well as implement the proposed changes in the code itself.

6.4 Practical Guidelines for LLM-Assisted Migration

Beyond analyzing the performance of specific models, our findings offer actionable insights for developers and organizations aiming to adopt LLMs for software maintenance. In Section 6.4.1, we discuss the necessity of stabilizing the execution environment to prevent operational failures. Section 6.4.2 emphasizes the critical role of robust verification mechanisms as a prerequisite for autonomy. Finally, Section 6.4.3 outlines the need for targeted human intervention when dealing with complex structural changes.

6.4.1 Stabilizing the Execution Environment

Our results from the Agent-Based study indicate that the primary bottleneck for the LLM is in the environment setup and code execution, not in the writing of correct code itself. The agent frequently failed due to environmental inconsistencies and ambiguity in project structure. Therefore, a critical prerequisite for using LLM agents is the usage of well-defined coding standards and explicit environment configurations. This goes beyond the use of specific tools like Docker. It requires the presence of comprehensive dependency files, such as explicit declarations of the Python version and libraries used in the project. We observed that many repositories lacked these definitions, forcing the agent to guess the execution context. By adopting standard architectural patterns and explicitly documenting dependencies (e.g., via `pyproject.toml` or lock files), developers reduce the "implicit knowledge" required to run the project. This allows the agent to focus its reasoning capabilities exclusively on code transformation rather than environment troubleshooting.

6.4.2 Ensuring Test Suite Robustness

The contrast in success rates between our Prompt-Based and Agent-Based studies highlights that an automated migration is only as reliable as its verification suite. The agent's ability to achieve nearly 40% passing tests in SQLAlchemy was driven entirely by the feedback loop provided by existing tests. Consequently, developers must ensure that their test suites are comprehensive and deterministic. They must also be decoupled from external factors before attempting automation. If a test suite is flaky or depends on unstable external services, the agent can interpret a transient failure as a code error. This leads to fixes being made due to the wrong reason, such as the model modifying correct code to satisfy a broken test, instead of the opposite. Thus, a robust, runnable test suite acts as an essential functional correctness validation tool for the LLM. It allows the Agent to ensure that, after each modification, the code still behaves the same as it did previously.

6.4.3 Choosing the Right strategy

Finally, our comparison between SQLAlchemy and Pydantic suggests that developers must select their strategy based on the nature of the breaking changes. For migrations involving paradigm changes or syntactic replacements, such as `asyncio` adoption, the Agent-Based strategy showed high efficacy. However, for structural changes that alter the API of a library, it was not enough. In these scenarios, developers should not rely solely on the model's training data. Instead, they must actively intervene by providing specific examples or curated migration guides. As observed in our Prompt-Based study, providing a single relevant example significantly improved compilation rates for Pydantic. This implies that for complex migrations, the developer's role shifts. It moves from writing code to giving the LLM sufficient context that allows it to correctly understand the changes between the old and new library versions.

6.5 Future Work

While this dissertation demonstrates the potential of LLMs to automate library migrations, it also uncovers several avenues for further investigation. Future research could expand on the findings presented here in the following directions:

Context-Aware Migration using Retrieval-Augmented Generation (RAG): Our Prompt-Based study showed that a single static example (One-Shot) significantly improved results compared to generic instructions. Future work could implement RAG techniques to dynamically fetch migration examples relevant to the specific file being processed. By retrieving usage patterns from the library's documentation or external repositories, the model could access a broader and more specific knowledge base, potentially mitigating the structural hallucinations observed in the Pydantic migration.

Specialized Multi-Agent Architectures: A major bottleneck identified in our Agent-Based study was the monolithic nature of the agent, which struggled to balance environment management with code refactoring. Future research could explore a multi-agent system where responsibilities are decoupled. For instance, a dedicated "DevOps Agent" could handle dependencies and containerization, while a "Developer Agent" focuses solely on code transformation, and a "QA Agent" manages test execution and code validation.

This specialization could prevent the infinite loops observed during package installation and improve overall robustness.

Cross-Language Generalizability: This study focused exclusively on the Python ecosystem. Future work could investigate whether these findings hold true for statically typed languages such as Java or Go. A strict compiler might provide more precise feedback to the agent than the Python interpreter, potentially accelerating the self-correction loop. Conversely, strict typing might introduce a higher barrier to initial compilation, altering the trade-offs between Prompt-Based and Agent-Based strategies.

Human-in-the-Loop: Finally, while this dissertation focused on fully automated strategies with minimal intervention, real-world migrations are rarely hands-off. Future studies could evaluate semi-automated workflows where the developer interacts with the agent to resolve ambiguities. Additionally, a cost-benefit analysis measuring token consumption and execution time against the effort of manual migration would provide a practical assessment of the economic viability of LLM-based maintenance in industrial settings.

Breaking Change Characterization: Our results suggest that the nature of API changes significantly influences LLM migration performance, with structural breaking changes (as observed in Pydantic) posing a higher barrier than paradigm shifts (as in SQLAlchemy). Future work could systematically characterize breaking changes in widely-adopted Python libraries and investigate how their type, frequency, and severity correlate with LLM migration success rates [5]. This could guide the development of migration strategies tailored to specific breaking change patterns.

References

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2nd edition, 2006.

[2] Aylton Almeida, Laerte Xavier, and Marco Tulio Valente. Automatic library migration using large language models: First results. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '24, page 427–433. ACM, October 2024. doi: 10.1145/3674805.3690746. URL http://dx.doi.org/10.1145/3674805.3690746.

[3] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. The evolution of project inter-dependencies in a software ecosystem: the case of apache. *IEEE Transactions on Software Engineering*, 41(3):237–265, 2015.

[4] Aline Brito, Marco Tulio Valente, Laerte Xavier, and Andre Hora. You broke my code: Understanding the motivations for breaking changes in APIs. *Empirical Software Engineering*, 25:1458–1492, 2020.

[5] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. On the use of replacement messages in api deprecation: An empirical study. *Journal of Systems and Software*, 137:306–321, 2018.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 1877–1901, 2020.

[7] Federico Cassano, Ryan Rogers, and Robert Dyer. A large-scale empirical study of python 3 features. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*, pages 468–479. ACM, 2022. doi: 10.1145/3524842.3528456.

[8] Bruce Collie, Philip Ginsbach, Jackson Woodruff, Ajitha Rajan, and Michael F. P. O'Boyle. M3: Semantic api migrations. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 90–102. IEEE, 2020. doi: 10.1145/3324884.3416618.

[9] Blake E Cossette and Robert J Walker. Seeking the ground truth: a retroactive study on the effectiveness of migration tools. In *Proceedings of the 20th International Symposium on Foundations of Software Engineering*, pages 1–11. ACM, 2012.

[10] Joel Cox, Eric Bouwers, Marko Van Eekelen, and Joost Visser. Measuring dependency freshness in software systems. In *Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE)*, volume 2, pages 109–118. IEEE, 2015. doi: 10.1109/ICSE.2015.140.

[11] Danny Dig and Ralph Johnson. The role of refactorings in api evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, pages 389–398. IEEE, 2005. doi: 10.1109/ICSM.2005.90.

[12] Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: Quantifying detectable bugs in javascript. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, pages 758–769. IEEE, 2017.

[13] GitHub Copilot Documentation. Add repository instructions — github copilot. Online; accessed 2025-12-02. URL https://docs.github.com/pt/copilot/how-tos/configure-custom-instructions/add-repository-instructions?tool=vscode. Accessed: 2025-12-02.

[14] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. How do developers react to API evolution? the Pharo ecosystem case. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 251–260, 2015.

[15] Daqing Hou and Xiaojia Yao. Exploring the intent behind api evolution: A case study. In *Working Conference on Reverse Engineering (WCRE)*, pages 131–140, 2011.

[16] Md Mohayeminul Islam, Ajay Kumar Jha, May Mahmoud, Ildar Akhmetov, and Sarah Nadi. An empirical study of python library migration using large language models, 2025. URL https://arxiv.org/abs/2504.13272.

[17] Mohayeminul Islam, Ajay Kumar Jha, Ildar Akhmetov, and Sarah Nadi. Characterizing python library migrations. *Proceedings of the ACM on Software Engineering*, 1(FSE):92–114, July 2024. ISSN 2994-970X. doi: 10.1145/3643731. URL http://dx.doi.org/10.1145/3643731.

[18] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration. *Empirical Software Engineering*, 23:384–417, 2018.

[19] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. A systematic review of API evolution literature. *ACM Computing Surveys*, 54(8):1–36, 2021.

[20] Chengwei Lyu, Bihuan Chen, Kaifeng Luo, and Lin Zhu. Pycompat: A static analysis tool for detecting incompatibility between python library versions. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 662–665. ACM, 2021.

[21] Brian A. Malloy and James F. Power. Quantifying the transition from python 2 to 3: An empirical study of python applications. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 314–323. IEEE, 2017.

[22] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK, 2008.

[23] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004. doi: 10.1109/TSE.2004.1265817.

[24] Microsoft. Pyright — static type checker for python. https://github.com/microsoft/pyright, 2025. Accessed: March 2025.

[25] Stoyan Nikolov, Daniele Codecasa, Anna Sjovall, Maxim Tabachnyk, Satish Chandra, Siddharth Taneja, and Celal Ziftci. How is google using ai for internal code migrations?, 2025. URL https://arxiv.org/abs/2501.06972.

[26] Bob Nystrom. What color is your function? https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/, 2015. Accessed: 2025-03-10.

[27] William F Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[28] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Annual Technical Conference, General Track*, pages 199–212, 1999.

[29] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[30] Aske Plaat, Max van Duijn, Niki van Stein, Mike Preuss, Peter van der Putten, and Kees Joost Batenburg. Agentic large language models, a survey, 2025. URL https://arxiv.org/abs/2503.23037.

[31] Alexander Pucho, Alexandre Ferreira, Elder Cirilo, and Bruno Cafeo. Refactoring python code with llm-based multi-agent systems: An empirical study in ml software projects. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software*, pages 678–684, Porto Alegre, RS, Brasil, 2025. SBC. doi: 10.5753/sbes.2025.11033. URL https://sol.sbc.org.br/index.php/sbes/article/view/37044.

[32] Pydantic Developers. Pydantic — data validation and settings management using python type hints. https://github.com/pydantic/pydantic, 2025. Accessed: March 2025.

[33] Pylint Developers. Pylint — static code analysis tool for python. https://github.com/pylint-dev/pylint, 2025. Accessed: March 2025.

[34] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 36–45, 2020.

[35] Matthew Renze and Erhan Guven. The effect of sampling temperature on problem solving in large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 2871–2885. Association for Computational Linguistics, 2024. URL https://aclanthology.org/2024.findings-emnlp.432/.

[36] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. A systematic survey of prompt engineering in large language models: Techniques and applications, 2024.

[37] Maria Salama, Rami Bahsoon, and Patricia Lago. Stability in software engineering: Survey of the state-of-the-art and research directions. *IEEE Transactions on Software Engineering*, 47(7):1468–1510, 2019.

[38] Michael Salib. Starkiller: A static type inferencer and compiler for python. Master's thesis, Massachusetts Institute of Technology, 2004.

[39] Max Schäfer, Sarah Nadi, Amir Aryani, and Frank Tip. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527*, 2023.

[40] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, 2024.

[41] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. An analysis of the automatic bug fixing performance of ChatGPT. In *IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 23–30, 2023.

[42] SQLAlchemy Authors. Sqlalchemy — the python sql toolkit and object relational mapper. https://github.com/sqlalchemy/sqlalchemy, 2025. Accessed: March 2025.

[43] Haoye Tian, Weiqi Lu, Tiezhu Li, Xunzhu Tang, Cheung Shing, Jacques Klein, and Tegawendé F Bissyandé. Is chatgpt a good code assistant? a preliminary study. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1–13. IEEE, 2023.

[44] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. Towards automating code review activities. In *International Conference on Software Engineering (ICSE)*, pages 163–174, 2021.

[45] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *International Conferenceon Software Engineering (ICSE)*, pages 2291–2302, 2022.

[46] Alexi Turcotte, Mojan Keshani, A. Molnar, S. Obral, and Raffi Khatchadourian. Automatic migration from synchronous to asynchronous javascript apis. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 68–80. ACM, 2021.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008, 2017.

[48] Visual Studio Code Team. Introducing copilot agent mode. Online blog post, February 2025. URL https://code.visualstudio.com/blogs/2025/02/24/introducing-copilot-agent-mode. Accessed: 2025-12-02.

[49] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in java projects. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45, 2020.

[50] Louis Wasserman. Scaling automated api migrations. In *Proceedings of the 2013 ACM workshop on Refactoring tools*, pages 1–4, 2013.

[51] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[52] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for highly concurrent server applications. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 230–243. ACM, 2001.

[53] Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1–12. IEEE, 2023.

[54] Xu Xu, Christopher Collier, and Ryan O'Hayon. Python: The full monty - a study of dynamic features in python code. In *Proceedings of the 2016 ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–10, 2016.

[55] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *IEEE Transactions on Software Engineering*, 38(2):531–568, 2012.

[56] Yuqi Zhu, Jia Li, Ge Li, YunFei Zhao, Jia Li, Zhi Jin, and Hong Mei. Hot or cold? adaptive temperature sampling for code generation with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 437–445, 2024. doi: 10.1609/aaai.v38i1.27798.

[57] Celal Ziftci, Stoyan Nikolov, Anna Sjövall, Bo Kim, Daniele Codecasa, and Max Kim. *Migrating Code At Scale With LLMs At Google*, page 162–173. Association for Computing Machinery, New York, NY, USA, 2025. ISBN 9798400712760. URL https://doi.org/10.1145/3696630.3728542.

[58] Celal Ziftci, Stoyan Nikolov, Anna Sjövall, Bo Kim, Daniele Codecasa, and Max Kim. Migrating code at scale with llms at google. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, FSE Companion '25, page 162–173. ACM, June 2025. doi: 10.1145/3696630.3728542. URL http://dx.doi.org/10.1145/3696630.3728542.

Appendix A

Detailed Results by Repository

This appendix presents the per-repository results for Migration Coverage and Passing Tests across all migration strategies evaluated in this dissertation. The values complement the aggregate and median figures reported in the main text by providing individual repository-level data, allowing readers to assess the full distribution of results.

Table A.1: Migration Coverage per repository and migration strategy.

Repository	**Zero-Shot(%)**	**One-Shot(%)**	**CoT(%)**	**Agent(%)**
SQLAlchemy				
`sqlalchemy-adapter`	3.2	14.5	100	9.7
`db_to_sqlite`	100	100	100	100
`alembic_utils`	50	100	100	100
`Fast-Api-example`	100	100	100	100
`fastapi-debug-toolbar`	33	100	100	83.3
`FastApi-Straw[...]-BoilerPlate`	28.5	93	93	100
`fastapi-template`	31.2	81.2	81.2	100
`graphene-sqlalchemy-filter`	5	0	100	81.7
`nebulo`	10	10	10	46.4
`paracelsus`	0	0	66.7	100
Aggregate	9.40	22.30	85.8	45.48
Median	29.9	87.1	100	100
Pydantic				
`djantic`	50	50	80	60
`FastAPI-Pydantic[...]_CRUD_API`	54.5	63.6	100	81.8
`finstruments`	60	60	60	40
`mongox`	88.9	88.9	33.3	55.6
`ormdantic`	50	50	0	0
`pydantic-sqlalchemy`	100	20	20	0
`pydantic-vault`	8.3	95.8	95.8	95.8
`pydastic`	85.7	85.7	71.4	100
`pydbantic`	0	0	6.2	100
`lnurl`	77.3	40.9	18.2	36.4
`pydantic-numpy`	20	20	20	100
Aggregate	44.96	55.81	52.34	66.41
Median	54.55	50	33.3	60

Table A.2: Passing Tests per repository and migration strategy.

Repository	Before(%)	Zero-Shot(%)	One-Shot(%)	CoT(%)	Agent(%)
SQLAlchemy					
sqlalchemy-adapter	100	0	0	0	0
db_to_sqlite	68	0	0	0	5.3
alembic_utils	100	0	0	2	0
Fast-Api-example	100	0	0	0	91.7
fastapi-debug-toolbar	91	0	0	0	100
FastApi-Straw[...]BoilerPlate	100	0	0	0	100
fastapi-template	100	25	25	25	50
graphene-sqlalchemy-filter	85	0	81	0	93.8
nebulo	100	0	0	0	0
paracelsus	100	0	36	0	95.5
Aggregate	95.16	0.32	15.08	0.77	39.75
Median	100	0	0	0	70.8
Pydantic					
djantic	56.2	0	0	0	0
FastAPI-Pydantic[...]_CRUD_API	100	0	0	14	0
finstruments	100	0	0	0	92.2
mongox	100	0	0	0	0
ormdantic	100	0	0	0	0
pydantic-sqlalchemy	100	0	0	0	0
pydantic-vault	100	0	0	6.9	37.2
pydastic	100	0	0	0	0
pydbantic	100	0	0	0	0
lnurl	100	0	0	0	0
pydantic-numpy	100	0	0	0	47.5
Aggregate	96	0	0	2	22.59
Median	100	0	0	0	0