UNIVERSITY OF PASSAU

MASTER'S THESIS
MASTERARBEIT

A Reproducible Evaluation Framework for AI-Driven Angular Migrations

Author / Autor:
Samuel MALLET

Matriculation Number / Matrikelnummer:
118278

First Examiner / Erstprüfer:
Prof. Dr. Harald KOSCH

Second Examiner / Zweitprüfer:
Prof. Dr. Michael GRANITZER

A thesis submitted in fulfillment of the requirements for the degree of Master of Science in Computer Science

Masterarbeit zur Erlangung des akademischen Grades Master of Science in Computer Science

in the / am

Chair of Data Science (Lehrstuhl für Data Science)
Faculty of Computer Science and Mathematics (Fakultät für Informatik und Mathematik)

Submission Date / Abgabedatum: October 13, 2025

Abstract

Samuel MALLET

A Reproducible Evaluation Framework for AI-Driven Angular Migrations

Software companies must regularly upgrade their frameworks to stay secure and maintainable, but doing this manually is slow, expensive, and error-prone. AI coding assistants powered by large language models (LLMs) promise to automate these upgrades, but we lack reliable ways to measure how well they actually work on real projects. Most existing benchmarks focus on backend languages like Python or Java, leaving frontend frameworks like Angular largely unexamined.

This thesis introduces BenchMAC, a benchmark designed specifically to evaluate automated Angular framework upgrades. BenchMAC provides real codebases, standardized testing environments, and clear success metrics. Each system being tested attempts to migrate an Angular project from one major version to the next (for example, from version 11 to 12), producing a patch file that captures all necessary code changes. We then apply these patches in isolated Docker containers and measure whether the upgraded project successfully installs dependencies and builds for production.

We evaluated 19 different AI systems and one rule-based tool (Angular's built-in migration command) across 9 migration tasks. Six AI systems achieved perfect scores, successfully building all 9 projects. However, this came at a cost: while the rule-based tool completed each migration in 2 commands at zero API cost, successful AI agents required 15-50 LLM calls on average and cost between $0.03 to $0.50 per migration. Surprisingly, we found no relationship between an AI system's price and its performance. We also identified two common failure patterns: some agents got stuck in infinite loops repeating the same commands, while others submitted empty solutions after failing to properly set up the development environment.

BenchMAC is released as open source, including the dataset, evaluation code, and all experimental results, enabling future research on AI-driven software modernization.

Acknowledgements

This thesis is the result of a collaboration between Onepoint, INSA Lyon, and the University of Passau. I would like to thank everyone who supported this project.

I am especially grateful to my academic supervisor at INSA Lyon, Professor Elöd Egyed-Zsigmond. Thank you for your continuous guidance since my first internship. Your support was essential throughout this Master's thesis, culminating in our successful publication at the CORIA-TALN 2025 conference.

I thank the company Onepoint for the welcoming and supportive environment. I would like to acknowledge Benoit Deléglise for introducing me to the project and for his mentorship and the autonomy he provided. I also thank Joe El Khoury for his role as a research mentor; his expertise and critical feedback were invaluable, particularly for our collaboration on the CORIA-TALN paper.

My gratitude also extends to Rahul Ramsaha and the entire WeTransform and OneMigration teams for the collaborative atmosphere.

Finally, I thank the coordinators of the double-degree program for their support, especially Professor Dr. Lionel Brunie, Ophélie Coueffe, Professor Dr. Harald Kosch, and Natalia Lucari.

Contents

Abstract iii

Acknowledgements v

1 Introduction 1

2 Background & Related Work 5

2.1 AI for Coding . . . 5

2.1.1 From Autocomplete to Agents: A Historical Perspective . . . 5

2.2 Benchmarks in the generative AI era . . . 6

2.2.1 Defining Benchmarks and Their Fundamental Role . . . 6

2.2.2 The Evolution and Proliferation of LLM Benchmarks . . . 6

2.2.3 Goals, Benefits, and Methodological Diversity . . . 7

2.2.4 The Benchmark Lifecycle and Saturation Crisis . . . 7

2.2.5 Critical Challenges and Systemic Flaws . . . 8

Data Contamination . . . 8

Gaming and Goodhart's Law . . . 8

Evaluation Reliability and Reproducibility . . . 9

Gap Between Benchmark Performance and Real-World Utility . 9

2.2.6 Emerging Solutions and Future Directions . . . 9

Dynamic and Contamination-Resistant Benchmarks . . . 9

Human Preference and Multi-Modal Evaluation . . . 10

2.2.7 Implications for the Field . . . 10

2.3 Code Generation Benchmarks . . . 10

2.3.1 From Functions to Repositories: A Historical Progression . . . 11

Function-Level Benchmarks . . . 11

Improving Coverage and Freshness . . . 11

Repository-Level Benchmarks . . . 11

2.3.2 Execution-based Evaluation . . . 12

2.3.3 Expanding the Scope of Tasks . . . 13

2.3.4 Persistent Challenges . . . 13

2.3.5 Lessons for Benchmark Design . . . 13

2.3.6 Bridge to Migration Benchmarks . . . 14

2.4 Benchmarks in code transformation . . . 14

2.4.1 Code translation . . . 14

Introduction and Scope . . . 14

Industrial Context and Motivations . . . 15

Traditional Approaches and Limitations . . . 15

Neural and LLM-Based Translation . . . 16

Evaluation Challenges and Evolution . . . 17

Current Benchmarks and Metrics . . . 18

Remaining Gaps and Limitations . . . 19

Bridge to Framework Migration . . . 19

2.4.2 Version Upgrades: A Critical and Evolving Challenge in Software Maintenance . . . 20
The Industrial Imperative for Modernization . . . 20
Automating Upgrades: From Rule-Based Tools to LLM-Powered Transformation . . . 20
Benchmarking the Evolution: A Maturing Evaluation Landscape 21
Identifying the Research Gap and Positioning BenchMAC . . . 22

3 Methodology **23**
3.1 Guiding Principles and Overall Design . . . 23
3.1.1 The Unified Diff as a Standard Interface . . . 24
3.2 Dataset Curation . . . 24
3.2.1 Instance Definition . . . 24
3.2.2 Curation Principles . . . 25
Reproducibility Over Size . . . 25
Open-Source and Licensing . . . 25
Application-Centric Focus . . . 25
Quality Signals . . . 25
Modern Angular Scope . . . 25
Sandboxing Constraints . . . 26
The "Green Baseline" Principle . . . 26
Multiple Instances Per Repository . . . 26
Silver Patches for Harness Validation . . . 26
3.2.3 The BenchMAC v1.0 Dataset . . . 27
3.3 Reproducible Environments . . . 27
3.3.1 Rationale for Using Docker . . . 27
3.3.2 Environment Configuration Details . . . 28
Repository Integration . . . 28
Node.js Runtime and Package Manager . . . 28
System Packages and Environment Variables . . . 28
3.3.3 Enforcing Strict Reproducibility . . . 29
3.4 Systems Under Test (SUTs) . . . 29
3.4.1 Choice of Baseline SUTs . . . 30
3.5 Stage 1: Patch Generation . . . 30
3.5.1 Pipeline for the AI Agent (Mini-SWE-Agent) . . . 31
3.5.2 Pipeline for the Rule-Based Baseline (Angular Schematics) . . . 32
3.6 Stage 2: Patch Evaluation . . . 32
3.6.1 Evaluation Pipeline . . . 33
3.6.2 Canonical Evaluation Sequence . . . 33
3.6.3 Evaluation Metrics . . . 34
3.7 Scope and Methodological Boundaries . . . 34

4 Experiments **37**
4.1 Experimental Design and Objectives . . . 37
4.1.1 Evaluation Philosophy . . . 37
4.1.2 Dimensions of Model Diversity . . . 37
4.2 Systems Under Test . . . 38
4.2.1 OpenAI Models . . . 38
4.2.2 XAI (Grok) Models . . . 38
4.2.3 Mistral Models . . . 39
4.2.4 Anthropic Models . . . 39

4.2.5 Google Gemini Models 40
4.2.6 Rule-Based Baseline: Angular Schematics 40
4.2.7 Cost and Step Constraints 40
4.2.8 Summary of System Configurations 41
4.3 Research Questions 41
4.4 Summary 42

5 Results **43**
5.1 Overall Performance Landscape 43
5.2 Comparative Analysis of Model Characteristics 45
5.2.1 The Impact of Reasoning Capabilities 45
5.2.2 The Role of Domain Specialization 45
5.3 Instance-Level Difficulty Analysis 45
5.4 Economic and Efficiency Analysis 46
5.5 Analysis of Experiment Failures 48
5.5.1 Step Limit Exceeded 48
5.5.2 Empty Patch Submissions 49
5.6 Addressing the Research Questions 49
5.7 Summary 50

6 Discussion **51**
6.1 Interpreting the Core Findings 51
6.1.1 The Surprising Strength of the Rule-Based Baseline 51
6.1.2 Why AI Agents Underperformed Expectations 52
6.1.3 The Heterogeneity of Real-World Projects 53
6.2 Economic and Practical Implications 54
6.2.1 Cost-Performance Trade-offs 54
6.2.2 Guidance for Practitioners 55
When Rule-Based Tools Suffice 55
When to Consider AI Agents 55
Model Selection Recommendations 55
The Hybrid Approach 56
6.3 Failure Mode Analysis 56
6.3.1 Infinite Loops and the Meta-Cognition Problem 56
Anatomy of the Failure Pattern 57
Comparison to Prior Work 57
The Meta-Cognition Gap 57
The Role of Temperature Settings 58
6.3.2 Empty Patches and the Multi-Step Reasoning Challenge 58
Diverse Failure Patterns 58
Contrast with Capable Models 58
The False Completion Problem 59
Implications for Model Selection 59
6.4 The Role of Scaffolding and Training 59
6.4.1 Why Minimal Scaffolding? 60
6.4.2 What This Means for Our Findings 60
6.4.3 What This Doesn’t Diminish 61
6.4.4 Scaffolding as a Research Variable 61
6.5 What BenchMAC Successfully Demonstrates 61
6.5.1 Effective Differentiation 61
6.5.2 Identification of Systematic Failure Modes 62

6.5.3 Economic Transparency 62
6.5.4 Reproducible Methodology 62
6.6 Contributions Beyond the Leaderboard 62
6.6.1 A Methodology for Repository-Level Migration Evaluation . . 63
6.6.2 Framework-Specific Evaluation 63
6.6.3 A Baseline for Agentic System Research 63
6.7 Implications for Future Research 64
6.7.1 The Dataset Diversity Challenge 64
6.7.2 Sub-Task Decomposition 64
6.7.3 Beyond Build Success: Quality Metrics 64
6.7.4 Scaffolding and Tool Optimization 65
6.7.5 The Reasoning Question 65
6.7.6 Cross-Framework Generalization 65
6.8 Summary 66

7 Limitations & Threats to Validity **67**
7.1 Dataset and Benchmark Design Limitations 67
7.1.1 Dataset Scale and Diversity 67
7.1.2 Base Commit Selection Strategy 68
7.1.3 Evaluation Scope and Metrics 68
7.1.4 Reproducibility and Infrastructure 69
7.1.5 Contamination and Generalization 69
7.2 Experimental Design Limitations 70
7.2.1 Limited Sampling Strategy 70
7.2.2 System Diversity Limitations 70
7.2.3 Analysis Depth 71
7.3 Threats to Validity 72
7.3.1 Internal Validity 72
7.3.2 External Validity 72

8 Conclusion & Future Work **73**
8.1 Summary and Contributions 73
8.2 Key Findings 74
8.3 BenchMAC as an Exploratory Tool 74
8.4 Implications 75
8.5 Limitations and Future Work 75
8.6 Closing Remarks 76

A Reproducible Evaluation Environment **77**
A.1 Dockerfile for Reproducible Environment 77

B System Prompts **79**
B.1 System Prompt for mini-swe-agent 79

C Agent Traces and Run Details **83**
C.1 Example Agent Reasoning and Execution Trace 83

Bibliography **87**

Eigenständigkeitserklärung **95**

List of Tables

3.1 BenchMAC v1.0 instances: repositories, base commits, and version jumps . . . 27
3.2 Canonical Evaluation Command Sequence . . . 33

4.1 Complete Specification of Systems Under Test . . . 41

5.1 Agent Performance Leaderboard (ranked by build success rate) . . . 43
5.2 Distribution of step limit failures across agents . . . 48
5.3 Distribution of empty patch failures across agents . . . 49

List of Abbreviations

ACI	Agent-Computer Interface
AI	Artificial Intelligence
API	Application Programming Interface
AST	Abstract Syntax Tree
BenchMAC	Benchmark for Migrating Angular Codebases
CI	Continuous Integration
CLI	Command-Line Interface
E2E	End-to-End (testing)
IDE	Integrated Development Environment
LLM	Large Language Model
LOC	Lines of Code
LTS	Long-Term Support
MCP	Model Context Protocol
NL	Natural Language
npm	Node Package Manager
PL	Programming Language
RAG	Retrieval-Augmented Generation
SUT	System Under Test

Chapter 1

Introduction

Modern enterprises depend on large software systems that evolve over many years. This natural evolution creates legacy code that is tied to old frameworks, outdated toolchains, and accumulated technical debt. When organizations delay upgrades, the cost and risk of change increase. Security fixes stop arriving, dependencies diverge, and development slows down [1, 2]. As a result, code migration is not only a technical task. It is a strategic activity that affects reliability, compliance, and the ability to deliver new features.

This thesis focuses on migrations within the Angular framework. Angular is a front-end platform with strong adoption in industry. It provides routing, forms, dependency injection, internationalization, a component library, server-side rendering, and a command-line interface in a single, coherent stack. Angular follows a predictable release cadence with long-term support windows, which is valuable for planning in large organizations. At the same time, this cadence makes upgrades unavoidable. Major releases often introduce breaking changes in the compiler, templates, testing tools, TypeScript versions, and third-party libraries. Teams must adjust build configurations, refactor application and test code, and handle ecosystem churn, such as the deprecation of end-to-end testing tools or shifts to new idioms. In large multi-team codebases and monorepos, these changes interact with CI pipelines and organizational processes, which adds another layer of complexity.

Manual migration practices are common but have clear limits. They depend on expert knowledge, they do not scale well across many repositories, and they are difficult to reproduce in a controlled way. Built-in tooling such as Angular schematics can assist with mechanical changes, but it does not cover the full scope of real projects. In practice, developers still perform many repository-wide edits, investigate failing builds and tests, and adapt to new framework idioms. These activities are labor-intensive and error-prone, and they are repeated at each major version jump.

Recent advances in generative AI suggest an opportunity to accelerate these upgrades. Large Language Models and coding agents can read and write code, run commands, and reason about errors. They can propose refactorings, apply systematic edits, and iterate against build and test feedback [3]. In the best case, they shift the human role from manual editing to review and supervision. However, the effectiveness of these systems remains unclear without careful evaluation. Results can vary with model choice, prompting strategy, tool access, and repository characteristics. Informal testing on a few projects is not enough to guide engineering decisions or research progress.

The current evaluation landscape shows a gap for this problem. Existing benchmarks in software engineering largely focus on code generation for isolated functions [4, 5], issue fixing in Python repositories [6, 7], or program translation between languages [8, 9]. Some datasets study version-conditioned code completion [10, 11] or library API changes [12, 13] but they do not cover end-to-end, repository-level framework upgrades in modern front-end stacks. There is no widely adopted, objective, and reproducible evaluation specifically designed for Angular migrations. As a result, it is difficult to compare systems, identify failure modes, or track progress over time.

To ground the rest of this thesis, we first conducted a systematic analysis of existing code migration benchmarks. This work, published as a peer-reviewed paper at CORIA-TALN 2025 [14], documents how state-of-the-art benchmarks create an illusion of competence by operating on miniature repos with simplified dependency management, limited technological diversity, and binary pass/fail metrics. The findings formalize the research gap that BenchMAC addresses and articulate the design requirements that shaped the benchmark.

This thesis addresses this gap by introducing **BenchMAC** (Benchmark for Migrating Angular Codebases), a systematic evaluation framework designed specifically for Angular version upgrades. BenchMAC provides a standardized way to measure how well different systems can migrate real Angular projects from one major version to the next.

The core idea is simple but rigorous. We take actual open-source Angular applications and define specific upgrade tasks, such as migrating from version 11 to version 12. Any system being tested (whether an AI agent, a rule-based tool, or even a human developer) attempts the migration and produces a set of code changes. We then apply those changes in a clean, isolated environment and check if the upgraded application can be built successfully. The primary success criterion is straightforward: does the migrated code install its dependencies and compile into a production-ready application?

This approach offers several advantages. First, it focuses on real projects rather than synthetic examples, capturing the complexity of actual codebases. Second, it verifies that migrations produce working code rather than just syntactically plausible changes. Third, the evaluation runs in controlled, reproducible environments using Docker containers, ensuring that results remain valid over time and can be verified by other researchers.

Contributions

This thesis makes four primary contributions to the field of AI-assisted software engineering:

1. A peer-reviewed analysis of the evaluation gap. We synthesize the limitations of existing code migration benchmarks, quantifying their scale, dependency assumptions, metric design, and technology coverage. The work, published at CORIA-TALN 2025 [14], formalizes the problem statement for BenchMAC and provides a community-vetted foundation for subsequent contributions.

2. A high-quality benchmark dataset. We provide 9 carefully curated migration instances derived from real open-source Angular projects. Each instance represents a complete, reproducible upgrade task from one major Angular version to the next. While small in scale, the dataset prioritizes quality and reproducibility over quantity. Each instance includes a frozen version of the codebase, a standardized Docker environment, and verified baseline behavior. All instances are publicly available with their complete specifications.

3. A reproducible evaluation methodology. We establish a two-stage framework that separates how systems generate solutions from how those solutions are evaluated. Systems produce a standard artifact (a Git patch file) that captures all code changes. The benchmark then applies these patches in isolated containers and executes a fixed sequence of commands to assess success. This design accommodates diverse system architectures while ensuring fair, deterministic evaluation. The methodology is transferable to other frameworks and migration scenarios beyond Angular.

4. An empirical study of migration systems. We evaluate 19 different AI configurations using a minimalist agent framework that isolates model capabilities from sophisticated scaffolding. This includes models from five major providers (OpenAI, Anthropic, Google, XAI, and Mistral), spanning different sizes, specializations, and price points. We also include Angular Schematics, the framework's built-in rule-based migration tool, as a baseline. This comparison reveals which systems succeed on these tasks, at what cost, and with what failure patterns. The study provides both practical guidance for organizations and a baseline for future research.

Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 reviews related work in three areas: the evolution of AI for coding, the landscape of benchmarks for evaluating language models, and existing work on code translation and migration. This establishes where BenchMAC fits in the broader research context.

Chapter 3 presents the methodology in detail. We explain the design principles behind BenchMAC, how we curate migration tasks, how we create reproducible evaluation environments, and how systems are tested. This chapter provides the technical foundation needed to understand and reproduce our experiments.

Chapter 4 describes the experimental design. We specify which AI systems we evaluate, how we configure them, and what research questions guide our investigation. The focus is on establishing a fair comparison across diverse systems while isolating the effects of model capabilities from other factors.

Chapter 5 presents the results. We show which systems succeeded and failed, at what cost, and with what patterns of behavior. The data reveals both expected and surprising findings about the current state of AI-assisted migration.

Chapter 6 interprets these findings. We discuss why certain systems performed well or poorly, what the economic implications are for practitioners, and what the results reveal about the capabilities and limitations of current AI agents. This chapter also contextualizes the strong performance of the rule-based baseline within the constraints of our dataset.

Chapter 7 examines limitations and threats to validity. We acknowledge the restricted scope of the current dataset, discuss what aspects of real-world migrations are not yet captured, and outline the boundary conditions for our findings.

Finally, Chapter 8 concludes by summarizing contributions, discussing implications for research and practice, and proposing directions for future work. The complete benchmark, including all data, code, and experimental results, is released as open source to enable further research in this area.

Chapter 2

Background & Related Work

This chapter provides the necessary background and reviews the related work to contextualize the contributions of this thesis. To understand the design and significance of BenchMAC, we must first examine three key areas. We begin by tracing the evolution of AI for coding, from early autocompletion to modern, tool-using agents. Next, we explore the landscape of benchmarks for generative AI, focusing on the principles, challenges, and methodologies that ensure rigorous evaluation. Finally, we narrow our focus to benchmarks specifically designed for code transformation tasks, such as code translation and version upgrades. This review will systematically identify the critical gap in evaluating frontend framework migrations, thereby establishing the clear need for the BenchMAC framework.

2.1 AI for Coding

The research and tooling landscape around AI for code has expanded rapidly, making it difficult to compare alternatives without consistent benchmarks and reproducible protocols. Recent surveys synthesize hundreds of papers across software engineering tasks, code generation, completion, program repair, and agentic systems, and converge on the same takeaway: the space is vast, fast-moving, and increasingly agentic. This motivates the need for rigorous, objective evaluation frameworks [15, 16, 17, 18, 19, 20, 21].

2.1.1 From Autocomplete to Agents: A Historical Perspective

Code-specialized Transformers and IDE autocomplete. Early code language models adapted natural language pretraining to source code and to aligning natural language (NL) with programming language (PL) representations. Examples include CodeBERT [22] and CodeT5 [23], which established strong baselines for code search, summarization, and generation. OpenAI's Codex, a GPT-3 variant fine-tuned on GitHub code, showed large gains on HumanEval [4] and powered the first widely-used in-IDE assistant, GitHub Copilot [24] which made "AI pair programming" mainstream [15].

Infilling and open models. Models began to support fill-in-the-middle code infilling, a key capability for editing and refactoring, via bidirectional context training (e.g., InCoder [25]). Open code LLMs such as CodeGen [26] and StarCoder or StarCoder2 [27, 28] extended context lengths and democratized access to strong performance. This marked a shift from autocomplete to true editing, where models reason about surrounding code and not just the next token .

Planning, tools, and agents. As developer tasks require iteration and tool use, LLMs began to *plan and act* with external tools such as search, shells, compilers, and test runners. ReAct [29] formalized prompting strategies that interleave reasoning traces with actions, catalyzing the agentic patterns later embedded in IDEs and CLIs [21].

Agentic coding models, protocols, and ecosystems. Recent dedicated coding models such as Codestral [30] and DevStral [31], alongside open families like Qwen-Coder [32], focus on long context, fast patching, and integration with test feedback. At the same time, the Model Context Protocol (MCP) [33] emerged as a standard for connecting agents to tools and organizational data, and is now itself the subject of new benchmarks [34].

2.2 Benchmarks in the generative AI era

2.2.1 Defining Benchmarks and Their Fundamental Role

A benchmark in artificial intelligence represents a standardized evaluation framework that provides quantitative measures of model performance across specific tasks and domains. At its core, a benchmark comprises several essential components: datasets containing structured data for evaluation, defined tasks that models must perform, metrics for measuring performance, evaluation protocols that ensure consistent assessment, and documentation that enables reproducibility. These components collectively establish benchmarks as crucial tools for measuring progress, ensuring objectivity, and enabling direct comparison between different approaches in the rapidly evolving landscape of artificial intelligence[35].

Benchmarks serve multiple fundamental purposes within the AI research ecosystem. They provide a standardized, objective framework that replaces subjective and ad-hoc evaluations, allowing researchers to make meaningful comparisons between different models and approaches. Beyond mere comparison, benchmarks create shared goals for the research community, helping to focus efforts and accelerate progress in specific areas through the incentive structures created by public leaderboards and competitive scoring systems. They also serve as accessibility tools, providing clear entry points for newcomers to the field by offering defined problems, datasets, and metrics to begin working on, while simultaneously preventing regressions by establishing baseline performance levels that new models should exceed.

2.2.2 The Evolution and Proliferation of LLM Benchmarks

The modern landscape of language model evaluation began with a fragmented approach where individual models were designed and optimized for specific tasks, making cross-system comparisons nearly impossible. This changed dramatically in 2018 with the introduction of GLUE (General Language Understanding Evaluation) [36], which represented a paradigm shift by bringing together multiple different language understanding tasks into a unified benchmark with a single overall score displayed on a public leaderboard. GLUE comprised nine English sentence understanding tasks covering a broad range of domains, data quantities, and difficulties, including tasks like sentiment analysis, textual entailment, and question answering.

The success of GLUE quickly revealed both its utility and limitations. Within just over a year of its introduction, state-of-the-art models began achieving performance levels that surpassed non-expert human baselines, suggesting diminished headroom for measuring further progress. This rapid saturation led to the development of SuperGLUE in 2019 [37], which retained only two of the hardest tasks from the original GLUE benchmark while introducing six more challenging tasks specifically designed to require deeper reasoning, more complex contextual understanding, and robust generalization beyond simple sentence classification.

The benchmark explosion continued with increasingly specialized evaluations targeting specific capabilities. The Massive Multitask Language Understanding (MMLU) [38] benchmark emerged as a comprehensive test covering 57 subjects across STEM, humanities, social sciences, and professional domains, featuring 15,908 multiple-choice questions designed to evaluate models' breadth of knowledge and reasoning capabilities. Mathematical reasoning became a focal point with benchmarks like GSM8K [39], containing 8,500 grade school math word problems requiring multi-step reasoning, and MATH, featuring competition-level mathematical problems. Safety and truthfulness evaluation gained prominence through benchmarks like TruthfulQA [40], comprising 817 questions across 38 categories designed to measure whether models generate truthful responses or mimic human misconceptions learned from training data.

2.2.3 Goals, Benefits, and Methodological Diversity

Modern benchmarks serve multifaceted objectives beyond simple performance measurement. They function as progress tracking mechanisms, providing quantitative measures that chart advancement toward specific research goals while creating objective standards for comparison that replace subjective evaluations. The competitive nature of public leaderboards creates powerful incentives for the research community, driving innovation and establishing clear targets for improvement. Additionally, benchmarks serve crucial quality assurance roles by preventing performance regressions and revealing model weaknesses across different dimensions of capability.

The methodological approaches to benchmark construction vary significantly across different domains and objectives. Data curation strategies range from crowdsourcing approaches that leverage distributed human intelligence to expert-authored datasets created by domain specialists with deep knowledge of specific fields. Evaluation metrics have evolved beyond simple accuracy measures to include sophisticated assessments like F1 scores for balancing precision and recall, BLEU [41] scores for translation quality, and domain-specific metrics tailored to particular tasks. The emergence of holistic evaluation frameworks represents a significant advancement, with initiatives like HELM [35] (Holistic Evaluation of Language Models) evaluating multiple dimensions simultaneously, including accuracy, calibration, robustness, fairness, bias, and efficiency.

2.2.4 The Benchmark Lifecycle and Saturation Crisis

A recurring pattern has emerged in the lifecycle of AI benchmarks that poses significant challenges for the field. Benchmarks typically begin with low model performance that gradually improves through research advances, eventually reaching saturation points where top models achieve 90-95% accuracy or even surpass human-level performance. This trajectory renders benchmarks obsolete as discriminative tools, necessitating the development of more challenging alternatives[42].

The website "Killed by LLM"[43] tracks how benchmarks quickly become obsolete as top models reach near- or superhuman scores, sometimes within months. Recent cases include ARC-AGI (2019–2024; 87.5%), MATH (2021–2024; 94.8%), BIG-bench Hard (2022–2024; 93.1%), HumanEval (2021–2024; 90.2%). Once saturated, these datasets no longer separate frontier systems, limiting their value for tracking progress and comparisons and pushing work toward harder, contamination-resilient evaluations.

The challenge is compounded by the acceleration of AI development, which means that models quickly "saturate" benchmarks by achieving near-perfect or superhuman scores at an unprecedented pace. This has led researchers to explore alternative approaches, including dynamic benchmark generation, contamination-resistant evaluation methods, and more challenging task formulations that can provide sustained evaluation utility [44, 45]

2.2.5 Critical Challenges and Systemic Flaws

Data Contamination

One of the most pervasive and concerning issues in contemporary benchmark evaluation is data contamination [46, 47], where test data appears in the massive web-scraped datasets used to train large language models. This occurs when benchmark data is present in the massive web-scraped datasets used to train models, allowing them to succeed through memorization rather than genuine problem-solving ability. This fundamentally undermines the validity of evaluation results.

The scale of this problem is substantial, with studies suggesting that significant portions of popular benchmarks may have been included in training data for major models [48]. This creates a critical challenge for the research community, as it undermines confidence in reported performance metrics and makes it difficult to assess genuine model capabilities versus memorized responses. Mitigation strategies include private holdout sets, dynamic benchmark generation, and contamination detection protocols, but these approaches require significant resources and careful implementation.

Gaming and Goodhart's Law

The competitive nature of benchmark leaderboards has led to sophisticated gaming behaviors that exemplify Goodhart's Law: "when a measure becomes a target, it ceases to be a good measure". Models are increasingly optimized to excel at the specific formats common in benchmarks, such as multiple-choice questions, rather than developing genuine understanding or capability. This optimization pressure can lead to benchmark manipulation through techniques like prompt engineering, selective reporting of results, or training specifically on benchmark-like tasks [49].

Recent controversies in the AI leaderboard community have highlighted how these gaming behaviors can undermine the integrity of evaluation systems [45]. The phenomenon extends beyond individual model optimization to include systemic issues like the tendency for models to exploit statistical regularities in benchmark construction rather than learning the underlying concepts being tested. This creates a disconnect between high benchmark scores and actual practical utility, as models that excel at standardized tests may fail at genuine real-world applications.

Evaluation Reliability and Reproducibility

The reproducibility crisis in large language model evaluation represents a fundamental challenge to scientific rigor in the field. Many evaluations suffer from missing implementation details, inconsistent evaluation setups, and sensitivity to seemingly minor hyperparameter changes [50]. Studies have revealed that minor perturbations to evaluation protocols, such as changing question formats [51], altering option ordering [52], or modifying prompt templates [53], can dramatically alter model performance, sometimes by tens of percentage points [51].

This sensitivity extends to broader methodological concerns, including the choice of evaluation metrics, the handling of edge cases, and the interpretation of results. The lack of standardized evaluation protocols across the community means that results from different research groups may not be directly comparable, undermining the fundamental purpose of benchmarks as objective comparison tools. Addressing these issues requires more rigorous experimental design, comprehensive reporting of methodological details, and community-wide adoption of standardized evaluation practices.

Gap Between Benchmark Performance and Real-World Utility

A critical limitation of current benchmarking approaches is the growing disconnect between high benchmark scores and actual practical capability. Models can achieve impressive performance on standardized tests while failing at tasks that require genuine understanding, adaptability, or robustness to real-world variation. This gap is particularly pronounced in domains where benchmark tasks are simplified or abstracted versions of real-world challenges, leading to overestimation of model capabilities based on benchmark results alone [54].

The focus on narrow, well-defined benchmark tasks can create misleading impressions of model generalization abilities. Real-world applications often require handling ambiguity, adapting to novel contexts, and integrating multiple types of knowledge in ways that are difficult to capture in standardized benchmark formats. This has led to calls for more holistic evaluation approaches that better reflect the complexity and variability of practical applications.

2.2.6 Emerging Solutions and Future Directions

Dynamic and Contamination-Resistant Benchmarks

The research community has begun developing innovative approaches to address the fundamental limitations of static benchmarks. Dynamic benchmark generation represents a promising direction, with systems like YourBench [55] creating fresh evaluation tasks automatically to prevent contamination and gaming. These approaches leverage large language models themselves to generate diverse, novel tasks that maintain the same difficulty and evaluation principles as original benchmarks while using entirely new content.

Contamination-resistant benchmark design has emerged as another critical research area, with approaches like MMLU-CF [56] implementing decontamination rules and maintaining private test sets to ensure evaluation integrity. The concept of "platinum benchmarks" has been proposed for scenarios requiring 100% reliability, where benchmark creators maintain strict control over test data and evaluation protocols to prevent any form of contamination or gaming [57].

Human Preference and Multi-Modal Evaluation

The field is witnessing a significant shift toward human preference based evaluation methods that better capture the subjective aspects of model performance that traditional metrics miss [58]. These approaches recognize that many important aspects of language model behavior, such as helpfulness, harmlessness, and honesty, are difficult to capture through objective metrics alone. The development of "LLM as judge" frameworks allows for scalable evaluation using models to assess other models' outputs according to human defined criteria [59].

Multi-modal evaluation has become increasingly important as AI systems expand beyond pure text processing to include visual, auditory, and other modalities. Benchmarks like MMMU [60] (Massive Multi-discipline Multimodal Understanding and Reasoning) assess models' abilities to process and reason across multiple modalities simultaneously, reflecting the complexity of real-world AI applications. This trend toward multi-modal assessment represents a fundamental evolution in how we evaluate AI capabilities.

2.2.7 Implications for the Field

The current state of AI benchmarking presents both significant challenges and unprecedented opportunities for the research community. While benchmarks remain essential tools for measuring progress and enabling scientific comparison, their limitations require careful interpretation and complementary evaluation methods. The rapid pace of model advancement has outstripped the development of evaluation methodologies, creating urgent needs for innovation in assessment approaches.

The systemic issues of contamination, gaming, and reproducibility threaten the scientific integrity of AI research if left unaddressed. However, these challenges have also led to important innovations in evaluation methodology, including dynamic benchmark generation, contamination detection techniques, and more sophisticated approaches to measuring model capabilities. The field is evolving toward more nuanced, multi-dimensional evaluation approaches that better capture the complexity of real-world AI applications.

Future progress in AI evaluation will likely require addressing several key areas: developing contamination-resistant benchmarks that maintain evaluation integrity over time, creating more sophisticated metrics that capture important aspects of model behavior beyond simple accuracy, establishing standardized evaluation protocols that enhance reproducibility across research groups, and bridging the gap between benchmark performance and real-world utility through more authentic evaluation tasks. The continued evolution of evaluation practices to match the pace of model advancement will be crucial for maintaining scientific rigor and ensuring that reported progress reflects genuine advances in AI capabilities rather than optimization to particular evaluation metrics.

2.3 Code Generation Benchmarks

The evaluation of code generation capabilities in large language models (LLMs) has progressed rapidly over the past few years. Early benchmarks introduced simple function completion tasks, but as models became more capable, researchers recognized the need for more realistic and comprehensive evaluation methods. Today, the field spans benchmarks at multiple scales, from isolated function problems to repository level assessments, each addressing different aspects of software development.

At the same time, persistent challenges such as contamination, limited realism, and fragile evaluation methodologies continue to shape the landscape.

This section reviews the evolution of code generation benchmarks, the methods used to evaluate them, and the main challenges that remain. The goal is not to provide an exhaustive list, but to distill the lessons that matter for designing new evaluation frameworks.

2.3.1 From Functions to Repositories: A Historical Progression

Function-Level Benchmarks

The modern era of code generation evaluation began with **HumanEval** [4], introduced by OpenAI in 2021 to assess Codex models. It contains 164 short Python programming problems, each defined by a function signature, a docstring, and a unit test suite. The key contribution was not only the dataset itself but also its **execution-based metric**, known as *pass@k*. This metric evaluates whether at least one of k generated code samples passes all unit tests, thereby assessing the *semantic correctness* of the generated code rather than its syntactic similarity to a reference solution. In other words, it focuses on whether the code actually behaves as intended, regardless of how it is written, which is crucial for capturing the true functional capability of code generation models.

HumanEval quickly became a de facto standard, but its limitations were soon apparent. The small number of problems made memorization plausible, and the relatively shallow unit tests meant many incorrect solutions could still pass [61].

Shortly after, **MBPP (Mostly Basic Programming Problems)** expanded the scope with 974 beginner-friendly problems [5]. MBPP focused less on algorithmic puzzles and more on practical tasks (e.g., string manipulation, list processing) representative of what novice programmers encounter. While this broadened coverage, MBPP shared HumanEval's weaknesses: shallow testing and limited realism.

Improving Coverage and Freshness

To address coverage issues, **EvalPlus** [61] introduced enhanced test suites, creating HumanEval+ and with tens of times more test cases per problem. This revealed that many models previously believed to be strong were in fact generating fragile or incorrect code that slipped past the limited original tests. For example, GPT-3.5's reported pass@1 score dropped by over 10 percentage points once evaluated under the stricter test sets.

Another pressing issue was **contamination**, where benchmark problems leak into training data. Because HumanEval and MBPP are relatively small and widely distributed, solutions often appear in the datasets of large models, inflating reported performance [20]. To combat this, **LiveCodeBench** [44] proposed a continuously updated benchmark that sources problems from recent competitive programming contests (e.g., LeetCode, AtCoder, Codeforces) that occur *after* model training cutoffs. This ensures models face genuinely novel problems.

Repository-Level Benchmarks

Function-level evaluations, while convenient, fail to capture the complexity of real software projects. Modern development involves multiple files, dependencies, and cross-module reasoning. To address this gap, **RepoBench** [62] introduced repository-level tasks. Its design combines three subtasks:

- **Retrieval (RepoBench-R):** identify relevant snippets across files,
- **Completion (RepoBench-C):** predict code while using cross-file context,
- **Pipeline (RepoBench-P):** combine retrieval and completion in an end-to-end workflow.

Performance drops sharply in these settings: models that perform well on single functions often struggle when cross-file reasoning is required.

SWE-Bench [6] further advances repository-level realism by posing end-to-end issue resolution on active, real-world Python projects. Each instance pairs a GitHub issue with the repository state prior to the fix; a system must produce a patch that makes the relevant tests pass, and results are reported as the percentage of issues *resolved*.

Follow-up efforts such as **CrossCodeEval** [63] extended repository-level evaluation across multiple languages (Python, Java, TypeScript, C#), while **ExecRepoBench** [64] pushed further by collecting active GitHub repositories and executing generated changes against full test suites. These efforts highlight a clear trend: benchmarks are moving from artificial puzzles toward real software engineering environments.

2.3.2 Execution-based Evaluation

As code generation benchmarks evolved from simple function completion to complex repository-level tasks, the evaluation methodology underwent a fundamental shift: from static syntactic comparison to dynamic execution-based metrics.

From Syntax to Semantics. Early approaches like CodeBLEU [65] attempted to measure code quality by comparing generated code against reference solutions using n-gram overlap and abstract syntax tree (AST) matching. However, these metrics proved inadequate. Syntactically different code can be semantically identical, and conversely, syntactically similar code may behave entirely differently. The field converged on a more pragmatic principle: **does the code actually work?**

The pass@k Metric. HumanEval [4] introduced the pass@k metric, which measures the percentage of problems where at least one of k generated solutions passes a suite of predefined unit tests. This execution-based approach became the de facto standard, evaluating functional correctness rather than surface-level similarity. However, HumanEval's test suites were manually curated and, as EvalPlus [61] later demonstrated, often failed to cover edge cases. This highlighted the challenge of test quality in execution-based evaluation.

Scaling to Repositories. SWE-Bench [6] marked a paradigm shift by moving from isolated functions to real-world repositories. Rather than generating standalone functions, systems must produce Git patches that modify existing codebases to make failing tests pass. Crucially, SWE-Bench automated test collection by mining GitHub pull requests, dramatically improving scalability. The primary metric became **resolution rate**: the percentage of instances where the generated patch both applies cleanly and causes all tests to pass.

The Reproducibility Challenge. Execution-based evaluation introduces a critical requirement: **reproducible environments**. For function-level tasks with minimal dependencies, this is trivial. For repository-level benchmarks, however, inconsistent dependency versions or system configurations can produce unreliable results. SWE-Bench Verified [7] addressed this by mandating Docker containers, ensuring that every evaluation runs in an identical, isolated environment. This principle is one that BenchMAC directly adopts.

2.3.3 Expanding the Scope of Tasks

Beyond simple function generation, benchmarks now cover a wider set of developer activities:

- **Class and object-oriented code:** ClassEval [66] requires generating coherent classes with interdependent methods.
- **Test generation:** TestGenEval [67] evaluates models' ability to write unit tests for given functions.
- **Feature implementation:** FEA-Bench [68] assesses whether models can implement new features across multiple files based on natural language specifications.
- **Domain-specific coding:** specialized sets exist for data science (DS-1000 [69]), security (CyberSecEval [70]), and competitive programming (CodeContests [44]).

Each of these task families highlights a different competency: reasoning about object hierarchies, understanding test design, repairing faulty logic, or implementing features at scale. Together, they show that "code generation" is not a single ability but a spectrum of skills.

2.3.4 Persistent Challenges

Despite progress, several issues remain unsolved:

- **Contamination and staleness:** widely shared benchmarks are easily memorized, inflating performance.
- **Limited realism:** even repository-level tasks often lack the messy build systems, dependency resolution, and collaborative workflows of real projects.
- **Language bias:** Python dominates most datasets ($\approx$78%), while languages like C++, JavaScript, and Rust remain underrepresented.
- **Evaluation costs:** large-scale benchmarks like SWE-Bench or ExecRepoBench are computationally expensive, making them difficult to run at scale.
- **Semantic understanding vs. pattern matching:** models can pass tests without truly understanding specifications, leading to fragile or hallucinated solutions.

2.3.5 Lessons for Benchmark Design

From these developments, several design principles emerge that inform new evaluation frameworks:

1. **Execution in realistic environments:** evaluation should occur in standardized sandboxes or Dockerized setups to mimic developer workflows.

2. **Comprehensive metrics:** correctness must be complemented with robustness, efficiency, and cost considerations.
3. **Freshness and contamination control:** benchmarks should evolve or source tasks after training cutoffs.
4. **Transparency and reproducibility:** results must be based on open artifacts (patches, logs, test suites) to allow verification.
5. **Domain and scale awareness:** benchmarks should reflect diverse languages, paradigms, and repository-level contexts.

2.3.6 Bridge to Migration Benchmarks

General-purpose code generation benchmarks provide valuable insights into how LLMs reason about functions, classes, and repositories. However, they fall short when evaluating **software migrations**, which involve not only generating code but also dealing with framework versioning, dependency conflicts, configuration files, and build systems. The next section focuses specifically on benchmarks for **translation and migration tasks**, where these additional complexities come to the forefront.

2.4 Benchmarks in code transformation

2.4.1 Code translation

Introduction and Scope

Code translation, also known as transcompilation, transpilation, or source-to-source compilation, refers to the process of converting source code written in one programming language into an equivalent program in a different programming language, while preserving the original program's functionality and logic. Unlike traditional compilers that translate high-level code to lower-level representations (such as assembly language or machine code), transcompilers convert between programming languages that operate at similar levels of abstraction.

To clarify terminology used throughout this thesis, we adopt the following working definitions:

Code translation (cross-language, code→code): Convert a program from source language A to target language B while preserving externally observable behavior. The primary concern is semantic equivalence despite different syntax and idioms.

Framework/library migration (intra-language, cross-framework): Move an application from framework F_1 to F_2 (e.g., AngularJS→Angular) or across major framework families (React↔Angular) within the same language. The primary concern is preserving behavior while re-mapping concepts, APIs, and adopting modern idioms.

Version upgrade (intra-language, intra-framework): Move an application from F@v to F@v+1 (e.g., Angular 17→18). The primary concern is satisfying specific version constraints, handling deprecations and configuration changes while keeping behavior stable.

Code translation research provides valuable insights for framework migrations, particularly in evaluation methodologies and the challenges of maintaining semantic equivalence across different representations of the same underlying functionality.

Industrial Context and Motivations

The motivations for code translation, which inform the broader landscape of code migration, stem from several critical industrial needs. Modernization of legacy systems represents a primary driver, as organizations seek to convert code written in outdated languages (such as COBOL or FORTRAN) to modern alternatives (like Java, Python, or C++) to improve maintainability, security, performance, and adaptability. This addresses the lack of community support and expertise for legacy languages by transitioning to languages with more active development communities.

Security and reliability improvements constitute another critical motivation, particularly for languages like Rust. Such migrations aim to eliminate memory bugs, data races, and other vulnerabilities by leveraging Rust's strong compile-time guarantees, ownership model, and type system to eradicate these issues [71].

The scale and cost of manual translation efforts highlight the urgent need for automated approaches. Manual translation processes are inherently slow, repetitive, and error-prone. Historical examples demonstrate the severity of these challenges: the Commonwealth Bank of Australia spent approximately $750 million and five years translating its platform from COBOL to Java [2]. More broadly, failed language conversion attempts have led to severe consequences, including companies going bankrupt, departments being dismantled, and millions of dollars lost [1].

Test case generation and validation present additional complexities in manual translation efforts. Generating or adapting unit tests for the target language requires deep understanding of the codebase and functional equivalence verification. Existing tests often have low coverage, further complicating validation efforts [72].

Traditional Approaches and Limitations

Rule-based transpilers represent the traditional approach to automated code translation. These systems operate by representing source code as an Abstract Syntax Tree (AST) and applying manually defined grammar or rewrite rules to transform it into the target language. Examples include CxGo, C2Rust, Tractor, JavaToCSharp/Sharpen, JSweet, GWT, Java2Python, and py2java.

Rule-based transpilers offer several advantages. Their deterministic nature ensures predictable and consistent program transformations [73]. The translation process is generally deterministic, leading to more predictable reasoning compared to probabilistic approaches. These systems can be highly effective for closely-related language pairs (such as Java to Scala) and are capable of handling large codebases efficiently.

However, rule-based approaches face significant limitations. While syntactic transformation can preserve functional behavior, achieving correct and idiomatic translation often requires semantic understanding. By relying solely on syntactic operations, rule-based programs risk producing code that fails to respect the best practices and readability conventions of the target language, leading to incomprehensible and unmaintainable results.

For example, in C to Rust translations, rule-based tools like TRACTOR typically produce code that contains more unsafe code than traditional Rust implementations [71], undermining the safety guarantees that motivated the migration. In the COBOL to Java domain, this phenomenon is colloquially termed "JOBOL" – Java code that retains COBOL idioms rather than adopting Java conventions [74].

The output of rule-based systems often retains the idiom of the source language rather than adopting the conventions of the target language. Translating between languages with fundamentally different programming philosophies (such as object-oriented versus functional, or dynamically-typed versus statically-typed) proves particularly challenging, sometimes even impossible. When language structures differ significantly (for example, functional language Haskell and procedural object-oriented language Java), writing comprehensive conversion rules may not be feasible[75].

Scalability and adaptability represent additional concerns. Rule-based systems are typically limited to specific language pairs and struggle to generalize to newer features of the same language pairs, let alone other programming languages. Manual updates are required after each major programming language version release [76].

Neural and LLM-Based Translation

Why Neural Approaches Emerged Neural Machine Translation (NMT) models, including Large Language Models (LLMs), trained on existing human-readable code, demonstrate the potential to produce idiomatic, easy-to-understand translations. Unlike rule-based systems, neural approaches can "understand" semantics and generate code that is idiomatic to the target language.

LLMs possess multilingual capabilities that facilitate translation across numerous language pairs without exhaustively defining rules for every possible pair. They demonstrate zero-shot code translation abilities, meaning they can translate between languages even without specific training for that pair, by leveraging learned representations from reference code in multiple languages. Lachaux et al. [2] shown that LLMs can outperform commercial alternatives based on traditional transpiler technology.

The field has evolved through several distinct phases:

Early Neural and Unsupervised Approaches (Pre-2022) This initial phase marked the transition from hard-coded rules to learned representations. Chen et al. [77] introduced the first deep learning approach to program translation by operating on hierarchical code structure rather than treating code as flat text sequences. Their tree-to-tree neural network processed abstract syntax trees (ASTs), enabling better capture of structural dependencies and demonstrating up to 15% better accuracy than sequence-to-sequence models on tasks like CoffeeScript-to-JavaScript translation.

TransCoder, introduced by Lachaux et al. [2], was a major breakthrough because it was the first system to automatically translate code between programming languages without needing matching examples in both languages. Instead of requiring large datasets of the same program written in two languages, TransCoder learned from regular (monolingual) code by using special techniques such as masked language modeling (where parts of the code are hidden during training), denoising auto-encoding (learning to fix "noisy" or corrupted code), and back-translation (teaching itself by translating code to another language and back again). This made it possible to train the model using only ordinary code from each language. TransCoder achieved remarkable results for its time (over 80% accuracy for Java to C++), proving that neural models could learn translation patterns without direct supervision. Crucially, this work pioneered the use of "computational accuracy" for evaluation, shifting focus from textual similarity to functional correctness.

Pre-Trained Models Era (2022-2023) The rise of pre-trained transformers for code, such as CodeBERT, PLBART, and CodeT5, created new foundations for translation systems. CodeT5 introduced "identifier-aware" pretraining, recognizing the semantic importance of variable and function names, leading to state-of-the-art performance on translation benchmarks like CodeXGLUE. These models served as bridges from task-specific architectures to the powerful, general-purpose language models that followed.

Large Language Model Era (2022-2024) Massive-scale LLMs marked a qualitative transformation. Models like OpenAI's Codex[4] and CodeGeeX [78] demonstrated that scaling up model size and training data led to significant improvements in cross-language generalization. CodeGeeX, a 13-billion parameter model, showed strong performance across 23 different programming languages.

Architectural innovations emerged for efficiency. SteloCoder[79] employed a Mixture-of-Experts (MoE) approach with five specialized experts, each focused on a specific source language (C++, Java, JavaScript), achieving high performance while requiring dramatically less training time than monolithic models.

Systems began incorporating execution feedback. CoTran[76] combined reinforcement learning with compiler feedback and symbolic execution, allowing models to learn from compilation and execution attempts, leading to +14.89% improvement in functional equivalence over baseline approaches.

Repository-level complexity became a focus. AlphaTrans [72] used a neuro-symbolic approach to successfully translate 10 real-world projects, representing a breakthrough in scaling translation to entire software repositories. While functional correctness remained challenging (25.14%), it proved the feasibility of project-level analysis.

Modern Agentic and Specialized Systems (2024-Present) Current approaches recognize that complex code migration requires decomposition into specialized, iterative, and tool-using systems. TRANSAGENT [80] exemplifies multi-agent decomposition, featuring four distinct agents: an Initial Code Translator, a Syntax Error Fixer, a Code Aligner, and a Semantic Error Fixer. By leveraging domain expertise for each step, this approach achieves substantial improvements over monolithic, single-LLM systems.

Current Limitations Despite advances, neural approaches face significant challenges. Functional correctness remains problematic, as AI models often struggle to generate functionally accurate code [81]. Code may appear superficially correct but not function as intended. Unlike rule-based systems, LLM-generated code is not guaranteed to be correct, and deploying such code without rigorous testing could create an ecosystem of untested, unproven, and potentially insecure software.

Scalability issues persist. Studies on LLM performance in long context scenarios, including [82] focusing specifically on code, show that model performance deteriorates strongly when context increases, even when largely under the limit of the context window. Thus, LLM performance can decrease significantly when dealing with large codebases.

Evaluation Challenges and Evolution

Why Code Translation Needs Specialized Evaluation General benchmarks like HumanEval and MBPP primarily evaluate a model's ability to generate code from natural language descriptions. Code translation, however, requires ensuring that translated code is functionally equivalent and semantically preserves the behavior of the original source code, regardless of syntactic changes. This represents a stronger requirement than simply producing working code from a natural language prompt.

LLMs can struggle to infer a program's underlying functionality from its appearance, leading to failures in preserving semantics. Furthermore, modern code-generation benchmarks do not apply to low-resource languages such as COBOL. There exists a significant scarcity of high-quality parallel code corpora for training and evaluating translation models.

Evolution of Evaluation Approaches Early studies often borrowed metrics from natural language translation, such as BLEU or Edit Distance, to compare generated code to references. However, lexical similarity proved to be an imperfect proxy, as two programs can be textually dissimilar yet functionally identical, or vice versa.

The field shifted towards execution-based metrics with HumanEval's introduction of pass@k, which checks if generated code passes predefined unit tests. Execution metrics directly measure functional correctness and quickly became the de facto standard for both natural language-to-code and code-to-code tasks.

Specialized metrics emerged for code translation scenarios. TransCoder introduced Computational Accuracy, requiring translated functions to produce the same outputs as originals for test inputs. ClassEval-T [83] added Dependency Preservation (DEP) metrics to verify that models correctly maintain relationships between class members and external calls during translation.

The Debugging Success Rate (DSR@k), introduced by Yan et al. [9], evaluates a model's ability to self-correct by measuring the fraction of generated code translations that can reach a passing state within k automated debug attempts. This metric reflects an important practical aspect: in real migrations, developers or agentic systems might iteratively refine code.

Current Benchmarks and Metrics

Benchmark Progression Code translation benchmarks have evolved from simple function-level tasks to complex repository-level evaluations. Early datasets like TransCoder-Test and code translation tasks in CodeXGLUE focused on translating self-contained functions or code snippets between languages. While valuable as initial baselines, these tasks did not capture the challenges of migrating large, industrial codebases.

ClassEval-T [83] introduced class-level translation tasks, requiring models to handle internal class dependencies and external library calls, representing a step between function-level and full-project translation.

RepoTransBench [84] pushed further by evaluating translations at the repository level, testing model performance on entire codebases with multiple files and project structures. This marked a substantial increase in realism, as models must resolve cross-file references and produce code that integrates into larger software systems.

Metric Evolution and Limitations Repository-level benchmarks expanded metric suites to capture multiple dimensions of success. RepoTransBench introduced Success@k (percentage of projects where all tests pass in at least one of k tries), Build@k (percentage of projects that at least compile), and Average Pass Rate (APR, the average percentage of tests passed across all projects and attempts).

However, these metrics face limitations. Binary success metrics at the repository level mean that one critical error can mask partial successes. Under such metrics, a single compilation error can prevent running any tests, making it impossible to recognize components that were correctly migrated. This all-or-nothing scoring can be overly harsh and uninformative for diagnosing model performance on large tasks.

To address this, researchers have proposed more nuanced evaluation methods. The skeleton-guided translation method by Zhang et al. [85] explicitly breaks migration into stages: first extracting and translating a structural "skeleton" (class signatures, function interfaces, import statements), then iteratively filling in implementation details. This approach enables modular evaluation where parts of the code translation can be validated independently, yielding finer-grained assessment than traditional binary metrics.

Even with structured approaches and multiple iterations of automated debugging, the best models only reached about 21% success on full repository migrations, reinforcing the persistent nature of these challenges while demonstrating the value of richer metrics.

Remaining Gaps and Limitations

Despite advances in benchmarks and metrics, significant gaps remain between current evaluation setups and real-world needs. A major shortcoming is the lack of representativeness: even recent benchmarks cover only a narrow slice of industrial migration scenarios.

RepoTransBench, while a leap forward, uses relatively small projects with simplified dependency management and limited technological diversity. The projects represent less than 2% of the average size of an industrial project. Furthermore, 98 out of 100 projects in RepoTransBench remain below the 32,000-token threshold, where LLM performance in long-context scenarios significantly drops, thus limiting its relevance for evaluating LLMs in industrial settings [14].

Current benchmarks omit important facets of real-world migration. RepoTransBench does not include tasks for test generation or adaptation, instead using unit tests pre-written by benchmark creators. In practice, migrating a codebase often requires not just translating syntax but also adapting test suites, build scripts, and deployment configurations to the new environment – aspects rarely reflected in current benchmarks.

Wang et al. [84] identified complexity factors unique to large-scale migrations as major contributors to failure. As project size grows, so does the difficulty for models to handle dependency management and project structure. Their error analysis found that nearly half of failures stem from misconfigured build files or environment settings, about one-third from limited code understanding, and a quarter from incomplete code generation.

Bridge to Framework Migration

Many methods and evaluation practices from code translation research strengthen version-upgrade and framework migration research. Three key insights transfer directly:

First, execution-based evaluation proves superior to surface similarity metrics. Using compile, build, and test-driven metrics instead of lexical metrics detects semantic regressions regardless of syntax changes. Second, granular, partial-credit scoring approaches avoid all-or-nothing repository scores and expose where an upgrade fails, proving useful for large repositories. Third, tool-in-the-loop feedback from compilers, TypeScript, Angular CLI, unit tests, end-to-end tests, and linters can guide automated repair exactly like compilers and executors in translation scenarios.

However, version upgrades introduce constraints and artifacts that do not usually appear in pure code translation, necessitating specialized evaluation approaches. Tests must evolve alongside code, as assertions, test utilities, and harnesses often change across versions. Adapting and generating tests becomes part of the task, whereas translation benchmarks typically supply fixed tests.

Project setup matters as much as the code itself. Configuration files, build settings, and automated checks all require updates. If any component is incorrect, the entire upgrade can fail even if the source code appears correct.

Finally, version upgrades involve more than line-by-line translation. They require making projects follow newer conventions, dropping outdated features, and adopting recommended practices. This semantic evolution represents a fundamental difference from pure translation tasks and necessitates evaluation frameworks that can assess adherence to version-specific best practices and idioms.

2.4.2 Version Upgrades: A Critical and Evolving Challenge in Software Maintenance

The Industrial Imperative for Modernization

Beyond the cross-language translation of entire codebases, a more frequent and equally critical task in software engineering is the *version upgrade*: the process of migrating a project to a newer version of its existing language, library, or framework. For businesses, regular upgrades are a strategic necessity. New versions often include patches for security vulnerabilities, bring performance improvements, and give developers access to new tools and capabilities that can improve the product. Staying on a supported version ensures a project can receive bug fixes, making it easier to maintain over the long term.

Despite these benefits, manually upgrading software is a significant challenge. The process is often labor-intensive, prone to human error, and can be very expensive [86]. This difficulty is made even harder by the rapid pace of API evolution in modern ecosystems. In fast-moving languages like Rust, for instance, hundreds of API changes can occur over just a couple of years, creating a constant flow of old features being replaced by new ones that developers must adapt to [87]. This constant flux, combined with the risk of introducing subtle semantic regressions where the upgraded code compiles but behaves incorrectly, motivates the search for automated solutions.

Automating Upgrades: From Rule-Based Tools to LLM-Powered Transformation

The initial wave of automation for version upgrades relied on *rule-based tools*. These range from simple text replacement with regular expressions to more advanced systems that operate on the code's Abstract Syntax Tree (AST), such as OpenRewrite [88]. In the context of this thesis, the most relevant example is *Angular Schematics*[1], the framework's built-in tool that automates many common changes during an `ng update` operation.

While powerful for well-defined, syntactic changes, these rule-based systems have fundamental limitations. They are rigid, capable only of applying transformations that have been explicitly programmed. They struggle to reason about code semantics, cannot address behavioral changes in APIs where the signature remains

[1] https://angular.dev/tools/cli/schematics

the same, and often cannot make the architectural or design decisions required to adopt new framework idioms.

The recent advent of LLMs represents a paradigm shift, offering a solution to the rigidity of these tools. LLMs excel at processing diverse code structures and can flexibly handle a wide variety of coding patterns without requiring manually crafted rules for each scenario [86]. Their ability to reason about the semantics of code allows them to tackle complex refactorings that are beyond the scope of AST-based transformations. The industrial viability of this approach has been demonstrated at scale in experience reports from Google, where LLM-assisted workflows have achieved an estimated 50% reduction in the total time spent on massive internal migrations [86, 89].

Two key techniques have emerged for applying LLMs to version upgrades:

1. **Retrieval-Augmented Generation (RAG)** has proven to be a critical technique for overcoming the knowledge cutoff dates of pre-trained models. By providing the LLM with up-to-date documentation and API changelogs at inference time, RAG significantly improves performance on tasks involving recent library versions [87, 90].

2. **Iterative Repair and Agentic Systems** move beyond one-shot generation. These systems create a feedback loop where an AI "agent" tries to perform the migration, runs tools like the compiler or test suite to check its work, and then uses the error messages to try again. This approach allows the agent to solve more complex problems, mimicking a human debugging process, and has been shown to be effective in automatically repairing breaking dependency changes [91].

Benchmarking the Evolution: A Maturing Evaluation Landscape

To measure the capabilities of these new LLM-based systems, the research community has developed a new generation of specialized benchmarks that have rapidly matured in realism and rigor. This evolution can be understood in three phases.

Defining the Problem at the Snippet-Level (Version-Awareness). The first challenge was to measure an LLM's ability to generate a small piece of code for a specific library version. *VersiCode* [10] pioneered this area with a large-scale Python dataset for "version-controllable" code generation, though its evaluation relied primarily on static metrics. A direct response, *GitChameleon* [11], argued for greater rigor by introducing a smaller, manually curated benchmark where every problem was evaluated using *execution-based metrics* (*pass@k*), establishing functional correctness as the key measure of success.

Curating Real-World Failures and Ensuring Reproducibility. To evaluate an AI's ability to repair migrations, researchers began curating datasets of known failures. *CompSuite* [92] was the first to provide a dataset of real Java library upgrades that cause breaking changes, each paired with a unit test to reproduce the issue. *BUMP* [12] made a significant methodological leap forward by focusing on *guaranteed, long-term reproducibility*. It packages each of its 571 breaking Java dependency updates into a pair of Docker containers, ensuring a stable and consistent evaluation environment. This containerized approach is now considered the gold standard for creating scientifically robust benchmarks and is a core principle adopted by this thesis.

Scaling to Full Repository-Level Migration. Most recently, research has shifted to the more realistic and complex challenge of migrating entire software repositories. *COUPJAVA* [93] provided the first large-scale dataset of Java upgrade histories, distinguishing between fine-grained (method-level) and coarse-grained (repository-level) tasks and establishing compilation checks as a critical metric. The most direct precedent for this thesis is *MigrationBench* [94], a large-scale benchmark for migrating entire Java repositories from version 8 to later LTS versions. It employs an execution-based evaluation harness within a controlled environment to assess the success of a migration based on compilation, test execution, and correct dependency resolution.

Identifying the Research Gap and Positioning BenchMAC

A review of the state-of-the-art in version upgrade benchmarks reveals a clear pattern: an overwhelming focus on backend programming languages, primarily *Java and Python*. While this research has established robust evaluation principles—namely, a focus on repository-level tasks, execution-based metrics, and containerized reproducibility, it has largely overlooked the unique challenges posed by modern frontend frameworks.

This constitutes not just a language gap but a critical domain gap. Migrating a complex frontend application, particularly within an opinionated framework like Angular, involves more than just updating language syntax. It requires an AI to:

- Interact with a complex build toolchain (the Angular CLI).
- Understand and modify declarative template syntax (HTML) in addition to imperative code (TypeScript).
- Correctly update a web of configuration files (`package.json`, `angular.json`, `tsconfig.json`).
- Reason about framework-specific idioms, such as component lifecycles, dependency injection, and state management.

While the field has converged on *how* to build a rigorous migration benchmark, a significant gap remains in *what* is being benchmarked. This thesis addresses this gap directly. It introduces **BenchMAC**, a benchmark that applies the state-of-the-art evaluation principles established in the Java and Python ecosystems to the specific, complex, and industrially relevant domain of repository-level Angular migrations. In doing so, it provides the first rigorous framework for measuring and advancing AI performance on this critical software modernization task.

Chapter 3

Methodology

The literature review presented in Chapter 2 revealed a significant gap in the evaluation landscape for AI-driven software engineering. While numerous benchmarks exist for backend languages and function-level code generation, there is a clear need for a framework that addresses repository-level, frontend-specific migration tasks. Given the widespread adoption of the Angular framework in industry and its relevance to projects at Onepoint, we chose to focus our efforts on this ecosystem.

This chapter introduces **BenchMAC**, a *Benchmark for the Migration of Angular Codebases*, designed to meet this need. The primary objective of BenchMAC is to evaluate the ability of an AI system to migrate an Angular codebase from one major version to the next, for example, from version 9 to version 10. To standardize this evaluation, we define a set of reproducible migration tasks called *instances*. Each instance consists of:

- An Angular codebase from a real-world, open-source project.
- A specific starting point in its version history, defined by a Git commit hash.
- The associated task of migrating the project to its next major version.

A key principle of BenchMAC is its agnosticism regarding the system being evaluated. Throughout this thesis, we use the term *System Under Test* (SUT) to refer to any system, tool, or process tasked with performing the migration. An SUT can range from a rule-based tool like Angular Schematics to a complex AI agent, or even a human developer serving as a baseline.

3.1 Guiding Principles and Overall Design

The methodology is built upon a fundamental design choice: the strict decoupling of the solution generation from its evaluation. This separation manifests as a two-stage process: **Patch Generation** and **Patch Evaluation**. This architecture ensures that SUTs can be developed with maximum freedom, while their outputs are graded with maximum rigor and reproducibility. The overall methodology is structured around a two-stage process: Patch Generation and Patch Evaluation.

In the Patch Generation stage, the System Under Test (SUT), which could be an AI agent, a rule-based tool, or even a human, undertakes the migration task. The SUT operates with complete freedom, choosing its own workflow. It might run commands like `ng update`, search through documentation, repetitively build the project for error feedback, or leverage large language models. There are no restrictions imposed by the benchmark regarding the internal processes used. However, by the end of this stage, the SUT must produce a single, static artifact that fully represents the proposed transformation of the codebase for one migration instance. The ultimate objective at this stage is to achieve the best possible upgrade of the repository from its source version to the target version.

Following this, the Patch Evaluation stage is carried out by the automated BenchMAC harness. Here, the evaluation is conducted in a disciplined, fixed process that is identical for every submission on a given instance, ensuring results are fairly comparable. This evaluation takes place within a clean, standardized, and isolated Docker environment tailored specifically for the instance at hand. The harness receives two primary inputs: the formal instance definition and the patch artifact generated in the previous stage. It then clones the repository at the defined `base_commit`, applies the submitted patch, and executes a canonical set of shell commands (such as those for installation and building). Throughout the process, all outputs are recorded to compute objective metrics. The key goal of this stage is to rigorously and reproducibly verify the success of the migration.

3.1.1 The Unified Diff as a Standard Interface

The interface connecting these two decoupled stages is a single, standardized artifact: a unified diff file. The SUT's only obligation is to produce a diff that represents all proposed changes to the codebase. This simple yet powerful interface is the cornerstone of the benchmark's architecture.

This decoupling offers several significant advantages for scientific rigor and long-term viability:

- **Separation of Concerns:** Researchers can innovate on agent design without modifying the evaluation harness, and the harness can remain strict and reproducible across all SUTs.
- **Maintainability:** Improvements or bug fixes in the evaluation harness do not require re-engineering the generation methods, and vice versa.
- **Extensibility:** The architecture allows for incremental evolution. New metrics can be added to the evaluation stage without constraining future SUTs, while new agentic strategies can be tested against the existing, stable harness.

The foundation of any robust benchmark lies in the quality of its evaluation tasks. The following section details the principles and processes used to curate the BenchMAC dataset of migration instances, which form the basis of all experiments conducted in this thesis.

3.2 Dataset Curation

At this stage of the project, the curation of benchmark instances is a deliberate and manual process, prioritizing the quality and reproducibility of each task over the sheer quantity of the dataset. While the criteria employed could be automated in the future to expand the benchmark, this initial focus ensures a robust and reliable foundation for evaluation.

3.2.1 Instance Definition

An *instance* in BenchMAC is a formal definition of a self-contained migration task. It is composed of the following key elements:

- **Repository URL:** The location of the open-source Git repository.

- **Base Commit:** The specific commit hash that marks the state of the codebase before the migration.
- **Commands:** A set of shell commands required to interact with the project, including:
  - `install`: The command to install all project dependencies (e.g., `npm ci`).
  - `build`: The command to compile the application for production (e.g., `npx ng build`).
- **Source Angular Version:** The major version of Angular at the `base_commit`.
- **Target Angular Version:** The target major version for the migration task.
- **Dockerfile:** The path to a specific Dockerfile that defines the reproducible environment for this instance.

3.2.2 Curation Principles

The selection of projects and the creation of instances are guided by a set of strict principles designed to ensure the scientific validity of the benchmark.

Reproducibility Over Size

The primary guiding principle is to create tasks that are perfectly reproducible. This sometimes means favoring smaller, well-structured projects with clear build steps over larger, more complex codebases that might introduce environmental variability.

Open-Source and Licensing

All instances are derived from publicly available repositories on GitHub. We exclusively select projects with a permissive open-source license (e.g., MIT, Apache 2.0, or BSD) to ensure that the dataset can be freely used and distributed for research purposes.

Application-Centric Focus

The benchmark focuses on evaluating migrations of runnable web applications, not libraries or tools. This choice ensures that the SUT is tested on tasks that involve a complete front-end stack, including user interface components, templates, and application-specific build configurations.

Quality Signals

To filter for well-maintained projects, we use several proxies for quality. We require a minimum number of GitHub stars as an indicator of community usage and trust. Furthermore, the presence of a Continuous Integration (CI) configuration file (e.g., GitHub Actions workflows) is considered a strong positive signal, as it often simplifies the process of creating a correct and reproducible build environment.

Modern Angular Scope

The benchmark exclusively targets migrations for Angular version 9 and later. In the current industrial context, modernization efforts overwhelmingly start from these versions, making this scope highly relevant to real-world challenges.

Sandboxing Constraints

To ensure that each evaluation can run in a fully isolated, sandboxed environment, we exclude repositories with hard external dependencies. This includes projects that require access to databases, external API keys, or other secrets to complete their build process.

The "Green Baseline" Principle

A non-negotiable criterion for any instance is the existence of a "green baseline." This means that the project, at its specified `base_commit`, must install its dependencies and build successfully within our controlled environment *before* any migration is attempted. This principle is critical for experimental control for two reasons:

1. It ensures that any failure observed after the SUT has acted is attributable to the migration process itself, not to a pre-existing problem with the project.
2. It establishes a clear and valid starting point, allowing for fair and unambiguous judgment of whether the migration improved or degraded the codebase's state.

The BenchMAC test suite includes automated validation tests that enforce this principle by building every instance's environment and confirming that the baseline state is indeed "green."

Multiple Instances Per Repository

Open-source repositories are not static; their Git history naturally captures the evolution of a codebase across successive framework versions. We leverage this history to define multiple benchmark instances from a single repository. For example, one project can yield a task for v11 to v12, another for v12 to v13, and so on. This approach reduces the manual overhead of creating and maintaining environments (e.g., fewer Dockerfiles) while increasing the diversity of migration tasks in the dataset.

Silver Patches for Harness Validation

To ensure the benchmark harness itself is reliable, we generate *Silver Patches*. These are diffs extracted directly from the Git history of the curated projects, representing how human developers performed the real migration (e.g., the set of changes between the v14 and v15 release tags).

The purpose of a Silver Patch is not to serve as a unique or perfect solution for an SUT to replicate. Instead, it is an essential developer tool for **validating the correctness of the evaluation harness**. When the harness applies a Silver Patch, the subsequent install and build commands are expected to succeed. If they do not, it signals a problem with the benchmark's setup (e.g., a misconfigured Dockerfile or a bug in the harness) rather than a failure of the migration logic.

We use the term "Silver" instead of "Golden" to emphasize that a migration task can have multiple equally valid solutions. An SUT might produce a different, yet perfectly correct, set of changes. Silvers are therefore a tool for ensuring the reliability of our measurement system, not the reference against which AI agents are ultimately judged.

3.2.3 The BenchMAC v1.0 Dataset

At the time of writing, the initial dataset comprises 9 instances derived from a single, well-structured repository: `gothinkster/angular-realworld-example-app`.

- The instances cover migrations from Angular v11→v12 up to v19→v20.
- The codebase size ranges from approximately 1,400 to 1,500 lines of TypeScript code and 600 to 690 lines of HTML across these versions.

We acknowledge the limited size and diversity of this initial dataset, a point which will be discussed further in the Limitations section (Chapter 7). The complete dataset definitions are publicly available[1], with their corresponding Dockerfiles also available in the repository[2].

TABLE 3.1: BenchMAC v1.0 instances: repositories, base commits, and version jumps

Repository	Base Commit	Source→Target
gothinkster/angular-realworld-example-app	4f29e0e	11→12
gothinkster/angular-realworld-example-app	57eeb6a	12→13
gothinkster/angular-realworld-example-app	db8c6b0	13→14
gothinkster/angular-realworld-example-app	e2f6f4c	14→15
gothinkster/angular-realworld-example-app	e28c896	15→16
gothinkster/angular-realworld-example-app	bd914dc	16→17
gothinkster/angular-realworld-example-app	f218b2f	17→18
gothinkster/angular-realworld-example-app	2555e2f	18→19
gothinkster/angular-realworld-example-app	a6f16d0	19→20

3.3 Reproducible Environments

Similar to established benchmarks like HumanEval [4], SWE-Bench [6], and MigrationBench [94], BenchMAC relies on execution-based metrics to assess correctness. HumanEval executes generated functions to verify their outputs; SWE-Bench runs unit tests to validate bug fixes. Likewise, because Angular migrations can produce many syntactically valid but non-functional code variants, **BenchMAC relies on execution-based metrics to verify that the migrated project actually runs correctly, rather than merely "looking right" at the code level.**

When evaluation is based on execution, ensuring a consistent and reproducible environment is paramount. Minor differences in the environment such as the operating system, installed system packages, or the version of the Node.js runtime can lead to different outcomes for reasons unrelated to the SUT's performance. To eliminate this variability and ensure that results are fair and comparable, every evaluation in BenchMAC is conducted within a strictly controlled and reproducible containerized environment powered by Docker.

3.3.1 Rationale for Using Docker

Docker has become the de facto standard for packaging reproducible environments in both software engineering research and industrial practice. Its adoption for BenchMAC is motivated by several key advantages:

[1] https://github.com/SuperMuel/BenchMac/blob/main/data/instances.jsonl
[2] https://github.com/SuperMuel/BenchMac/tree/main/data/dockerfiles

- **Isolation and Immutability.** Containers provide a clean, isolated filesystem for each run, preventing state leakage between evaluations. Once an image is built from a Dockerfile, it is immutable, guaranteeing that the same environment can be instantiated every time, regardless of the host system.
- **Explicit Specification.** The Dockerfile serves as an explicit, version-controlled specification of the entire environment. Every dependency, from system packages and the Node.js version to project files and environment variables, is declared in code.
- **Consistency Over Time.** By pinning the versions of base images and all installed software, we ensure that an evaluation can be re-run months or years later under identical conditions, a cornerstone of scientific reproducibility.

3.3.2 Environment Configuration Details

Each benchmark instance is associated with a Dockerfile that defines its specific environment. While tailored to each project's needs, these definitions follow a common set of rigorous practices.

Repository Integration

Instead of cloning a full repository with its history, we directly download a tarball of the exact `base_commit` into the Docker image.

```
RUN curl -L https://github.com/<owner>/<repo>/archive/<
    base_commit>.tar.gz \
     | tar -xz --strip-components=1
```

This approach reduces network usage, build time, and final image size. Crucially, it also prevents an SUT from accessing the repository's Git history, which could otherwise be used to find a pre-existing solution by looking at future commits. After extraction, a new single-commit Git repository is initialized to simulate a minimal developer workspace.

Node.js Runtime and Package Manager

Angular projects have strict dependencies on specific Node.js versions. For each family of instances, the Dockerfile installs a Node.js version that is compatible with both the source and target Angular versions. All current instances use **npm** as the package manager, which is bundled with the selected Node.js distribution, ensuring its version is also implicitly pinned.

System Packages and Environment Variables

A minimal set of system packages, such as `git`, `python3`, and `build-essential`, are installed to support the build process. Standard environment variables are also set to ensure consistent behavior from command-line tools.

```
ENV CI=true \
    TZ=UTC \
    LANG=C.UTF-8 \
    NG_CLI_ANALYTICS=false \
    NPM_CONFIG_AUDIT=false \
    NPM_CONFIG_FUND=false
```

3.3.3 Enforcing Strict Reproducibility

To achieve the highest possible degree of determinism, we employ several advanced techniques that go beyond simple version pinning.

First, all benchmark instances are tied to a **pinned base image**, specified not only by its version tag but also by its unique SHA-256 digest. This prevents "tag drift," where an upstream provider might update an image tag (e.g., `node:18-bookworm-slim`) over time.

```
FROM node:18-bookworm-slim@sha256:f9ab18e3...
```

Second, for system packages, we configure the package manager to use **historical snapshots** of the underlying Linux distribution's repositories. Instead of downloading the "latest" package versions, which change daily, we fetch packages as they existed on a specific date. This guarantees that the exact same package versions are installed in every run, eliminating hidden variability from operating system updates.

```
RUN echo 'deb http://snapshot.debian.org/archive/debian/20250918
    T000000Z bullseye main contrib non-free' > /etc/apt/sources.
    list && \\
     echo 'deb http://snapshot.debian.org/archive/debian-security
        /20250918T000000Z bullseye-security main contrib non-
        free' >> /etc/apt/sources.list && \\
     echo 'Acquire::Check-Valid-Until "false";' > /etc/apt/apt.
        conf.d/99snapshot && \\
     apt-get update && apt-get install -y curl git ...
```

Together, these measures ensure the environment is fully deterministic. The same controlled containers provide SUTs with a stable workspace for patch generation and an identical, fair arena for patch evaluation. By relying on the exact same environment in both stages, we reduce variability and ensure that the patches produced by different systems can be directly and fairly compared. A representative Dockerfile is provided in Appendix A.1.

3.4 Systems Under Test (SUTs)

As stated previously, the BenchMAC harness is designed to be agnostic to the system being evaluated. This flexibility allows for the evaluation of a diverse range of approaches, including rule-based methods, simple LLM-powered scripts, complex AI agents, and commercial tools. The only strict requirement for an SUT to be compatible with the benchmark is its ability to produce a single unified diff file representing its proposed migration solution. A typical diff file might look like this:

```
diff --git a/package.json b/package.json
index 1234567..89abcde 100644
-- a/package.json
++ b/package.json
@@ -5,7 +5,7 @@
  "dependencies": {
     "@angular/core": "^14.0.0",
     "@angular/cli": "^14.0.0"
     "@angular/core": "^15.0.0",
     "@angular/cli": "^15.0.0"
  }
 }
```

This format is portable, language-agnostic, and compact, making it an ideal interface.

A naive "one-shot" approach, where the entire codebase is passed to an LLM to generate a complete diff in a single turn, is generally not viable. It is often prohibitively expensive, exceeds context window limits for large projects, and forgoes the critical feedback loop that human developers rely on (e.g., compiler errors, test results).

In practice, effective AI-driven coding solutions employ more advanced, "agentic" methods. An AI agent typically operates in an iterative loop over multiple turns. In each turn, it performs an action, observes the result, and uses that feedback to decide its next action [20]. For software engineering agents, this process almost always involves the use of tools, such as the ability to read and write files, search for code within the repository, and execute shell commands to run tests or compilers. An agent is therefore not just the LLM itself, but the entire system or "scaffolding" built around the LLM, which acts as its reasoning engine.

This complexity underscores the need for a robust benchmark. To build an effective AI-driven migration tool, one must optimize not only the choice of LLM but also the scaffolding, the available tools, and the prompting strategy. BenchMAC provides the necessary infrastructure to measure the impact of these variables.

3.4.1 Choice of Baseline SUTs

For this thesis, we implement and evaluate two distinct baseline systems to represent different ends of the automation spectrum.

1. **A Rule-Based Approach.** We use Angular Schematics, the framework's native, rule-based scaffolding tool. It automates project modifications by applying predefined transformations to the codebase's Abstract Syntax Tree (AST). This baseline helps answer the question: how much better are complex and expensive AI agents compared to existing, deterministic tools?

2. **A Minimalist AI Agent Framework.** To establish a simple AI baseline where the performance of the LLM and prompt are the primary variables, we use `mini-swe-agent`[95]. This framework is a minimal implementation (around 100 lines of code) of an agentic loop. It does not rely on model-specific tool-calling APIs, meaning any LLM can be used as its reasoning engine. The agent interacts with its environment by emitting shell commands in its output, making it easy to connect to our sandboxed Docker environments. As its authors note, this simplicity makes it "perfect as a baseline system and for a system that puts the language model (rather than the agent scaffold) in the middle of our attention."

While more sophisticated, production-grade AI tools exist (e.g., Claude Code, Gemini CLI, Cursor CLI), automating and sandboxing them presents significant engineering challenges. Therefore, their implementation is considered outside the scope of this initial research.

3.5 Stage 1: Patch Generation

This section details the concrete pipeline used by our baseline SUTs to generate a migration patch.

3.5.1 Pipeline for the AI Agent (Mini-SWE-Agent)

The process for generating a patch with the AI agent involves several automated steps.

1. **Environment Preparation.** A fresh Docker container is provisioned from the instance-specific image. This provides the agent with a sandboxed virtual machine containing the codebase and all necessary developer tools.

2. **Agent Initialization.** The agent is initialized with a system prompt. The prompt is intentionally minimalist and contains:

   - **The Task:** A clear instruction, e.g., "Migrate the application from Angular version {source_version} to {target_version}."
   - **Context:** Information about the location of the codebase within the container (`/app/project`).
   - **Tool Instructions:** Instructions specific to the `mini-swe-agent` scaffold, explaining how to format thoughts and how to execute shell commands to read, write, and edit files.

   The full system prompt is available in Appendix B.1. This minimal design serves as a baseline to understand core model capabilities before introducing more sophisticated prompting strategies.

3. **Agent-Environment Connection.** The `mini-swe-agent` framework runs on the host machine, while all shell commands emitted by the agent are securely redirected for execution inside the Docker container. This separation ensures that the agent's internal logic does not interfere with the pristine migration environment, and it simplifies the collection of artifacts like command logs and agent reasoning traces.

4. **Execution of the Agent Loop.** The agent begins its iterative process of analyzing the codebase, editing files, and running commands. To prevent infinite loops and manage costs, a "maximum steps" parameter is set to terminate the process if it runs for too long. An example of a full agent trace is provided in Appendix C.1 to illustrate this interactive process.

5. **Artifact Capture.** Once the agent terminates, all changes it has made to the codebase must be captured. To do this reliably, we initialize a Git repository inside the container at startup and create a single initial commit tagged as `baseline`. The following commands are run when building the Docker image:

   ```
   # Initialize git repo and create baseline commit for
     diffing
   RUN git init && \
       git config --global --add safe.directory /app/project
         && \
       git config user.email "benchmark@benchmac.dev" && \
       git config user.name "BenchMAC Baseline" && \
       git add . && \
       git commit -m "first commit" && \
       git tag baseline
   ```

   After the agent completes its work, we compute a diff against this initial state using `git diff baseline`. This command captures all committed, staged, and unstaged changes, producing the final unified diff artifact.

6. **Teardown.** With the diff file saved, the container has served its purpose and is discarded.

3.5.2 Pipeline for the Rule-Based Baseline (Angular Schematics)

The process for the rule-based baseline is designed to mirror the AI agent pipeline, ensuring the only variable is the migration logic itself. Instead of an iterative AI loop, this SUT executes a predefined, deterministic sequence of shell commands.

1. **Environment Preparation.** This step is identical to the AI agent's setup. A fresh Docker container is provisioned with the codebase and a Git repository tagged at the `baseline` commit.

2. **Migration Plan Definition.** The "plan" for this SUT is a static sequence of two commands representing the standard, officially recommended procedure for an automated migration:

   (a) Install Dependencies: `npm ci` (or `npm ci --legacy-peer-deps` as required). This is a prerequisite to ensure the Angular CLI is available.

   (b) Run Schematics:

   ```
   ng update @angular/core@{target_version} @angular/cli@{
       target_version} --allow-dirty --force
   ```

3. **Plan Execution.** The harness executes these commands sequentially inside the container. A failure during the `install` command is treated as a critical harness error. However, a failure during the `ng update` command is considered a normal, expected outcome, providing valuable data that the automated schematics were insufficient for this task. The process continues regardless to capture any partial changes made.

4. **Artifact Capture.** This step is identical to the AI pipeline. The command `git diff baseline` is run to capture all modifications made by the schematics tool.

5. **Teardown.** The container is discarded, leaving the generated diff file ready for evaluation.

Having described how SUTs operate within a controlled container to produce a single, unified diff artifact, we now turn to the second stage: evaluating that artifact under identical, repeatable conditions.

3.6 Stage 2: Patch Evaluation

Once the patch generation phase concludes, the SUT has produced its single artifact: a unified diff that encodes its proposed migration. The Patch Evaluation stage is where the BenchMAC harness takes over, applying this artifact in a clean, controlled environment and computing objective metrics based on the outcome. This process is fully automated and identical for every submission.

3.6.1 Evaluation Pipeline

The evaluation workflow proceeds as follows:

1. **Container Provisioning.** A fresh, new Docker container is provisioned from the exact same instance-specific image used during the generation phase. This ensures environmental consistency. The repository code is present at its original `base_commit`.

2. **Patch Application.** The submitted diff file is copied into the container. The harness first checks if the patch can be applied cleanly using the command `git apply --check`. This step validates the integrity of the diff itself and guards against malformed submissions. Such errors are a known risk when SUTs generate patches directly, for instance, through token-by-token LLM inference, an approach that Mündler et al. [96] demonstrated is prone to producing syntactically invalid diffs. If the check fails, the evaluation for that instance halts, and the patch application is marked as a failure. If it succeeds, the patch is applied with `git apply`. The codebase is now in its post-migration state as proposed by the SUT.

3. **Execution of Evaluation Commands.** The harness executes a predefined, canonical sequence of shell commands to assess the health of the migrated codebase. The exit code, standard output (`stdout`), and standard error (`stderr`) of each command are meticulously recorded for later analysis.

4. **Teardown.** Once all commands have been executed and their outputs stored, the container is discarded.

5. **Metric Computation.** Finally, the harness computes a set of metrics based solely on the stored artifacts from the command executions.

3.6.2 Canonical Evaluation Sequence

To determine if a migration was successful, the harness runs a short, standardized sequence of commands. This sequence is defined per instance to allow for flexibility, though in practice it is consistent across most instances. The process follows a fail-fast policy, where a failure at any stage prevents subsequent stages from running.

TABLE 3.2: Canonical Evaluation Command Sequence

Order	Stage	Default Command
1	Install Dependencies	`npm ci`
2	Version Check	`npm ls @angular/cli @angular/core --json`
3	Build	`npx ng build --configuration production`

Dependency Installation Stage. For dependency installation, we use `npm ci`. Unlike `npm install`, this command is designed for reproducible, automated environments. It installs dependencies matching the exact versions specified in the project's lockfile, ensuring consistency and failing immediately if the `package.json` and `package-lock.json` files are out of sync. For older Angular projects ($\leq$v14), we use the `--legacy-peer-deps` flag to bypass strict peer dependency checks that were not enforced by older versions of npm, thus avoiding false negatives while still ensuring a reproducible installation from the lockfile.

Version Check Stage. After installation, the harness verifies that the migration successfully updated the core framework dependencies. The command `npm ls @angular/cli @angular/core --json` lists the resolved versions of Angular's two central packages. The `--json` flag provides structured output that the harness can parse deterministically to confirm that the installed major versions match the target version specified by the instance.

Build Stage. The final and most critical step is the build stage. The command `npx ng build --configuration production` invokes the Angular CLI to transpile, bundle, and optimize the application source code into production-ready artifacts. A successful build provides a strong signal that the migrated codebase is at least syntactically consistent and structurally sound. A build failure indicates that the migration introduced a fundamental error, such as a missing import, the use of a deprecated API, or an incompatible library.

3.6.3 Evaluation Metrics

Metrics are derived programmatically from the stored outputs of the evaluation commands. This design means that the virtual machine is not needed for metric computation, and metrics can be re-calculated at any time from the raw logs.

- **Patch Application Success.** A boolean metric indicating if the submitted patch applied cleanly. It is measured by checking the exit code of the `git apply --check` command.
- **Install Success.** A boolean metric indicating if all dependencies were installed successfully. It is measured by the exit code of the `npm ci` command.
- **Target Version Achieved.** A boolean metric indicating if the installed versions of `@angular/core` and `@angular/cli` match the target major version. It is measured by parsing the JSON output of the `npm ls` command and comparing the resolved major versions. The metric is `True` if they match, `False` if they do not, and `None` if the command failed or could not be run.
- **Build Success.** A boolean metric indicating if the application compiles successfully with production settings. It is measured by the exit code of the `npx ng build` command. The value is `None` if a prior stage failed.

At this stage of the benchmark's development, `BuildSuccess` represents the highest tier of success that can be measured. It confirms the syntactic and structural integrity of the migrated codebase. However, a successful build is not synonymous with a complete or correct migration, as it does not account for functional correctness, migration completeness, or code quality. Integrating test suite execution as a further validation step is a planned extension discussed in Chapter 8.

3.7 Scope and Methodological Boundaries

To ensure a rigorous and focused evaluation in its initial version, the BenchMAC methodology is defined by the following scope and boundaries.

- **End-to-End, Single-Version Jumps.** BenchMAC evaluates the SUT's ability to perform a complete, repository-level migration for a single major version jump (e.g., v11→v12). It does not provide granular diagnostics on sub-tasks or evaluate multi-version leaps.

- **Primary Metric is Build Success.** The highest tier of measured success is a clean production build. The execution of unit or end-to-end tests is deliberately excluded at this stage to avoid the significant environmental complexities they introduce.

- **"Green Baseline" Prerequisite.** The methodology requires that all instances start from a state that is known to install and build successfully, inherently biasing the dataset towards well-maintained repositories.

- **Standard Angular CLI Repositories.** The current scope is limited to single-application repositories following the standard Angular CLI structure, excluding complex monorepos.

- **Quality-Over-Quantity Dataset.** The initial dataset is curated manually to ensure high quality and reproducibility, resulting in a smaller set of instances primarily derived from a single repository as a proof-of-concept.

- **Evaluation of CLI-Based Systems.** The benchmark is designed for SUTs that can be automated via a command-line interface, which naturally excludes direct evaluation of interactive, IDE-integrated systems.

These design choices prioritize foundational reliability and controlled experimentation, establishing a solid baseline upon which future extensions can be built.

Chapter 4

Experiments

4.1 Experimental Design and Objectives

The primary objective of our experiments is to evaluate the capability of diverse AI systems to perform repository-level Angular migrations, comparing them against a rule-based baseline. We aim to provide a comprehensive landscape view by testing a wide range of modern LLMs across multiple dimensions of variation.

4.1.1 Evaluation Philosophy

Our experimental design follows a **breadth-first** rather than **depth-first** strategy. We prioritize evaluating many different models with standard configurations over exhaustively tuning a few models. This approach serves two purposes:

1. **Practical Guidance**: Organizations considering AI-assisted migration tools need comparative data across available options to inform procurement and adoption decisions.

2. **Research Baseline**: By establishing performance across a diverse model landscape, we provide reference points for future research on prompting strategies, agent architectures, and specialized fine-tuning.

Each system under test performs a single migration attempt per benchmark instance, using default or near-default inference parameters. Beyond the core evaluation metrics defined in Chapter 3, we also collect supplementary statistics including execution time, total cost (in USD) and the number of LLM calls.

4.1.2 Dimensions of Model Diversity

To ensure comprehensive coverage of the LLM landscape, we deliberately select models that vary across five key dimensions:

Ownership Model: We include both closed-source proprietary systems (OpenAI GPT-5, Anthropic Claude, Google Gemini) and open-source models (Mistral Small variants) to assess whether accessibility and transparency trade off against performance.

Model Scale: Our selection spans from small, efficient models (GPT-5-Nano, Devstral-Small) to very large flagship systems (GPT-5, Claude Sonnet 4.5, Gemini 2.5-Pro), testing whether the additional computational cost of larger models translates to better migration outcomes.

Reasoning Capability: We compare **reasoning models** that perform explicit internal reasoning before responding (OpenAI GPT-5 series, Mistral Magistral series, XAI Grok-4 variants, Google Gemini 2.5) against **non-reasoning models** that generate responses directly (e.g Mistral base series). This distinction is particularly relevant for multi-step tasks like code migration that require planning.

Domain Specialization: We contrast **general-purpose** models trained on broad data distributions against **coding-specialized** models explicitly optimized for software engineering tasks (GPT-5-Codex, Devstral series, Grok-Code-Fast). Coding-specialized models are typically trained on larger proportions of code or have been optimized for command-line and agentic coding workflows.

Release Timing: All evaluated models were released in 2025, ensuring they represent the current state-of-the-art and have comparable training data cutoffs.

4.2 Systems Under Test

Our experimental corpus comprises **21 distinct system configurations** spanning five major AI providers, including a rule-based baseline. Table 4.1 provides a complete specification of all evaluated systems.

4.2.1 OpenAI Models

OpenAI contributes four model variants to our evaluation, all from the GPT-5 family released in 2025:

GPT-5 (gpt-5-2025-08-07): The flagship general-purpose model, representing the largest and most capable system in OpenAI's lineup. It employs internal reasoning mechanisms controlled by a `reasoning_effort` parameter, which we set to `medium` (the default). A `verbosity` parameter, also set to `medium`, controls the detail level of generated responses.

GPT-5-Mini (gpt-5-mini-2025-08-07): A smaller, more cost-effective variant of GPT-5 that maintains reasoning capabilities while reducing inference latency and cost. It uses identical parameter settings to the full GPT-5 model.

GPT-5-Nano (gpt-5-nano-2025-08-07): The most compact model in the GPT-5 family, designed for scenarios where speed and cost are prioritized over maximum capability. Configuration matches GPT-5 and GPT-5-Mini.

GPT-5-Codex (gpt-5-codex): A specialized variant explicitly optimized for agentic coding tasks and command-line interactions. It shares the same parameter configuration as other GPT-5 variants.

All GPT-5 models currently do not expose `temperature`, `top_p`, or other sampling parameters, as these are managed internally by the provider. We set `seed=42` for reproducibility where applicable.

4.2.2 XAI (Grok) Models

XAI provides four Grok-4 variants that allow us to directly compare reasoning versus non-reasoning architectures:

Grok-4 (grok-4-0709): The primary flagship model with reasoning capabilities. We set `temperature=0.0` for deterministic behavior.

Grok-Code-Fast (grok-code-fast-1-0825): A coding-optimized reasoning model designed for fast inference while maintaining reasoning capability. Configured with `temperature=0.0`.

Grok-4-Fast-Reasoning (grok-4-fast-reasoning): A streamlined reasoning variant that balances speed and performance capability. Configured with `temperature=0.0`.

Grok-4-Fast-Non-Reasoning (grok-4-fast-non-reasoning): The same underlying architecture as Grok-4-Fast-Reasoning, but with reasoning explicitly disabled. We set `temperature=0.0`.

The XAI suite is particularly valuable because it enables controlled comparison: Grok-4-Fast-Reasoning and Grok-4-Fast-Non-Reasoning share the same base model but differ only in their reasoning mechanism, allowing us to isolate the impact of reasoning on migration performance.

4.2.3 Mistral Models

Mistral AI contributes six models, split evenly between reasoning-capable (Magistral) and non-reasoning variants (Mistral base, Devstral):

Reasoning Models:

- **Magistral-Medium (magistral-medium-2509)**: A proprietary reasoning model (version 1.2), Mistral's flagship commercial offering. Configured with `temperature=0.7` (default), `top_p=1.0`, `random_seed=42`.
- **Magistral-Small (magistral-small-2509)**: An open-source reasoning model (version 1.2) released under the Apache 2.0 license, making it the only freely accessible reasoning model in our evaluation. Configured with `temperature=0.7`, `top_p=0.95`, `random_seed=42` following Mistral's recommendations.

General-Purpose Models:

- **Mistral-Medium (mistral-medium-2508)**: A mid-tier general-purpose model (version 3.1) without reasoning capabilities. Configured with `temperature=0.3`, `top_p=1.0`, `random_seed=42`.
- **Mistral-Small (mistral-small-2506)**: An open-source general-purpose model (version 3.2) under Apache 2.0 license. Configured with `temperature=0.3`, `top_p=1.0`, `random_seed=42`.

Coding-Specialized Models:

- **Devstral-Medium (devstral-medium-2507)**: A proprietary model optimized for agentic coding workflows. Configured with `temperature=0.15` (recommended), `top_p=1.0`, `random_seed=42`.
- **Devstral-Small (devstral-small-2507)**: A smaller version (version 1.1) under Apache 2.0 license. Configured with `temperature=0.15` (recommended), `top_p=1.0`, `random_seed=42`.

An important advantage of Mistral's reasoning models is their transparency: the full internal reasoning trace is accessible even in commercial models like Magistral-Medium, enabling qualitative analysis of model decision-making.

4.2.4 Anthropic Models

Anthropic contributes three models from the Claude 4 family:

Claude-Opus-4.1 (claude-opus-4-1-20250805): The larger, most capable model in the Claude 4 series, representing Anthropic's flagship offering for complex reasoning tasks. Configured with `temperature=0.0`.

Claude-Sonnet-4 (claude-sonnet-4-20250514): An efficient variant designed to balance capability with reduced inference cost and latency. Configured with `temperature=0.0`.

Claude-Sonnet-4.5 (claude-sonnet-4-5-20250929): The latest and most capable variant in the Sonnet series, released September 2025. Configured with `temperature=0.0`.

We do not use the "Extended Thinking" [97] mode for these models, making them non-reasoning. Additionally, we do not run experiments with the smallest model in the series, Claude 3.5 Haiku, as it was only released in 2024 and falls outside the scope of our evaluation.

4.2.5 Google Gemini Models

Google provides three Gemini 2.5 variants spanning different capability-cost trade-offs:

Gemini-2.5-Pro (gemini-2.5-pro): The flagship model, optimized for complex tasks requiring deep reasoning. Configured with `temperature=0.0` and `thinkingBudget=-1`, which enables dynamic thinking mode where the model adjusts its reasoning budget based on request complexity (default behavior for this model).

Gemini-2.5-Flash (gemini-2.5-flash): A mid-tier model balancing performance and efficiency. Configured with `temperature=0.0` and `thinkingBudget=-1`, enabling dynamic thinking mode (default behavior for this model).

Gemini-2.5-Flash-Lite (gemini-2.5-flash-lite): The most lightweight variant, prioritizing speed and cost-effectiveness. Configured with `temperature=0.0` and `thinkingBudget=0`, which disables thinking and is the default behavior for this model.

4.2.6 Rule-Based Baseline: Angular Schematics

As described in Section 3.4.1, we include Angular Schematics as a deterministic, rule-based baseline.

This baseline serves two critical purposes:

1. **Floor Performance**: It establishes the minimum capability that AI systems should exceed to justify their complexity and cost.

2. **Capability Comparison (Hypothesis)**: We hypothesize that, because AI agents have the freedom to execute any commands (including `ng update`), they theoretically represent a superset of the rule-based approach and should achieve at least equivalent performance.

4.2.7 Cost and Step Constraints

To ensure fairness and prevent runaway experiments, all AI systems operated under standardized cost and step limits.

Each system was restricted to a **maximum of 100 steps** (i.e 100 LLM calls) per migration attempt and a **maximum API budget of 1.00 USD per run**.

If a system reached the step limit without completing the migration, the attempt was marked as *incomplete*. Similarly, if a model's accumulated API cost exceeded the budget threshold, the experiment was halted, and results were recorded as *terminated due to cost constraint*. Rule-based baselines such as Angular Schematics were exempt from these limits as they are deterministic and cost-free.

4.2.8 Summary of System Configurations

Table 4.1 provides a complete overview of all 21 evaluated systems. The table specifies the exact model identifier, parameter settings, and key characteristics for each configuration.

TABLE 4.1: Complete Specification of Systems Under Test

Provider	**Model**	**Reasoning**	**Open**	**Parameters**
OpenAI	gpt-5-2025-08-07	Yes	No	`verbosity=medium`, `reasoning_-effort=medium`, `seed=42`
OpenAI	gpt-5-mini-2025-08-07	Yes	No	`verbosity=medium`, `reasoning_-effort=medium`, `seed=42`
OpenAI	gpt-5-nano-2025-08-07	Yes	No	`verbosity=medium`, `reasoning_-effort=medium`, `seed=42`
OpenAI	gpt-5-codex	Yes	No	`verbosity=medium`, `reasoning_-effort=medium`, `seed=42`
XAI	grok-4-0709	Yes	No	`T=0.0`
XAI	grok-code-fast-1-0825	Yes	No	`T=0.0`
XAI	grok-4-fast-reasoning	Yes	No	`T=0.0`
XAI	grok-4-fast-non-reasoning	No	No	`T=0.0`
Mistral	magistral-medium-2509	Yes	No	`T=0.7`, `P=1.0`, `seed=42`
Mistral	magistral-small-2509	Yes	Yes	`T=0.7`, `P=0.95`, `seed=42`
Mistral	mistral-medium-2508	No	No	`T=0.3`, `P=1.0`, `seed=42`
Mistral	mistral-small-2506	No	Yes	`T=0.3`, `P=1.0`, `seed=42`
Mistral	devstral-medium-2507	No	No	`T=0.15`, `P=1.0`, `seed=42`
Mistral	devstral-small-2507	No	Yes	`T=0.15`, `P=1.0`, `seed=42`
Anthropic	claude-opus-4-1-20250805	No	No	`T=0.0`
Anthropic	claude-sonnet-4-20250514	No	No	`T=0.0`
Anthropic	claude-sonnet-4-5-20250929	No	No	`T=0.0`
Google	gemini-2.5-pro	Yes	No	`T=0.0`, `thinkingBudget=-1`
Google	gemini-2.5-flash	Yes	No	`T=0.0`, `thinkingBudget=-1`
Google	gemini-2.5-flash-lite	No	No	`T=0.0`, `thinkingBudget=0`
Baseline	angular-schematics	N/A	Yes	(deterministic)

4.3 Research Questions

Our experiments are guided by the following research questions:

- **RQ1: Can BenchMAC meaningfully differentiate between migration systems?**

Rationale: A useful benchmark must discriminate between approaches, revealing performance differences across systems with varying capabilities.

- **RQ2: What is the baseline performance of modern LLMs on Angular migrations, and how do they compare to rule-based approaches?**

  Rationale: Establishing baseline performance ranges and cost-effectiveness profiles enables practitioners to make informed tool selection decisions and provides reference points for future research.

4.4 Summary

This chapter has detailed the design and execution of a comprehensive evaluation spanning 20 AI configurations plus a rule-based baseline, across 9 real-world Angular migration tasks (171 total experiment runs). Our experiments prioritize breadth (many models, standard settings) over depth (single model, extensive tuning) to provide a landscape view of current capabilities.

Key design principles include:

- **Diversity**: Models vary by size, specialization, reasoning capability, and license
- **Fairness**: Identical task prompts, environments, and evaluation procedures for all systems
- **Transparency**: All configurations follow the reproducible methodology established in Chapter 3

The next chapter presents the results of these experiments, analyzing which systems succeeded, at what cost, and revealing systematic patterns of failure.

Chapter 5

Results

This chapter presents the empirical findings from the evaluation of 19 automated migration systems on the BenchMAC v1.0 benchmark. After filtering two agents that exceeded cost thresholds (see Section 4.2.7), the final analysis comprises 19 agents evaluated across 9 distinct migration instances, yielding 171 experiment runs. The primary metric is **build success**, which indicates whether an agent's proposed patch resulted in a clean, production-ready build of the application.

5.1 Overall Performance Landscape

Table 5.1 presents the overall performance leaderboard, ranking agents by their build success rate with 95% Wilson confidence intervals to reflect the statistical uncertainty inherent in the dataset size.

TABLE 5.1: Agent Performance Leaderboard (ranked by build success rate)

Rank	Agent	Build Success	95% CI
1	angular-schematics/789e301f	100.0%	[70.1, 100.0]
1	anthropic/claude-sonnet-4-20250514	100.0%	[70.1, 100.0]
1	anthropic/claude-sonnet-4-5-20250929	100.0%	[70.1, 100.0]
1	gemini/gemini-2.5-flash	100.0%	[70.1, 100.0]
1	gemini/gemini-2.5-pro	100.0%	[70.1, 100.0]
1	openai/gpt-5-codex	100.0%	[70.1, 100.0]
7	xai/grok-4-fast-non-reasoning	88.9%	[56.5, 98.0]
7	xai/grok-code-fast-1-0825	88.9%	[56.5, 98.0]
7	mistral/mistral-medium-2508	88.9%	[56.5, 98.0]
10	openai/gpt-5-2025-08-07	77.8%	[45.3, 93.7]
10	mistral/magistral-medium-2509	77.8%	[45.3, 93.7]
12	mistral/magistral-small-2509	66.7%	[35.4, 87.9]
13	mistral/devstral-medium-2507	55.6%	[26.7, 81.1]
13	openai/gpt-5-mini-2025-08-07	55.6%	[26.7, 81.1]
13	xai/grok-4-fast-reasoning	55.6%	[26.7, 81.1]
16	gemini/gemini-2.5-flash-lite	44.4%	[18.9, 73.3]
17	openai/gpt-5-nano-2025-08-07	22.2%	[6.3, 54.7]
17	mistral/mistral-small-2506	22.2%	[6.3, 54.7]
19	mistral/devstral-small-2507	0.0%	[0.0, 29.9]

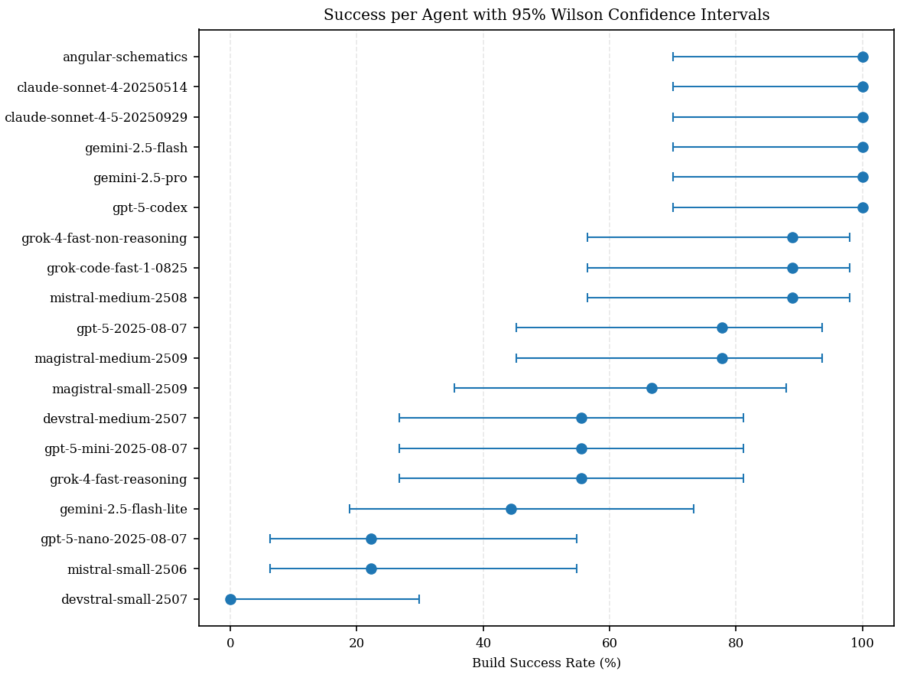

FIGURE 5.1: Build success rate per agent with 95% Wilson confidence intervals. Agents are ranked from highest (top) to lowest (bottom) success rate. Error bars represent statistical uncertainty given the 9-trial sample size for each agent.

Top-Tier Performers: Six agents, including the rule-based `angular-schematics` baseline, achieved a perfect 100% success rate. While this point estimate is perfect, the 95% confidence intervals ([70.1%, 100.0%]) indicate that with only 9 trials, their true success rate is statistically indistinguishable from other high-performing models. This top tier includes a mix of general-purpose and code-specialized models, such as `gemini-2.5-pro` and `openai/gpt-5-codex`. The performance of the `angular-schematics` baseline is particularly notable, as it sets a high bar for both success and efficiency.

High-Performing Tier: A second group of agents demonstrated strong, reliable performance with success rates between 77.8% and 88.9%. This group includes models like `xai/grok-4-fast-non-reasoning` and `mistral/mistral-medium-2508`. The confidence intervals for this tier (e.g., [56.5%, 98.0%] for 8/9 success) overlap with the top tier, suggesting that the performance gap may not be statistically significant with the current dataset.

Mid- and Low-Tier Performers: The benchmark clearly separated a third tier of agents that struggled with the migration tasks. Models such as `mistral/magistral-small-2509` (66.7%) and `openai/gpt-5-mini` (55.6%) had markedly lower success rates. At the bottom end, `mistral/devstral-small-2507` failed on all 9 instances, achieving a 0% success rate with a confidence interval of [0.0%, 29.9%], indicating a statistically significant lack of capability for this task compared to the other tiers.

While this leaderboard summarizes overall success, the underlying data reveals important patterns related to model architecture and cost, which are explored in the following sections.

5.2 Comparative Analysis of Model Characteristics

An analysis of the leaderboard reveals that performance is not simply a function of model provider or cost. However, the influence of architectural characteristics such as reasoning capabilities, domain specialization, and model scale must be interpreted with caution due to the statistical uncertainty associated with the small number of trials.

5.2.1 The Impact of Reasoning Capabilities

A direct comparison within the XAI Grok family yielded a notable, though not statistically significant, result. The non-reasoning `grok-4-fast-non-reasoning` achieved a build success rate of 88.9% (8/9), while its reasoning-enabled counterpart, `grok-4-fast-reasoning`, achieved only 55.6% (5/9).

However, a comparison of their 95% confidence intervals, [56.5%, 98.0%] for the non-reasoning model and [26.7%, 81.1%] for the reasoning model—shows considerable overlap. Therefore, while there is a trend suggesting the non-reasoning model performed better on these tasks, **we cannot conclude with statistical confidence that its true success rate is higher.** This finding suggests that for the straightforward, iterative workflow required by these instances, the additional overhead of a reasoning mechanism may not have provided a clear benefit and could have potentially hindered performance.

5.2.2 The Role of Domain Specialization

The effectiveness of code-specific models appears to be highly context-dependent, with no clear, statistically significant trend emerging from the data.

For the OpenAI models, the specialized `gpt-5-codex` was a top performer with a 100% success rate ([70.1%, 100.0%]), outperforming the general-purpose `gpt-5`, which scored 77.8% ([45.3%, 93.7%]). Although `gpt-5-codex` had a higher point estimate, **the overlapping confidence intervals mean we cannot definitively state that specialization was the cause of its superior performance.**

Conversely, for Mistral models, the data suggests a trend in the opposite direction. The general-purpose `mistral-medium-2508` (88.9%, CI [56.5%, 98.0%]) outperformed the code-specialized `devstral-medium-2507` (55.6%, CI [26.7%, 81.1%]). While the point estimates are far apart, their confidence intervals still overlap. **This prevents a strong statistical claim, but the consistent pattern across both medium and small Mistral models suggests that for this benchmark, the general-purpose models were more effective.**

5.3 Instance-Level Difficulty Analysis

Before presenting the results, it is crucial to clarify the scope of this analysis. This section compares the *relative difficulty of the specific migration instances within the benchmark*, not the absolute difficulty of upgrading between Angular versions in general. For example, if the v16→v17 instance in our dataset appears more challenging than the v18→v19 instance, this reflects the specific state of the codebase for those tasks. It does not imply that upgrading from v16 to v17 is inherently harder than from v18 to v19 for all projects.

With this context established, the analysis reveals a clear difficulty gradient across the benchmark instances. The observed failure rate (our proxy for difficulty) ranges from a low of 10.5% for the v16→v17 migration to a high of 47.4% for the v18→v19 migration. This wide variance demonstrates that BenchMAC is not monolithic; it successfully presents tasks of varying complexity to the agents, which is a desirable characteristic for a robust evaluation tool.

However, this trend must be interpreted with statistical caution. As shown in Figure 5.2, the 95% confidence intervals for the difficulty of all instances exhibit considerable overlap. For instance, the interval for the easiest task, [2.9%, 31.4%], overlaps with that of the hardest task, [27.3%, 68.3%]. Therefore, while the data strongly suggests a difficulty gradient, **we cannot conclude with 95% statistical confidence that any single instance is definitively harder than another.** This highlights that a larger and more diverse pool of agents would be required to achieve the statistical power needed for a more fine-grained and definitive ranking of task difficulty.

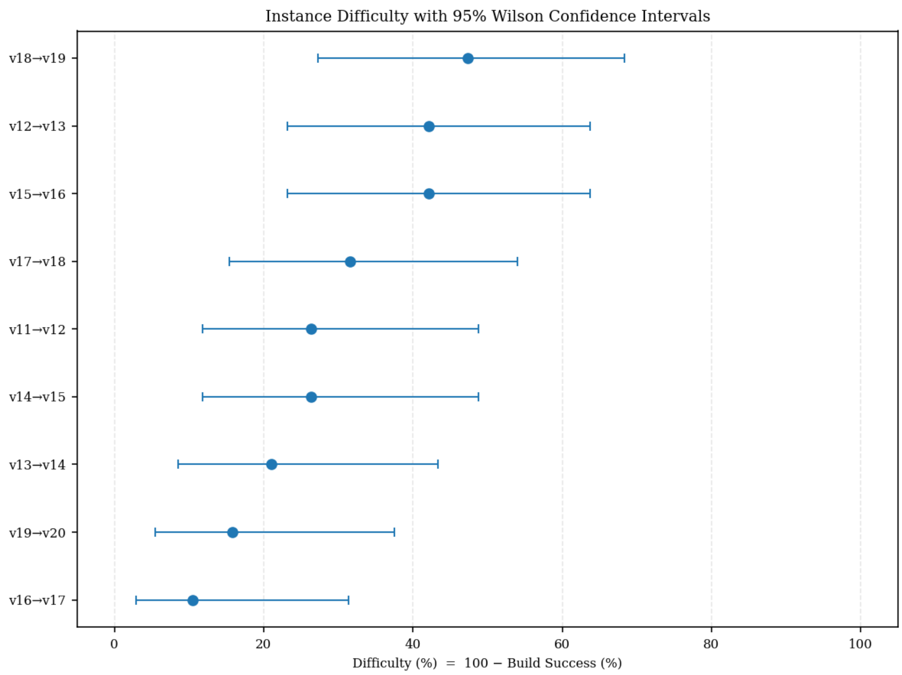

FIGURE 5.2: Instance difficulty (computed as 1 - build success rate) with 95% Wilson confidence intervals. Instances are sorted from hardest (highest difficulty) at the top to easiest at the bottom. Error bars represent statistical uncertainty in the difficulty estimate for each instance.

5.4 Economic and Efficiency Analysis

Beyond correctness, a critical dimension of agent performance is its economic and computational efficiency. An effective agent must not only solve the problem but do so within reasonable cost and time constraints. To analyze these trade-offs, we evaluated each agent's build success rate against its average API cost and examined other efficiency metrics such as the number of steps required.

Figure 5.3 plots the build success rate against the average cost for each of the 19 agents.

The analysis of efficiency and cost reveals several key findings:

1. No Correlation Between Cost and Success: The most striking finding is the clear lack of correlation between the cost of an agent and its performance on the

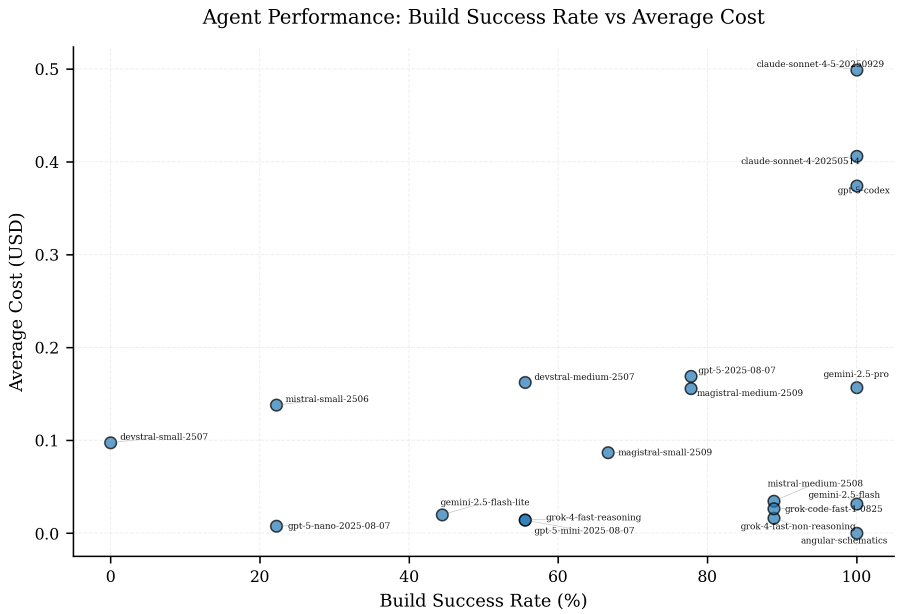

FIGURE 5.3: Build success rate (x-axis) versus average experiment cost in USD (y-axis) for each agent. Each point shows one agent, labeled by model name. Agents in the lower-right are both low-cost and reliable, while those in the upper-right are reliable but more expensive.

benchmark. As shown in Figure 5.3, the results are distributed across three quadrants. This strongly suggests that for this task, factors like model architecture and training data are more significant drivers of performance than an agent's price point alone.

2. A Clear "Efficient Frontier": The plot highlights a group of agents in the bottom-right quadrant that represent the most desirable balance of high success and low cost. The `angular-schematics` tool is the theoretical ideal, delivering 100% success at zero API cost. Among the LLM-driven agents, `gemini-2.5-flash` stands out as a clear leader on this frontier, achieving a perfect 100% success rate at a remarkably low average cost of only $0.032 per run. Other models, like `xai/grok-4-fast-non-reasoning`, also proved highly efficient, delivering an 88.9% success rate for an average cost of just $0.016.

3. The High Price of Top Performance: While several agents achieved a 100% success rate, they did so at vastly different price points. For example, `claude-sonnet-4-5-20250929` and `openai/gpt-5-codex` were also perfect performers, but their average costs of $0.499 and $0.374, respectively, were more than an order of magnitude higher than that of `gemini-2.5-flash`. This demonstrates a critical trade-off for practitioners: while multiple solutions may be effective, their operational costs can vary dramatically.

4. Step Economy vs. Iterative Flexibility: Another aspect of efficiency is how many steps an agent needs to complete a migration. The `angular-schematics` tool always uses exactly 2 steps (`npm ci` and `ng update`), since it follows a fixed sequence of commands. In comparison, successful AI agents required a median of 15 steps, and some attempts needed more than 50 steps (see Figure 5.4). This highlights the contrast between a straightforward rule-based approach and the more exploratory, iterative process of AI agents. While this flexible approach allows agents to debug, recover from errors, and adapt to the specific details of each project, it usually leads to more steps, longer duration, and greater computational cost.

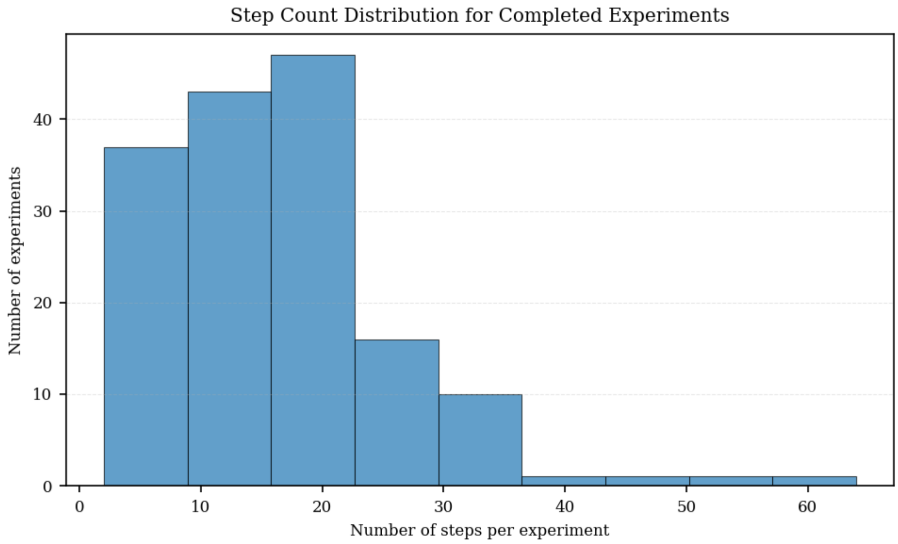

FIGURE 5.4: Distribution of step counts for experiments (experiments halted by the step limit are not shown)

5.5 Analysis of Experiment Failures

During our experiments, we encountered two major failure modes among AI agents: step limit exceeded and empty patch submissions.

5.5.1 Step Limit Exceeded

In the mini-SWE-agent framework, a "step" corresponds to a single command execution in the agent's iterative loop. To prevent unbounded costs and execution times, we set a maximum limit of 100 steps per experiment. If an agent did not complete the migration within this limit, the execution was terminated early.

We identified 14 experiments (8.2% of total runs) that were interrupted due to exceeding this step limit. These failures were highly concentrated among smaller Mistral models, as shown in Table 5.2.

TABLE 5.2: Distribution of step limit failures across agents

Agent	Failures
mistral/devstral-small-2507	7
mistral/mistral-small-2506	4
mistral/devstral-medium-2507	2
gemini/gemini-2.5-flash-lite	1

To verify that these interruptions were justified, we performed cycle detection on the command execution traces. We analyzed the final 50 steps of each terminated experiment to identify repeating command patterns. A cycle was defined as a sequence of n consecutive commands that repeats k times.

In 13 of 14 cases, agents were caught in clear repetitive loops, executing the same 1-4 commands between 12 and 50 times without making progress. For example, mistral/devstral-small-2507 on the v11→v12 instance alternated between npm install and a sed command to modify package.json 25 times in succession. This behavior suggests these models failed to recognize that their strategy was not working and lacked the metacognitive capability to attempt alternative approaches.

The single exception was gemini/gemini-2.5-flash-lite on the v18→v19 instance, where no repetitive cycle was detected in the final 50 steps. Whether this interruption was premature remains an open question for future analysis.

5.5.2 Empty Patch Submissions

A second failure mode occurred when agents signaled task completion but produced an empty Git diff. An empty diff indicates that the agent's final repository state was identical to the baseline commit, meaning no files were modified during the entire migration attempt.

We identified 5 experiments (2.9% of total runs) with empty patch submissions, distributed as follows:

TABLE 5.3: Distribution of empty patch failures across agents

Agent	Failures
openai/gpt-5-nano-2025-08-07	2
mistral/mistral-small-2506	2
gemini/gemini-2.5-flash-lite	1

Analysis of the execution traces revealed a common pattern: all failing agents attempted to run `ng update` without first installing dependencies via `npm ci`. Since the Angular CLI was not available in the environment without dependency installation, the `ng update` command produced errors that the agents did not recognize as critical. The agents then prematurely signaled completion without having made any file modifications.

This failure mode suggests that these smaller models (gpt-5-nano, mistral-small, gemini-flash-lite) lack sufficient knowledge of the Angular toolchain's prerequisite steps and failed to interpret error messages that would have indicated the need to install dependencies first.

5.6 Addressing the Research Questions

We conclude by directly addressing the research questions posed in Chapter 4.

RQ1: Can BenchMAC meaningfully differentiate between migration systems?

Yes, with statistical verification. The benchmark successfully differentiated systems across a wide performance spectrum, with build success rates ranging from 0% (mistral/devstral-small-2507) to 100% (six agents including the baseline). Critically, the 95% Wilson confidence intervals for the bottom-tier performers (e.g., [0.0%, 29.9%] for 0/9 success) do not overlap with those of top-tier performers ([70.1%, 100.0%] for 9/9 success), confirming that these performance differences are statistically significant and not artifacts of sampling variance. This demonstrates BenchMAC's capacity to provide objective, reproducible discrimination between approaches.

RQ2: What is the baseline performance of modern LLMs on Angular migrations, and how do they compare to rule-based approaches?

Six LLM-based agents achieved perfect 100% build success rates, matching the deterministic performance of Angular Schematics. However, this equivalence comes at substantial economic and computational costs: the most efficient AI agent (`gemini-2.5-flash`, \$0.032/run) operates at effectively infinite cost compared to the \$0.00 baseline, while premium models reach \$0.499/run. The Schematics baseline also completed migrations in exactly 2 steps (its fixed command sequence) versus a median of 15 steps for successful AI agents, with some runs exceeding 50 steps. However, these findings must be interpreted within the constraints of the benchmark design and dataset characteristics, as discussed in Section 6.1.1.

5.7 Summary

This chapter presented a comprehensive evaluation of 19 automated migration systems on 9 Angular upgrade tasks. The key findings are:

- BenchMAC successfully differentiated agents with statistically significant performance gaps (0% to 100% success rates).
- Six LLM-based agents matched the rule-based baseline's perfect performance, but at orders of magnitude higher cost (\$0.03–\$0.50 vs. \$0.00) and computational overhead (15–50 steps vs. 2 steps).
- No clear correlation exists between model cost and performance, suggesting architecture and training matter more than price.
- An "efficient frontier" emerged: `gemini-2.5-flash` achieved 100% success at \$0.032/run, making it the most cost-effective AI solution.
- Reasoning capabilities and domain specialization showed mixed, statistically inconclusive effects.
- Instance difficulty varied significantly (10.5%–47.4% failure rates), though overlapping confidence intervals prevent definitive difficulty rankings.
- Two systematic failure modes emerged: infinite loops (14 experiments, primarily smaller Mistral models) and empty patches (5 experiments, smaller models lacking Angular toolchain knowledge).

These results show that LLM-based migration agents are effective, while rule-based methods remain strong performers. However, as discussed in Chapter 6, the strong performance of the rule-based baseline should be viewed in light of the specific limitations and setup of this evaluation.

Chapter 6

Discussion

This chapter interprets the experimental findings presented in Chapter 5, examining why certain patterns emerged and what they reveal about the current state of AI-assisted framework migration. We contextualize the surprising strength of the rule-based baseline, analyze the economic and strategic implications for practitioners, dissect the failure modes of AI agents, and situate BenchMAC's contributions within the broader landscape of software engineering benchmarks.

6.1 Interpreting the Core Findings

6.1.1 The Surprising Strength of the Rule-Based Baseline

The most striking finding from our evaluation is that the rule-based `angular-schematics` baseline achieved a perfect 100% success rate while incurring zero API costs and requiring exactly two commands per migration. This outcome demands careful interpretation, as it challenges initial expectations about the relative capabilities of AI agents and traditional automation tools.

Angular provides an official command-line tool called **Schematics**, which automates many repetitive upgrade steps. When developers run `ng update`, the tool executes a series of pre-defined transformation scripts, called *migration schematics*, that rewrite code and configuration files to comply with the new version's requirements. These transformations are deterministic and rule-based: they modify the code according to patterns explicitly defined by the Angular team, such as renaming APIs, updating imports, or adjusting configuration fields.

In the controlled context of BenchMAC v1.0, these schematics performed extremely well. Every migration succeeded because the benchmark repositories are clean, minimal, and designed to isolate version-specific changes. The 9 instances in our dataset come from a single repository at 9 points in time corresponding to 9 Angular versions. The codebase is relatively small (approximately 1,400-1,500 lines of TypeScript code), follows standard Angular CLI conventions, and contains minimal third-party dependencies.

These characteristics represent an *ideal case*: a project that closely follows Angular's recommended structure, with a relatively minimal and well-maintained set of dependencies and configuration. In this constrained environment, the deterministic pattern matching of schematics is generally sufficient to handle all required changes.

However, real-world Angular projects are substantially more heterogeneous and unpredictable. They typically include numerous third-party libraries, custom build configurations, non-standard project structures, dynamic code generation, or legacy patterns that the official schematics are not designed to handle. Because schematics operate through static pattern matching and Abstract Syntax Tree (AST) transformations, they cannot reason about complex project-specific logic, resolve conflicts between dependencies, or adapt to unconventional codebases.

The perfect performance of the baseline thus reveals a fundamental characteristic of our current dataset: the migration path for these instances is unusually "schematics-friendly." This is an important limitation to acknowledge, and we address its implications for benchmark validity in Chapter 7. One of the key objectives for future versions of BenchMAC is to introduce greater diversity in the dataset to capture scenarios where rule-based tools struggle and AI agents can demonstrate their flexibility advantage.

6.1.2 Why AI Agents Underperformed Expectations

Given that AI agents operate with access to a full terminal environment, they are, in principle, able to execute the exact same commands as the rule-based baseline, such as running `npm ci` followed by `ng update`. In addition, they can perform arbitrary file operations, search through documentation, run diagnostic commands, and iteratively debug errors. These capabilities go far beyond what is possible with a fixed two-command script. For this reason, it was reasonable to expect AI agents to at least match, and potentially even surpass, the performance of the schematics baseline.

The fact that several agents underperformed the rule-based approach, even though they had access to a broader set of capabilities than the baseline, suggests that simply providing the right tools is not enough. The agent must also have the correct strategy, domain knowledge, and reasoning ability in order to use those tools effectively.

We identify three primary factors that likely contributed to the underperformance of some AI agents relative to the baseline:

1. Minimal Task Prompting. The system prompt used for all agents was deliberately minimalistic: "Migrate the application from Angular version {source_version} to {target_version}." It included no strategic guidance, no hints about the Angular upgrade process, and no recommendations such as "start by installing dependencies" or "verify your work by running a build before submission." This was a deliberate design decision intended to isolate the intrinsic capabilities of the models from any influence of prompt engineering. As a result, agents received no domain-specific guidance.

2. Lack of Contextual Knowledge. None of the agents were provided with Angular-specific documentation, migration guides, or API changelogs. While some agents attempted to fetch documentation themselves (e.g., issuing `curl` commands to `https://angular.dev`), this information was not systematically provided. In contrast, human developers or production migration systems would typically consult official upgrade guides, which detail breaking changes and provide step-by-step instructions.

3. Limitations of the Terminal Interface. The `mini-swe-agent` framework provides agents with a shell environment, which is extremely flexible but not optimized for common coding operations. For example, editing files requires using `sed` or similar command-line tools rather than structured file editing commands. Other scaffolding frameworks, such as OpenHands [98], provide specialized tools like `str_replace`, `view`, `write_file`, and `insert`, which are more intuitive for language models to use reliably.

Yang et al. [3] demonstrated that agent-computer interfaces (ACIs) play a critical role in LLM agent performance. They found that basic shell access, while flexible, presents several challenges: commands like `cat` can flood the context window with irrelevant file content, navigation within files using standard Unix tools is verbose and unintuitive, and there is no straightforward way to identify symbol definitions or jump to specific code locations. These limitations make it much harder for agents to perform well and highlight the value of providing more LLM-friendly, higher-level tools.

The fact that six AI agents *did* achieve perfect scores, despite these constraints, demonstrates that current frontier models possess sufficient capability to overcome these challenges when their base reasoning and code understanding are strong enough. However, the failures of other agents (particularly smaller models) suggest that success on this task requires a certain threshold of capability that not all models currently meet.

6.1.3 The Heterogeneity of Real-World Projects

The strong performance of the rule-based baseline should not be interpreted as evidence that AI agents are unnecessary for Angular migrations. Rather, it highlights a specific limitation of the current benchmark: the dataset does not yet capture the full complexity and diversity of real-world migration scenarios.

The `gothinkster/angular-realworld-example-app` repository, while a legitimate open-source project, represents a best-case scenario for automated migration:

- **Standard structure:** The project follows the conventional Angular CLI layout with no custom build scripts or unconventional configurations.
- **Minimal dependencies:** The application uses a small set of well-maintained libraries, all of which have official Angular version compatibility.
- **Clean history:** Each version in the repository's history corresponds to a deliberate upgrade, with no half-finished migrations or conflicting changes.
- **No monorepo complexity:** The repository is a single application, not a monorepo with multiple packages, shared libraries, or complex dependency graphs.
- **No test infrastructure challenges:** The instances selected for BenchMAC v1.0 do not include unit or end-to-end test suites, avoiding the complexity of upgrading testing frameworks, test utilities, and browser automation tools.

In contrast, industrial Angular codebases frequently exhibit characteristics that complicate automated migration:

- Custom build pipelines and tooling. Examples include Webpack configurations or bespoke bundlers.
- Extensive third-party dependencies with version conflicts
- Monorepo architectures (Nx, Turborepo) with shared code and complex interdependencies
- Legacy code patterns that predate current Angular idioms
- Custom authentication, state management, or architectural patterns

- Integration with backend services, databases, or external APIs
- Comprehensive test suites requiring coordinated upgrades of testing libraries

For such projects, the deterministic pattern matching of schematics is often insufficient. The tool may successfully apply syntactic transformations but fail to resolve dependency conflicts, adapt custom configurations, or handle non-standard code patterns. In these scenarios, the flexibility and reasoning capability of AI agents become valuable: they can analyze error messages, search for solutions, experiment with different approaches, and adapt to project-specific requirements in ways that rule-based tools cannot.

BenchMAC v1.0 establishes a baseline by evaluating performance on standardized, schematics-friendly projects. This is a deliberate methodological choice that ensures reproducibility and provides a foundation for comparison. However, future versions of the benchmark must expand to include more heterogeneous repositories to test whether AI agents can demonstrate their flexibility advantage in scenarios where rule-based tools struggle.

6.2 Economic and Practical Implications

6.2.1 Cost-Performance Trade-offs

One of the most striking findings from our economic analysis is the absence of a clear correlation between model price and migration performance. This has important implications for organizations considering AI-assisted migration tools.

As shown in Figure 5.3, agents are distributed across three quadrants of the cost-performance space:

- **High cost, high performance:** Models like `claude-sonnet-4-5` ($0.499/run) and `gpt-5-codex` ($0.374/run) achieved perfect scores but at premium prices.
- **Low cost, high performance:** Models like `gemini-2.5-flash` ($0.032/run) and `grok-4-fast-non-reasoning` ($0.016/run) also achieved strong results (100% and 88.9% respectively) at dramatically lower costs.
- **Variable cost, poor performance:** Models like `devstral-medium` ($0.155/run) and `magistral-medium` ($0.136/run) had moderate costs but lower success rates (55.6% and 77.8% respectively).

The existence of the low-cost, high-performance quadrant is particularly significant. The `gemini-2.5-flash` model represents the "efficient frontier" for this benchmark: it achieves perfect performance at less than one-tenth the cost of premium alternatives. This demonstrates that for relatively straightforward migration tasks, expensive flagship models do not provide a proportional return on investment.

However, this finding must be interpreted with caution. The current benchmark exhibits a ceiling effect: all six top-performing agents achieved 100% success with overlapping 95% confidence intervals ([70.1%, 100.0%]), meaning we cannot statistically distinguish their true capabilities. This suggests the tasks are not sufficiently challenging to differentiate among frontier models.

There is a strong likelihood that the most expensive flagship models such as `gpt-5-codex` and `claude-sonnet-4-5` would demonstrate clear advantages over `gemini-2.5-flash` when faced with more challenging migration tasks. These advantages would likely only become evident in scenarios that involve higher complexity, such as projects with dependency conflicts, custom build systems, monorepo structures, or cases that require multiple steps of reasoning to address breaking changes.

6.2.2 Guidance for Practitioners

Based on our findings, we can offer preliminary guidance for organizations considering AI-assisted migration approaches, while acknowledging the limitations of generalizing from our current dataset.

When Rule-Based Tools Suffice

For projects with the following characteristics, the Angular Schematics baseline remains the optimal choice:

- Standard Angular CLI project structure
- Minimal third-party dependencies
- No custom build configurations
- Following Angular's recommended patterns and idioms
- Single-application repositories (not monorepos)

The schematics approach offers zero cost, deterministic behavior, and instant execution. Organizations should exhaust this option before considering more complex alternatives.

When to Consider AI Agents

AI agents potentially justify their cost and complexity when migrations involve:

- Non-standard project structures or custom tooling
- Complex dependency graphs with version conflicts
- Integration of multiple frameworks or libraries
- Migration of custom code patterns not covered by schematics
- Projects requiring contextual reasoning about architectural decisions

Important caveat: The current benchmark does not yet provide empirical validation of this hypothesis, as our dataset lacks these complexities. This represents a key direction for future work.

Model Selection Recommendations

For organizations adopting AI-assisted migration, our results suggest the following:

Models to avoid: Based on consistent failure patterns, we recommend against:

- `mistral/devstral-small-2507` (0% success rate)
- `mistral/mistral-small-2506` (22.2% success rate)

- `openai/gpt-5-nano` (22.2% success rate)

These models failed even on the relatively simple tasks in BenchMAC v1.0, indicating insufficient capability for real-world migrations.

Cost-effective options: For straightforward migrations similar to our benchmark tasks:

- `gemini/gemini-2.5-flash` offers the best cost-performance ratio at $0.032/run with 100% success
- `xai/grok-4-fast-non-reasoning` provides strong performance at $0.016/run with 88.9% success

Premium options: For complex migrations requiring maximum reliability:

- `anthropic/claude-sonnet-4-5` ($0.499/run)
- `openai/gpt-5-codex` ($0.374/run)
- `gemini/gemini-2.5-pro` ($0.155/run)

All three achieved perfect scores in our evaluation, though their advantage over cheaper models is not yet demonstrated on the current benchmark.

The Hybrid Approach

The most pragmatic strategy may be a hybrid approach:

1. **First pass:** Always attempt migration using `ng update` (zero cost, often sufficient)
2. **Verification:** Run build and test suites to check for failures
3. **AI fallback:** If the rule-based approach fails, deploy an AI agent to:
   - Analyze error messages from the failed build
   - Research and apply fixes for dependency conflicts
   - Adapt custom code patterns
   - Iterate on solutions until tests pass

This approach minimizes cost by using free tooling when possible while leveraging AI capabilities for the subset of challenging cases where rule-based tools fail.

6.3 Failure Mode Analysis

Beyond aggregate success rates, analyzing how and why agents failed provides valuable insights into the current limitations of AI-assisted migration systems.

6.3.1 Infinite Loops and the Meta-Cognition Problem

One of the most problematic failure patterns we observed was agents becoming trapped in infinite loops, repeatedly executing the same commands without recognizing that their strategy was not working. This affected 14 experiments (8.2% of total runs), with failures heavily concentrated among smaller Mistral models.

As shown in Table 5.2, `mistral/devstral-small-2507` accounted for 7 of these failures, `mistral/mistral-small-2506` for 4, and `mistral/devstral-medium-2507` for 2. The single failure from a non-Mistral model was `gemini/gemini-2.5-flash-lite`, suggesting this is primarily a problem for smaller models.

Anatomy of the Failure Pattern

When we analyzed the command execution traces, we found clear evidence of repetitive behavior. In 13 of 14 cases where agents exceeded the 100-step limit, cycle detection revealed that the agent was repeating sequences of 1-4 commands between 12 and 50 times. For example:

- `devstral-small` on v11→v12 alternated between `npm install` and a `sed` command to modify `package.json` 25 times consecutively
- `mistral-small` on v12→v13 repeated a 3-command sequence 16 times. The cycle combined `ls /usr/bin/node*` with two npm commands.

The dominant repetitive action was the `npm install` command (or variations like `npm ci`), often interleaved with minor file edits. This pattern suggests the agents recognized that dependency installation was required but failed to diagnose why it was failing or to pivot to alternative strategies.

Comparison to Prior Work

This failure mode is consistent with findings from other agent evaluations. Yang et al. [99] observed similar behavior in their evaluation of SWE-agent systems, reporting that more than 25% of `SWE-agent-LM-32B` trajectories contained repetitive sequences of at least length 10, compared to less than 4% for the more capable `Claude 3.7 Sonnet`.

Critically, they found that repetitive sequences strongly correlated with task failure: a sequence of length 10 corresponded to an 89% failure probability. Their attempted intervention (injecting warnings and forcing action resampling) successfully reduced the symptom (repetitive commands) but did not improve overall success rates. This led them to conclude that repetition is not the root cause of failure, but rather a *symptom* of the model encountering a task beyond its capability.

The Meta-Cognition Gap

The persistence of these loops reveals a fundamental limitation in current AI agents: the lack of metacognitive ability to recognize when a strategy is failing and to deliberately pivot to an alternative approach. A human developer, after attempting the same command three or four times without success, would naturally step back to:

- Examine error messages more carefully
- Search for documentation or similar issues
- Try a different approach entirely
- Seek help or escalate the problem

The agents that became stuck lacked this self-reflective capability. They continued issuing similar commands until the 100-step limit was reached.

The Role of Temperature Settings

A simple but important note: for all Mistral models, we used the temperature settings suggested by the provider (`temperature=0.7` for Magistral models, `temperature=0.3` for Mistral base models, `temperature=0.15` for Devstral models). If these default settings caused the infinite loop problem, it shows that the models can be fragile: they may need careful tuning of these settings to avoid getting stuck, even on easy tasks.

6.3.2 Empty Patches and the Multi-Step Reasoning Challenge

The empty patch failures, while less common than infinite loops (2.9% vs 8.2% of runs), reveal a different category of limitation in smaller models: difficulty with multi-step reasoning and error recovery in complex procedural tasks.

Diverse Failure Patterns

Unlike the infinite loop pattern, which showed remarkable consistency (agents repeatedly executing the same 1-4 commands), the empty patch failures exhibited diverse root causes across the five affected experiments. No two failures followed the exact same pathway. This heterogeneity itself is informative: it suggests these models struggle not with a single missing capability (such as lacking knowledge of a specific command), but rather with the broader challenge of maintaining coherent plans across multiple steps when encountering unexpected obstacles.

A capable agent performing a migration must:

1. Assess the current environment state
2. Execute a sequence of dependent commands in the correct order
3. Interpret error messages and tool output
4. Adapt its strategy when blocked
5. Verify that its actions had the intended effect
6. Recognize when the task is complete versus incomplete

The empty patch failures indicate breakdown at various points in this pipeline. Some agents failed at step 2 (procedural ordering), others at step 4 (adaptation to environment constraints), and others at step 6 (premature completion signals despite errors).

Contrast with Capable Models

The concentration of empty patch failures exclusively in smaller models (`gpt-5-nano`, `mistral-small`, `gemini-2.5-flash-lite`) is particularly revealing when contrasted with the behavior of larger models that achieved perfect scores.

Successful agents consistently demonstrated:

- Correct initialization sequences (`npm ci` before `ng` commands)
- Recognition of and adaptation to error conditions
- Verification of work through build attempts before submission

- Accurate assessment of task completion state

These capabilities were absent or unreliable in the models that produced empty patches. This suggests a capability threshold: below a certain model size or training quality, the multi-step reasoning required for migration tasks becomes unreliable, leading to failures that manifest in unpredictable ways depending on the specific obstacles encountered.

The False Completion Problem

A particularly concerning aspect of the empty patch failures is that agents signaled successful completion despite producing no changes. This represents a failure of self-assessment: the models could not accurately evaluate whether they had accomplished the task.

This is different from the infinite loop failures, where agents kept trying to make progress, even if they were not successful. In the empty patch cases, the agents mistakenly believed they had finished the task, even though they had actually failed very early in the process.

For production deployment, this failure mode is especially problematic because it provides no signal that intervention is needed. An infinite loop will eventually trigger timeout alerts, but an agent that confidently submits an empty solution appears successful until human review reveals the lack of actual work.

Implications for Model Selection

The empty patch failures provide a clear negative signal: models that fail on BenchMAC's relatively simple tasks by producing empty solutions should not be deployed for real-world migrations. The failure mode indicates fundamental deficits in:

- Multi-step procedural reasoning
- Error interpretation and recovery
- Self-assessment of task completion

Organizations should prioritize models that demonstrate reliable performance on basic migration tasks before considering them for more complex scenarios. The fact that multiple frontier models achieved perfect scores with zero empty patch failures demonstrates that current state-of-the-art capabilities are sufficient to avoid this failure mode entirely, making it a useful filter for eliminating unsuitable systems.

6.4 The Role of Scaffolding and Training

Our experimental design deliberately employed minimal scaffolding, using only the `mini-swe-agent` framework with basic shell access and the simplest possible task prompt, to establish a baseline that isolates core model capabilities from the effects of sophisticated prompt engineering and tool optimization.

This design choice has important implications for interpreting our results and understanding the gap between current performance and the potential of AI-assisted migration systems.

6.4.1 Why Minimal Scaffolding?

The decision to use minimal scaffolding serves three methodological goals:

1. **Fair comparison across models:** By providing identical scaffolding to all systems, we ensure that performance differences reflect model capabilities rather than differences in prompt optimization or tool design.
2. **Reproducible baseline for future research:** Establishing a simple, well-documented baseline allows future work to systematically evaluate the impact of scaffolding improvements (better prompts, advanced tools, retrieval-augmented generation) by comparing against our results.
3. **Isolation of core model capabilities:** We strip away sophisticated prompts, specialized tools, and domain-specific context to measure the base reasoning and code understanding abilities of the models themselves. This reveals which models possess sufficient foundational capability before any optimization.

6.4.2 What This Means for Our Findings

The results presented in Chapter 5 represent a *lower bound* on LLM performance for Angular migrations. Several factors suggest that performance would improve with production-quality scaffolding:

1. Optimized prompting. Production systems would include strategic guidance such as:

- "Begin by installing dependencies using `npm ci`"
- "After making changes, verify your work by running `ng build`"
- "If you encounter errors, read the full error message and search for solutions"
- "Consult the Angular update guide at { angular_update_guide_url }"

Such prompts would likely reduce or eliminate failures like the empty patch submissions, where agents simply didn't know to install dependencies first.

2. Better tool abstractions. The `mini-swe-agent` framework provides only a Unix shell, requiring agents to use commands like `sed` for file editing. More sophisticated scaffolds provide:

- Structured file editing commands (`str_replace`, `insert`, `view`) that are less error-prone than shell text processing
- Search and navigation tools optimized for code repositories
- Integration with language servers for semantic code understanding
- Direct access to test runners and linters with structured output

Yang et al. [3] demonstrated that these abstractions significantly improve agent performance by reducing the cognitive load of low-level command construction and providing more reliable feedback.

3. Contextual knowledge retrieval. Production systems often employ Retrieval-Augmented Generation (RAG) to provide relevant documentation, migration guides, and API changelogs. Recent studies show its potential benefits and limitations in code migration contexts. Liang et al. [87] demonstrated that RAG can substantially improve post-cutoff code generation, achieving up to a 13.5% performance gain by retrieving version-specific documentation and examples for evolving APIs.

6.4.3 What This Doesn't Diminish

Importantly, the minimal scaffolding design does not diminish the validity of our key findings:

- **The benchmark successfully differentiates systems:** The 0-100% performance range demonstrates clear separation between capable and incapable models, even under minimal scaffolding.
- **Systematic failure modes remain diagnostic:** The infinite loops and empty patch failures reveal fundamental capability gaps (metacognition, domain knowledge) that better prompting might mitigate but cannot eliminate.
- **Economic analysis remains valid:** Even with optimized scaffolding, the cost differences between models would persist, and the absence of a cost-performance correlation would likely remain.
- **The baseline comparison is fair:** All agents received identical scaffolding, making relative performance comparisons valid even if absolute performance is conservative.

6.4.4 Scaffolding as a Research Variable

Our approach establishes scaffolding as a variable that future research can systematically manipulate. Researchers can now evaluate:

- The impact of prompt optimization by comparing against our baseline
- The value of specialized tools by testing agents with different Agent-Computer Interfaces (ACIs)
- The benefit of RAG by providing varying amounts of contextual documentation
- The effect of model-specific fine-tuning for agentic workflows

This progression from minimal to sophisticated scaffolding mirrors the natural evolution of production systems and provides a structured pathway for advancing the state of the art.

6.5 What BenchMAC Successfully Demonstrates

While we have discussed the limitations of the current dataset and the conservative nature of our baseline methodology, it is important to acknowledge what BenchMAC v1.0 successfully achieves as a benchmark.

6.5.1 Effective Differentiation

The benchmark successfully separates AI systems across a wide performance spectrum. The 0% to 100% range in build success rates, combined with non-overlapping confidence intervals between top and bottom performers, demonstrates clear discriminative power. Crucially, this differentiation occurred even under minimal scaffolding and with relatively simple tasks.

Models like `mistral/devstral-small-2507` failed on every instance, while six agents achieved perfect scores. This separation is statistically significant and actionable: organizations can confidently avoid the bottom-tier models and focus on systems that demonstrate basic competence.

6.5.2 Identification of Systematic Failure Modes

BenchMAC revealed two distinct failure patterns, infinite loops and empty patch submissions, that serve as diagnostic signatures for different types of capability gaps:

- **Infinite loops** indicate a lack of metacognitive ability to recognize failing strategies, primarily affecting smaller models
- **Empty patches** indicate insufficient domain-specific toolchain knowledge, also concentrated in smaller models

These patterns provide actionable insights for model developers and practitioners. They suggest specific areas where training or prompting interventions could improve performance, and they offer clear signals for when a model is unsuitable for real-world deployment.

6.5.3 Economic Transparency

By systematically tracking API costs alongside performance metrics, BenchMAC provides transparent data on the economic trade-offs of different AI systems. The finding that model price does not correlate with performance and that highly capable models exist at multiple price points empowers organizations to make cost-conscious decisions.

The existence of an "efficient frontier" (e.g., `gemini-2.5-flash` at $0.032/run) challenges the assumption that more expensive models are always better, at least for tasks within the current benchmark's scope.

6.5.4 Reproducible Methodology

The containerized evaluation harness, unified diff interface, and open-source implementation ensure that all results are reproducible. Future researchers can:

- Re-run experiments with identical environments
- Add new instances to the benchmark
- Evaluate new models or scaffolding approaches
- Verify or challenge our findings

This reproducibility is a core contribution that distinguishes BenchMAC from ad-hoc evaluations and informal benchmarks.

6.6 Contributions Beyond the Leaderboard

While the performance leaderboard provides immediate practical value, BenchMAC's broader contributions to the research landscape extend beyond raw success rates.

6.6.1 A Methodology for Repository-Level Migration Evaluation

The two-stage architecture, separating patch generation from patch evaluation, establishes a reusable pattern for benchmarking AI-assisted code transformation tasks. Key methodological innovations include:

- **The unified diff as a standard interface:** By requiring all systems to produce a single artifact (a patch file), we enable evaluation of heterogeneous systems without constraining their internal workflows.
- **Containerized reproducibility:** The use of Docker with pinned dependencies, snapshot package repositories, and immutable base images ensures long-term reproducibility even as the broader ecosystem evolves.
- **Execution-based metrics over surface similarity:** Rather than comparing generated code to reference solutions, we verify functional correctness through build success, acknowledging that many valid solutions exist for any migration task.

These patterns are transferable to other code transformation domains: API migrations, framework ports (e.g., React to Angular), language translations, or modernization tasks.

6.6.2 Framework-Specific Evaluation

BenchMAC addresses a significant gap in the evaluation landscape by focusing on frontend framework migration, a domain that has received far less attention than backend code generation or bug fixing. Most existing benchmarks (HumanEval, SWE-Bench, MBPP)

By providing a rigorous evaluation framework for Angular, we enable research on challenges specific to modern web development.

6.6.3 A Baseline for Agentic System Research

The minimal scaffolding approach, while conservative, provides a clean baseline for studying the impact of agent design choices. Future research can use BenchMAC to systematically evaluate:

- **Prompting strategies:** How much does performance improve with optimized instructions, few-shot examples, or chain-of-thought reasoning?
- **Tool design:** What is the marginal benefit of structured file editing commands versus raw shell access?
- **Retrieval augmentation:** Does providing Angular documentation improve success rates, and if so, by how much?
- **Model fine-tuning:** Can domain-specific training on Angular migration trajectories improve performance beyond general-purpose code training?
- **Multi-agent architectures:** Do systems with specialized sub-agents (e.g., one for dependency resolution, one for code refactoring) outperform monolithic agents?

Each of these research directions can now be pursued with BenchMAC as a standardized evaluation target.

6.7 Implications for Future Research

The findings from this thesis suggest several promising directions for advancing AI-assisted software migration.

6.7.1 The Dataset Diversity Challenge

The most pressing research need is the expansion of benchmark datasets to capture greater heterogeneity in project characteristics. Future versions of BenchMAC should include:

- **Monorepo architectures:** Projects using Nx or Turborepo with multiple applications and shared libraries
- **Non-standard configurations:** Custom build scripts, modified Angular CLI setups, or projects that have diverged from official conventions
- **Dependency conflicts:** Instances where migration requires resolving incompatibilities between third-party libraries
- **Test suite evolution:** Projects with comprehensive unit and E2E tests that must be updated alongside application code
- **Multi-version jumps:** Migrations spanning multiple major versions (e.g., v11→v15) that require sequential or compound transformations

The hypothesis that AI agents will demonstrate advantages over rule-based tools on more complex projects requires empirical validation. Without such validation, we cannot definitively claim that the flexibility of AI agents justifies their cost.

6.7.2 Sub-Task Decomposition

Current end-to-end migration evaluation provides a single binary signal (build success), which limits diagnostic insight. Future work should explore:

- **Micro-benchmarks:** Isolated tasks such as "update import statements," "modify configuration files," or "adapt deprecated API calls"
- **Staged evaluation:** Measuring success at intermediate milestones (dependency installation, compilation, test execution) rather than only the final build
- **Partial credit scoring:** Metrics that capture progress even when full migration fails, such as the fraction of files correctly modified or the percentage of breaking changes addressed

This finer-grained evaluation would better reveal where different agents excel or struggle, providing more actionable feedback for system improvement.

6.7.3 Beyond Build Success: Quality Metrics

Build success is a necessary but insufficient criterion for evaluating migration quality. Future benchmarks should incorporate:

- **Test suite pass rates:** What fraction of existing tests continue to pass after migration?

- **Code idiomaticity:** Does the migrated code follow current Angular best practices, or does it retain outdated patterns? This could be evaluated via LLM-as-judge approaches or static analysis tools.
- **Performance characteristics:** Are there regressions in bundle size, runtime performance, or build time?
- **Manual inspection protocols:** Structured human evaluation of code quality, maintainability, and adherence to project conventions

6.7.4 Scaffolding and Tool Optimization

Given that our results represent a lower bound due to minimal scaffolding, systematic research on agent-computer interfaces for migration tasks is warranted:

- Comparative evaluation of different tool abstractions (shell vs. structured editors)
- Optimal retrieval strategies for migration documentation
- Prompt templates and few-shot examples tailored to framework migration
- Integration of static analysis tools (type checkers, linters) into the agent feedback loop

6.7.5 The Reasoning Question

Our results on reasoning models were inconclusive. The XAI Grok comparison suggested non-reasoning models might perform better on straightforward tasks, but confidence intervals overlapped substantially. Systematic research is needed to understand:

- For what types of migration tasks (if any) does explicit reasoning provide measurable benefit?
- What is the cost-benefit trade-off of reasoning overhead versus improved solution quality?
- Can reasoning traces be used to explain agent decisions, improving trust and debuggability?

6.7.6 Cross-Framework Generalization

An ambitious long-term goal is to understand whether migration capabilities generalize across frameworks. Questions include:

- Does an agent trained or optimized for Angular migrations transfer to React or Vue migrations?
- Are there universal patterns in framework migration (e.g., dependency updates, configuration changes) that agents can learn?
- Can multi-framework benchmarks drive development of generalist migration assistants?

6.8 Summary

This chapter has interpreted the experimental findings of BenchMAC v1.0, revealing that the surprising strength of the rule-based baseline reflects the "schematics-friendly" nature of the current dataset rather than a fundamental superiority of rule-based approaches. AI agents underperformed expectations primarily due to minimal scaffolding and prompting, factors that were deliberately chosen to establish a clean baseline rather than to maximize performance.

Economic analysis revealed no correlation between model cost and success, with `gemini-2.5-flash` emerging as the most cost-effective solution at $0.032 per run. However, the benchmark's current ceiling effect prevents discrimination among top-tier models, limiting our ability to justify premium model selection.

Failure mode analysis identified systematic patterns, infinite loops indicating meta-cognition deficits and empty patches revealing toolchain knowledge gaps, that serve as diagnostic signatures for model capability. These patterns, concentrated in smaller models, provide clear signals for practitioners about which systems to avoid.

Despite its limitations, BenchMAC successfully differentiates systems across a 0-100% performance range, reveals actionable failure modes, provides transparent economic data, and establishes reproducible methodology transferable to other code transformation domains. The minimal scaffolding approach, while yielding conservative performance estimates, provides a baseline for systematic research on prompting, tool design, and agent architectures.

Future work must expand dataset diversity to validate the hypothesis that AI agents outperform rule-based tools on complex, heterogeneous projects. Sub-task decomposition, quality metrics beyond build success, and cross-framework generalization represent promising research directions.

Chapter 7

Limitations & Threats to Validity

This chapter acknowledges the limitations of BenchMAC v1.0 and discusses threats to the validity of our findings. We organize these into three categories: limitations inherent to the benchmark design, limitations in our experimental methodology, and broader threats to validity. We conclude by outlining a roadmap for addressing these limitations in future work.

7.1 Dataset and Benchmark Design Limitations

7.1.1 Dataset Scale and Diversity

Single Repository Source. The most significant limitation of the current benchmark is that all 9 instances are derived from a single repository (`gothinkster/angular-realworld-example-app`). While this repository represents a legitimate open-source project, it provides limited diversity in project characteristics, coding styles, and architectural patterns. A more robust benchmark would sample from multiple repositories across different domains (e-commerce, content management, data visualization, etc.) to better capture the heterogeneity of real-world Angular applications.

Small Codebase Size. The repository contains approximately 1,400-1,500 lines of TypeScript code and 600-690 lines of HTML templates. This is substantially smaller than typical enterprise Angular applications, which often span tens of thousands of lines across dozens of components. The small size may make migrations artificially easier, as agents have less code to analyze and fewer potential interaction points where changes could introduce errors.

Missing Test Infrastructure. None of the benchmark instances include unit tests or end-to-end (E2E) tests. This creates a fundamental limitation: we can only verify that migrated code *compiles*, not that it *functions correctly*. Build success does not guarantee functional correctness. Runtime errors, logic bugs, and behavioral regressions can go undetected. Without tests, the benchmark cannot distinguish between a minimal working migration and a high-quality one that preserves all application behavior.

Future versions of BenchMAC should include instances with comprehensive test suites, though we acknowledge this introduces significant environmental complexity. Setting up reliable testing environments with headless browsers (Chromium) and test frameworks across multiple Angular versions is challenging, particularly for older projects where testing tooling has evolved substantially.

Minimal Third-Party Dependencies. Most instances depend only on `@rx-angular/cdk` and `marked` (a simple Markdown parser). This provides little exposure to one of the most challenging aspects of real-world migrations: resolving version conflicts and compatibility issues in the broader ecosystem. A typical enterprise Angular application might depend on UI component libraries (`@angular/material`), state management (`@ngrx/store`), authentication libraries (`@auth0/angular-jwt`), data visualization tools (`ngx-charts`), and internationalization frameworks (`@ngx-translate/core`). Each of these introduces potential breaking changes during version upgrades that our current dataset does not capture.

No Monorepo Complexity. All instances are single-application repositories following the standard Angular CLI layout. Modern enterprise development increasingly uses monorepo architectures (managed by Nx, Turborepo, or custom tooling) containing multiple applications, shared libraries, and complex inter-project dependencies. The current benchmark pipeline assumes a classic single-app structure and cannot evaluate migrations in workspace environments where a single version change must be coordinated across multiple projects with different upgrade paths and compatibility constraints.

7.1.2 Base Commit Selection Strategy

Our approach to defining instance base commits introduces a potential systematic bias. For each instance, we identify commits in the Git history where the `@angular/core` version was bumped to the next major version, then select the immediately preceding commit as the base state. This means the migration task includes both the version bump and all necessary code changes.

However, repository maintainers may have performed preparatory work in earlier commits that is legitimately part of a real migration workflow. Examples include:

- Updating auxiliary dependencies to compatible versions
- Refactoring code to remove deprecated patterns before the breaking change
- Isolating risky changes in separate commits for easier rollback

By selecting only the immediate preceding commit, we potentially encode a minimal subset of the full migration work, making tasks appear simpler than a complete, production-grade migration would be.

Additionally, the "green baseline" principle inherently biases the dataset toward well-maintained projects where the source version builds cleanly. Real-world migrations often begin from broken or partially-working states (failed CI builds, known bugs, incomplete features). This selection bias may overestimate the effectiveness of automated tools, as they are not tested on the messier starting conditions that developers frequently encounter.

7.1.3 Evaluation Scope and Metrics

Build Success as Primary Metric. Our highest-tier success criterion is a clean production build (`npx ng build --configuration production`). While this is a strong signal of syntactic and structural correctness, it provides no information about:

- **Functional correctness**: Does the application behave as intended at runtime?
- **Test pass rates**: What fraction of existing tests continue to pass?
- **Code quality**: Is the migrated code idiomatic, maintainable, and following current best practices?
- **Linting compliance**: Does the code respect project style guidelines and avoid anti-patterns?

- **Performance characteristics**: Are there regressions in bundle size, runtime performance, or build time?
- **Security implications**: Does the migration introduce vulnerabilities?

The pass/fail nature of build success is also coarse-grained. An agent that correctly migrates 95% of files but introduces a single critical error receives the same score (failure) as one that makes no meaningful changes at all. Partial credit scoring would provide finer-grained diagnostic information.

Even test execution, the logical next evaluation step, is not a complete solution. [100] have shown that high test coverage does not guarantee detection of all regressions, and developers often distrust the completeness of their own test suites. Tests frequently suffer from low coverage of dependency interactions, and update-induced behavioral changes can go unnoticed if tests focus primarily on application logic rather than framework integration points.

7.1.4 Reproducibility and Infrastructure

Docker Strategy. While we have taken significant steps to ensure reproducibility (pinning base image SHA-256 digests, using snapshot Debian repositories with fixed dates, locking Node.js versions), several risks remain:

First, we directly clone repositories from GitHub during Docker image builds. If a repository is deleted, moved, or has its history rewritten, the Docker images cannot be rebuilt from the Dockerfiles alone. A more robust solution would fork repositories to a source under our control or store repository tarballs in stable archival storage.

Second, despite our best efforts, there may be reproducibility issues we have not identified.

The most foolproof approach would be to build all Docker images once, publish them to a registry (Docker Hub, GitHub Container Registry), and never rebuild them. Future evaluations would pull these immutable images rather than reconstructing them from Dockerfiles. This is feasible given our small dataset (approximately 300MB per instance, totaling under 3GB for all images) but would require ongoing infrastructure maintenance to ensure the registry remains accessible.

Scalability of Manual Curation. Each instance required manual creation of a custom Dockerfile, verification of the green baseline, and generation of a silver patch for harness validation. This process does not scale to hundreds or thousands of instances. Expanding the benchmark will require developing semi-automated tooling for:

- Automatic detection of Angular version bumps in repository histories
- Template generation for Dockerfiles based on detected Node.js requirements
- Automated baseline verification and health checks
- Batch processing and validation of candidate instances

7.1.5 Contamination and Generalization

Data Contamination Risk. The benchmark dataset is sourced from public GitHub repositories, making it theoretically vulnerable to contamination in LLM training data. The `gothinkster/angular-realworld-example-app` repository has a complete public Git history documenting how maintainers performed each version migration.

This history could have been included in the training corpora of some evaluated models, potentially inflating their performance.

However, we argue that contamination is not the most critical threat. The primary issue is task simplicity rather than data leakage. Even if models have memorized specific solutions from this repository, that knowledge must still be activated through appropriate reasoning and tool use. The failure of several models to achieve perfect scores despite potential training exposure suggests that memorization alone is insufficient.

Framework Generalization. While BenchMAC's core harness design (containerized environments, patch-based evaluation, execution-based metrics) is framework-agnostic, adapting it to other ecosystems requires non-trivial work:

- Command parsing and version detection logic is Angular-specific
- Each framework ecosystem has different version check mechanisms (`npm ls` for Node.js, `pip show` for Python, `gem list` for Ruby)
- Manual data curation and Dockerfile creation must be repeated for each target framework
- Framework-specific knowledge is needed to define appropriate evaluation command sequences

Extending to React, Vue, or other frontend frameworks would follow the same methodology but would require substantial engineering effort and domain expertise for each new target.

7.2 Experimental Design Limitations

7.2.1 Limited Sampling Strategy

Single Run Per Configuration. We executed each model once on each instance, providing no information about variance across multiple attempts. For models with non-zero temperature settings (Mistral family), different runs could potentially yield different outcomes. However, this limitation becomes less critical as the number of benchmark instances increases. With 50+ instances instead of 9, aggregate success rates become more statistically reliable even with single-run evaluations, as the law of large numbers provides effective averaging across tasks rather than across trials.

Determinism Challenges. To mitigate run-to-run variance, we used the most deterministic settings available: `temperature=0` where possible and random seeds when supported by the API. However, perfect reproducibility with LLMs remains a notoriously hard problem [101]. Providers that offer seed parameters explicitly warn that determinism is a best-effort service rather than a guarantee [102, 103]. Non-zero temperatures were used only when recommended by the provider (Mistral models), accepting some non-determinism as inherent to those systems.

7.2.2 System Diversity Limitations

Limited Open-Source Model Coverage. We included only one family of open-source models (Mistral). This creates an unfair comparison: a single open-source provider competing against multiple commercial providers (OpenAI, Anthropic, Google, XAI). A more balanced evaluation would include additional open-source alternatives:

- Qwen models (Alibaba Cloud)
- Llama models (Meta)
- Command R models (Cohere)
- DeepSeek model

The dominance of commercial models in our evaluation may overstate the necessity of paid services and underrepresent the capabilities of the open-source ecosystem.

Single Scaffolding Framework. All AI agents used the minimal `mini-swe-agent` framework. While excellent for controlled comparison and LLM capability isolation, this is far from representative of production-grade agentic systems. Real practitioners would use more sophisticated tools:

- **CLI-based agents**: Anthropic Claude Code, OpenAI Codex CLI, Google Gemini CLI, Qwen Code CLI, SWE-Agent (full)
- **IDE-integrated agents**: Cursor, Windsurf, VSCode with GitHub Copilot

Including some of these scaffolds (many are open-source) would provide insight into how LLMs perform with appropriate tooling, specialized prompts, and optimized interfaces. However, automation is more complex: these systems often require installation alongside the target codebase, making artifact collection (thought processes, command logs) and cost/step limit enforcement more challenging.

IDE Evaluation Gap. Many modern integrated development environments market AI-powered coding as a primary feature (Cursor, Kiro, Windsurf, VSCode with Copilot). While evaluating these systems is valuable, they are designed for interactive use and cannot be easily automated within our methodology. We can rely on their CLI equivalents (e.g., Cursor-CLI as a proxy for the Cursor IDE) as approximations.

No Human Baseline. We did not collect data on human developer performance, cost (hourly wages), or time requirements for the same migration tasks. Such a baseline would provide crucial context: how much faster or cheaper are AI agents compared to manual migration? At what quality trade-offs? Including even a small-scale human evaluation (e.g., 2-3 developers performing a subset of migrations) would ground the AI results in practical terms and help organizations understand the realistic value proposition of automation.

7.2.3 Analysis Depth

Limited Trajectory Analysis. We performed detailed analysis only on experiments that exhibited clear failure signatures (infinite loops, empty patches). A comprehensive evaluation would examine *all* failures (`build_success = False`) to construct a complete taxonomy of breakdown modes. This would require:

- Manual annotation of failure causes (requires Angular domain expertise)
- Categorization of error types (dependency conflicts, syntax errors, configuration mistakes, incomplete refactoring)
- Analysis of agent strategies (which commands were attempted, in what order, with what feedback interpretation)

Such analysis could answer important questions:

- How many agents verified their work by running `ng build` before submission?
- Did any agents attempt to search for or execute tests (even though none exist)?
- How many agents tried to fetch Angular documentation or migration guides?
- Did failures result from external constraints (network instability, npm registry timeouts) versus reasoning errors?
- What sub-problems are hardest for AI agents (dependency resolution, template migration, configuration updates)?

Identifying challenging sub-problems could inform the creation of fine-grained benchmark tasks focused on specific migration aspects rather than end-to-end workflows, providing more diagnostic signal than binary pass/fail evaluation.

7.3 Threats to Validity

7.3.1 Internal Validity

Confounding Variables. Multiple factors could influence agent performance beyond core LLM capability: the minimal scaffolding, lack of documentation access, simplistic prompts, and limited tool abstractions. We cannot definitively isolate which factors are responsible for observed failure patterns. For example, did smaller models fail due to insufficient base capabilities, or would better prompts and tools enable their success?

Evaluation Harness Reliability. While we validated the harness using silver patches, subtle bugs in the evaluation pipeline could systematically advantage or disadvantage certain agents. The harness is complex software with many conditional branches and edge case handling. Independent reproduction of our results by other researchers would increase confidence in the methodology.

7.3.2 External Validity

Generalization to Real Projects. The strong performance of the rule-based baseline on our benchmark tasks does not necessarily predict its performance on real-world, heterogeneous projects. The "schematics-friendly" nature of the current dataset limits our ability to make broad claims about when AI agents provide value over traditional automation.

Temporal Validity. LLM capabilities are advancing rapidly. Our evaluation captures a snapshot of model performance in late 2024 / early 2025. Findings may not generalize to future model generations.

Chapter 8

Conclusion & Future Work

8.1 Summary and Contributions

Software companies must regularly upgrade their frameworks to stay secure and maintainable. For Angular applications, these upgrades are often manual, slow, and error-prone. AI coding assistants promise to automate this work, but we lack reliable ways to measure how well they perform on real projects. Most existing benchmarks focus on backend languages like Python or Java, leaving frontend frameworks like Angular largely unexamined.

This thesis introduced BenchMAC, a benchmark specifically designed to evaluate automated Angular framework upgrades. The benchmark provides real codebases, standardized testing environments, and clear success metrics. Each system under test attempts to migrate an Angular project from one major version to the next, producing a patch file that captures all necessary code changes. We then apply these patches in isolated Docker containers and measure whether the upgraded project successfully installs dependencies and builds for production.

The main contributions of this work are:

- **Peer-reviewed analysis of migration benchmarks.** A survey of repository-level evaluation datasets, published at CORIA-TALN 2025 [14], quantifies the gaps in scale, dependency handling, and metric design that motivated BenchMAC's requirements.
- **A reproducible evaluation framework.** BenchMAC establishes a two-stage methodology that separates patch generation from evaluation. Systems produce a unified diff file, which is then evaluated in containerized environments with pinned dependencies and deterministic build processes. This design enables fair comparison across heterogeneous systems while ensuring long-term reproducibility.
- **Framework-specific evaluation.** This is the first rigorous benchmark for frontend framework migration. By focusing on Angular, we address challenges specific to modern web development: template syntax, component architecture, build system configuration, and the npm dependency ecosystem.
- **Empirical evaluation of 19 AI systems.** We evaluated multiple models from OpenAI, Anthropic, Google, XAI, and Mistral, alongside a rule-based baseline using Angular's built-in migration tool. This provides the first comprehensive comparison of AI approaches to framework migration.
- **Open-source release.** The complete benchmark, including the dataset, evaluation harness, and experimental results, is publicly available. This enables future research and allows practitioners to evaluate new systems as they emerge.

8.2 Key Findings

We evaluated 19 different AI systems and one rule-based tool across nine migration tasks. Six AI systems achieved perfect scores, successfully building all nine projects. These included models from Anthropic (Claude Sonnet 4 and 4.5), Google (Gemini 2.5-Pro and 2.5-Flash), and OpenAI (GPT-5-Codex). Notably, the rule-based Angular Schematics tool also achieved a perfect score while requiring exactly two commands and incurring zero API cost.

The successful AI agents required 15–50 LLM calls on average and cost between $0.03 and $0.50 per migration. We found no correlation between an AI system's price and its performance. Some expensive models performed well, but so did cheaper alternatives. Gemini 2.5-Flash emerged as the most cost-effective solution, achieving 100% success at only $0.032 per migration.

We identified two common failure patterns. Some agents got stuck in infinite loops, repeatedly executing the same commands without recognizing that their strategy was not working. This affected primarily smaller models and revealed a lack of metacognitive ability. Other agents submitted empty solutions after failing to properly set up the development environment, indicating insufficient knowledge of the Angular toolchain.

Addressing the Research Questions

RQ1: Can BenchMAC meaningfully differentiate between migration systems? Yes. The benchmark successfully separated systems across a 0% to 100% performance range. The statistical confidence intervals for bottom-tier performers do not overlap with those of top-tier performers, confirming that these differences are significant. This demonstrates that BenchMAC provides objective, reproducible discrimination between approaches.

RQ2: What is the baseline performance of modern LLMs on Angular migrations, and how do they compare to rule-based approaches? Six LLM-based agents matched the rule-based baseline's perfect performance. However, this came at substantial cost: the most efficient AI agent operates at $0.032 per run compared to $0.00 for the baseline, while premium models reach $0.499 per run. The baseline also completed migrations in two steps versus a median of 15 steps for AI agents. These findings must be interpreted carefully, as the current dataset is relatively simple and schematics-friendly.

8.3 BenchMAC as an Exploratory Tool

BenchMAC provides more than just a leaderboard. The benchmark serves as an exploratory tool for understanding how AI systems approach complex software engineering tasks and where they struggle.

By analyzing agent failures on complete migration tasks, we can identify specific difficulties and create a taxonomy of common problems. For example, our analysis revealed that some agents struggle with multi-step procedural reasoning (empty patches), while others lack the metacognitive ability to recognize failing strategies (infinite loops). These insights point to specific capability gaps that could be addressed through better training, prompting, or tool design.

This exploratory approach opens a path toward more focused evaluation. Once we understand which sub-problems are most challenging, we can develop targeted micro-benchmarks. Instead of only evaluating complete $N+1$ migrations, we could create focused tasks like "resolve peer dependency conflicts," "update template syntax," or "migrate deprecated API calls." These micro-benchmarks would provide finer-grained diagnostic information and drive more focused improvements in AI systems.

8.4 Implications

For practitioners. Organizations considering AI-assisted migration tools now have comparative data to inform their decisions. The findings suggest a pragmatic hybrid approach: start with the free, deterministic `ng update` command, verify the results with build and test suites, and deploy an AI agent only when the rule-based approach fails. For teams that do adopt AI tools, Gemini 2.5-Flash offers strong performance at low cost, while premium models like Claude Sonnet 4.5 may be justified for more complex scenarios.

The results also highlight the importance of testing beyond build success. A clean build is necessary but not sufficient for verifying migration quality. Organizations should maintain comprehensive test suites and consider additional quality checks like performance metrics, bundle size, and code idiomaticity.

For researchers. The methodology established by BenchMAC is transferable to other code transformation domains. The two-stage architecture, containerized evaluation, and execution-based metrics could be adapted for React or Vue migrations, API modernization, or cross-framework ports. The unified diff interface provides a clean abstraction that separates system capabilities from evaluation concerns.

The benchmark also provides a baseline for studying agent design choices. Researchers can now systematically evaluate the impact of different prompting strategies, tool abstractions, retrieval methods, or fine-tuning approaches by comparing against our results with minimal scaffolding.

For the field. This work demonstrates that rigorous, reproducible evaluation of AI-driven code migration is both feasible and necessary. The strong performance of the rule-based baseline, while partly reflecting dataset characteristics, reminds us that traditional automation remains effective for well-structured problems. As AI capabilities advance, benchmarks like BenchMAC will help distinguish genuine progress from incremental improvements on narrow tasks.

8.5 Limitations and Future Work

The current dataset is limited in scope. All nine instances come from a single repository with a small codebase, minimal dependencies, and no test infrastructure. This schematics-friendly dataset does not capture the full complexity of real-world migrations, where custom configurations, dependency conflicts, and non-standard architectures are common. The strong performance of the rule-based baseline should be interpreted with this limitation in mind.

Future work should expand the dataset to include multiple repositories with diverse characteristics: monorepo architectures, custom build systems, comprehensive test suites, and projects with complex dependency graphs. This expansion would test whether AI agents can demonstrate their flexibility advantage when rule-based tools struggle.

The evaluation currently uses build success as the primary metric. While this verifies syntactic and structural correctness, it provides no information about functional correctness, code quality, or adherence to best practices. Future versions could incorporate test execution, performance benchmarks, and code quality assessment through static analysis or LLM-based evaluation.

The benchmark focuses exclusively on Angular, but the methodology could extend to other frontend frameworks. Evaluating migrations for React, Vue, or Svelte would provide insights into whether capabilities generalize across frameworks and help develop more universal migration assistants.

Finally, the minimal scaffolding approach, while valuable for establishing a clean baseline, likely underestimates AI capabilities. Production systems would include optimized prompts, specialized tools, and retrieval-augmented generation. Research comparing minimal versus sophisticated scaffolding would help quantify the value of these enhancements.

8.6 Closing Remarks

Software modernization remains a persistent challenge in the industry. As frameworks evolve and security requirements change, teams must continuously upgrade their codebases. This work provides the first rigorous framework for evaluating AI assistance in this domain, specifically for Angular migrations.

The findings suggest that current AI systems can match deterministic tools on straightforward upgrade tasks, though at higher cost and with less reliability. The most promising path forward combines the strengths of both approaches: using rule-based tools for standard scenarios while leveraging AI flexibility for complex edge cases.

BenchMAC is released as an open-source project. We invite the research community to expand the dataset, refine the metrics, and extend the methodology to other frameworks. By establishing reproducible evaluation standards and sharing results, we can advance AI-assisted software engineering and develop tools that help developers maintain and modernize their systems.

Appendix A

Reproducible Evaluation Environment

A.1 Dockerfile for Reproducible Environment

The following Dockerfile defines the deterministic environment used for the instance migrating gothinkster/angular-realworld-example-app from v11→v12.

```
FROM node:16.20.2-bullseye-slim@sha256:503446
    c15c6236291222f8192513c2eb56a02a8949cbadf4fe78cce19815c734

# Set environment variables for CI
ENV CI=true \
    TZ=UTC \
    LANG=C.UTF-8 \
    NG_CLI_ANALYTICS=false \
    NPM_CONFIG_AUDIT=false \
    NPM_CONFIG_FUND=false

# Install system packages needed for deterministic Angular
    installs
# - git: for repository cloning
# - ca-certificates: for SSL certificate verification
RUN echo 'deb http://snapshot.debian.org/archive/debian/20250918
    T000000Z bullseye main contrib non-free' > /etc/apt/sources.
    list && \
    echo 'deb http://snapshot.debian.org/archive/debian-security
        /20250918T000000Z bullseye-security main contrib non-
        free' >> /etc/apt/sources.list && \
    echo 'Acquire::Check-Valid-Until "false";' > /etc/apt/apt.
        conf.d/99snapshot && \
    apt-get update && \
    apt-get install -y \
        curl \
        git \
        ca-certificates \
        --no-install-recommends && \
    rm -rf /var/lib/apt/lists/* && \
    apt-get clean

RUN node --version > /etc/benchmac-node-version && \
    npm --version > /etc/benchmac-npm-version

# Create project directory for the benchmark workspace
RUN mkdir -p /app/project
```

```
# Set working directory to the expected project path
WORKDIR /app/project

# Use a root-scoped npm cache so agents can install tooling as-
    needed
ENV NPM_CONFIG_CACHE=/root/.npm

# Download a history-free snapshot of the code at the specific
    commit
RUN curl -L https://github.com/gothinkster/angular-realworld-
    example-app/archive/4f29e0e.tar.gz \
     | tar -xz --strip-components=1

# Initialize git repo and create baseline commit for diffing
RUN git init && \
    git config --global --add safe.directory /app/project && \
    git config user.email "benchmark@benchmac.dev" && \
    git config user.name "BenchMAC Baseline" && \
    git add . && \
    git commit -m "baseline: gothinkster/angular-realworld-
        example-app@4f29e0e" && \
    git tag baseline

# Default command - idle state for external harness
CMD ["bash", "-lc", "sleep infinity"]
```

LISTING A.1: Representative Dockerfile for gothinkster/angular-realworld-example-app v11→v12

Appendix B

System Prompts

B.1 System Prompt for mini-swe-agent

System message

````
You are a helpful assistant that can interact with a computer.

Your response must contain exactly ONE bash code block with ONE command (or
    commands connected with && or ||).
Include a THOUGHT section before your command where you explain your
    reasoning process.
Format your response as shown in <format_example>.

<format_example>
Your reasoning and analysis here. Explain why you want to perform the action
    .

```bash
your_command_here
```
</format_example>

Failure to follow these rules will cause your response to be rejected.
````

First user message

```
Migrate the application from Angular {source_angular_version} to Angular {
    target_angular_version}.

You can execute bash commands and edit files to implement the necessary
    changes.

## Important Rules

1. Every response must contain exactly one action
2. The action must be enclosed in triple backticks
3. Directory or environment variable changes are not persistent. Every
    action is executed in a new subshell.
   However, you can prefix any action with `MY_ENV_VAR=MY_VALUE cd /path/to/
    working/dir && ...` or write/load environment variables from files
4. Avoid long-running or interactive commands (e.g., servers, watch modes,
    shells).
5. Prefer commands that finish on their own; add a time limit when uncertain
     (e.g., wrap with a timeout).
```

~~~~
## Submit and finish

1. When you are done, submit by running:
   ```bash
   echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT
   ```
   Do not combine it with any other command.

## Formatting your response

Here is an example of a correct response:

<example_response>
THOUGHT: I need to understand the structure of the repository first. Let me
    check what files are in the current directory to get a better
    understanding of the codebase.

```bash
ls -la
```
</example_response>

## Useful command examples

### Create a new file:

```bash
cat <<'EOF' > newfile.py
import numpy as np
hello = "world"
print(hello)
EOF
```

### Edit files with sed:

```bash
# Replace all occurrences
sed -i 's/old_string/new_string/g' filename.py

# Replace only first occurrence
sed -i 's/old_string/new_string/' filename.py

# Replace first occurrence on line 1
sed -i '1s/old_string/new_string/' filename.py

# Replace all occurrences in lines 1-10
sed -i '1,10s/old_string/new_string/g' filename.py
```

### View file content:

```bash
~~~~

```
# View specific lines with numbers
nl -ba filename.py | sed -n '10,20p'
```

Any other command you want to run

```bash
anything
```

Appendix C

Agent Traces and Run Details

C.1 Example Agent Reasoning and Execution Trace

Assistant message 1

````
THOUGHT: I'll update the Angular packages in package.json to Angular 19 by
    installing the Angular 19 packages and the likely required dev
    dependencies (CLI, devkit, TypeScript, RxJS). I'll back up package.json
    first, then run npm installs. I wrap the whole operation in a timeout (10
     minutes) to avoid accidentally running indefinitely.

```bash
timeout 600s bash -lc 'set -e; cp package.json package.json.bak || true; npm
     install @angular/core@^19.0.0 @angular/common@^19.0.0 @angular/compiler@
    ^19.0.0 @angular/compiler-cli@^19.0.0 @angular/forms@^19.0.0 @angular/
    router@^19.0.0 @angular/platform-browser@^19.0.0 @angular/platform-
    browser-dynamic@^19.0.0 --save && npm install --save-dev @angular/cli@
    ^19.0.0 @angular-devkit/build-angular@^19.0.0 typescript@~5.3 rxjs@^8 &&
    echo "Angular packages updated to ~19.0.0. Verify package.json, run ng
    update --all if needed, and run your test/build."'
```
````

User message 2

```
<returncode>1</returncode>
<output>
<stdout>

</stdout>
<stderr>
npm error code ERESOLVE
npm error ERESOLVE unable to resolve dependency tree
npm error
npm error While resolving: angular-realworld@0.0.0
npm error Found: zone.js@0.14.10
npm error node_modules/zone.js
npm error   zone.js@"~0.14.2" from the root project
npm error
npm error Could not resolve dependency:
npm error peer zone.js@"~0.15.0" from @angular/core@19.2.15
npm error node_modules/@angular/core
npm error   @angular/core@"^19.0.0" from the root project
npm error   peer @angular/core@"19.2.15" from @angular/common@19.2.15
npm error   node_modules/@angular/common
```

```
npm error     @angular/common@"^19.0.0" from the root project
npm error
npm error Fix the upstream dependency conflict, or retry
npm error this command with --force or --legacy-peer-deps
npm error to accept an incorrect (and potentially broken) dependency
    resolution.
npm error
npm error
npm error For a full report see:
npm error /root/.npm/_logs/2025-10-02T20_07_24_714Z-eresolve-report.txt
npm error A complete log of this run can be found in: /root/.npm/_logs
    /2025-10-02T20_07_24_714Z-debug-0.log

</stderr>
</output>
```

Assistant message 3

```
THOUGHT: The npm error shows a dependency conflict: Angular 19 requires zone
    .js ~0.15.0 but package.json currently pins zone.js to ~0.14.x. I'll back
     up package.json, update the zone.js entry to ~0.15.0, then install
    Angular 19 packages and required deps. I'll use --legacy-peer-deps to
    avoid resolution failures and wrap everything in a timeout to prevent
    long-running operations.

'''bash
timeout 600s bash -lc 'set -euo pipefail; cp package.json package.json.bak
    || true; if grep -q "\"zone.js\"" package.json; then sed -i -E "s/\"zone.
    js\":\s*\"[^\"]+\"/\"zone.js\":\"~0.15.0\"/g" package.json; else echo "
    zone.js entry not found in package.json"; fi; npm install --legacy-peer-
    deps @angular/core@^19.2.15 @angular/common@^19.2.15 @angular/compiler@
    ^19.2.15 @angular/compiler-cli@^19.2.15 @angular/forms@^19.2.15 @angular/
    router@^19.2.15 @angular/platform-browser@^19.2.15 @angular/platform-
    browser-dynamic@^19.2.15 rxjs@^8 tslib@^2.6 zone.js@~0.15.0 --save && npm
     install --legacy-peer-deps --save-dev @angular/cli@^19.2.15 @angular-
    devkit/build-angular@^19.2.15 typescript@~5.3 && echo "Angular 19 upgrade
     packages installed. Verify package.json and run your build/tests."'
'''
```

User message 4

```
<returncode>1</returncode>
<output>
<stdout>

</stdout>
<stderr>
npm error code ETARGET
npm error notarget No matching version found for rxjs@^8.
npm error notarget In most cases you or one of your dependencies are
    requesting
npm error notarget a package version that doesn't exist.
npm error A complete log of this run can be found in: /root/.npm/_logs
    /2025-10-02T20_07_48_711Z-debug-0.log
```

```
</stderr>
</output>
```

Assistant message 5

```
THOUGHT: The previous install failed because I requested rxjs@^8, which
    doesn’t exist. Angular 19 is compatible with RxJS 7.x, so I’ll install
    rxjs@^7.8.0 and the Angular 19 packages, update zone.js to ~0.15.0, and
    use --legacy-peer-deps to avoid peer resolution failures. I’ll back up
    package.json first and run installs with a timeout.

‘‘‘bash
timeout 600s bash -lc ’set -euo pipefail; cp package.json package.json.bak
    || true; npm install --legacy-peer-deps @angular/core@^19.2.15 @angular/
    common@^19.2.15 @angular/compiler@^19.2.15 @angular/compiler-cli@^19.2.15
     @angular/forms@^19.2.15 @angular/router@^19.2.15 @angular/platform-
    browser@^19.2.15 @angular/platform-browser-dynamic@^19.2.15 rxjs@^7.8.0
    tslib@^2.6 zone.js@~0.15.0 --save && npm install --legacy-peer-deps --
    save-dev @angular/cli@^19.2.15 @angular-devkit/build-angular@^19.2.15
    typescript@~5.3 && echo "Angular 19 packages installed (rxjs@^7.8.0).
    Verify package.json and run your build/tests."’
‘‘‘
```

User message 6

```
<returncode>0</returncode>
<output>
<stdout>

added 1024 packages in 46s

added 113 packages, removed 145 packages, and changed 325 packages in 49s
Angular 19 packages installed (rxjs@^7.8.0). Verify package.json and run
    your build/tests.

</stdout>
<stderr>
npm warn deprecated @types/marked@6.0.0: This is a stub types definition.
    marked provides its own type definitions, so you do not need this
    installed.

</stderr>
</output>
```

Assistant message 7

```
THOUGHT: npm successfully installed Angular 19 and updated dependencies (
    zone.js, rxjs). To finish the task and submit the final output as
    instructed, I will run the required completion command.

‘‘‘bash
echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT
‘‘‘
```

User message 8

```
COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT
```

Bibliography

[1] A. Terekhov and C. Verhoef, "The realities of language conversions," *IEEE Software*, vol. 17, no. 6, pp. 111–124, Nov. 2000, ISSN: 1937-4194. DOI: 10.1109/52.895180. Accessed: Sep. 2, 2025.

[2] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, *Unsupervised Translation of Programming Languages*, Sep. 2020. DOI: 10.48550/arXiv.2006.03511. arXiv: 2006.03511 [cs]. Accessed: Sep. 2, 2025.

[3] J. Yang et al., *SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering*, Nov. 2024. DOI: 10.48550/arXiv.2405.15793. arXiv: 2405.15793 [cs]. Accessed: Sep. 2, 2025.

[4] M. Chen et al., *Evaluating Large Language Models Trained on Code*, Jul. 2021. DOI: 10.48550/arXiv.2107.03374. arXiv: 2107.03374 [cs]. Accessed: Sep. 2, 2025.

[5] J. Austin et al., *Program Synthesis with Large Language Models*, Aug. 2021. DOI: 10.48550/arXiv.2108.07732. arXiv: 2108.07732 [cs]. Accessed: Sep. 2, 2025.

[6] C. E. Jimenez et al., *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* Nov. 2024. DOI: 10.48550/arXiv.2310.06770. arXiv: 2310.06770 [cs]. Accessed: Sep. 2, 2025.

[7] N. Chowdhury et al., *Introducing SWE-bench Verified*, Aug. 2024. Accessed: Sep. 2, 2025.

[8] S. Lu et al., *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation*, Mar. 2021. DOI: 10.48550/arXiv.2102.04664. arXiv: 2102.04664 [cs]. Accessed: Sep. 2, 2025.

[9] W. Yan, Y. Tian, Y. Li, Q. Chen, and W. Wang, *CodeTransOcean: A Comprehensive Multilingual Benchmark for Code Translation*, Oct. 2023. DOI: 10.48550/arXiv.2310.04951. arXiv: 2310.04951 [cs]. Accessed: Sep. 2, 2025.

[10] T. Wu et al., *VersiCode: Towards Version-controllable Code Generation*, Oct. 2024. DOI: 10.48550/arXiv.2406.07411. arXiv: 2406.07411 [cs]. Accessed: Sep. 2, 2025.

[11] N. Islah et al., *GitChameleon: Unmasking the Version-Switching Capabilities of Code Generation Models*, Nov. 2024. DOI: 10.48550/arXiv.2411.05830. arXiv: 2411.05830 [cs]. Accessed: Sep. 4, 2025.

[12] F. Reyes, Y. Gamage, G. Skoglund, B. Baudry, and M. Monperrus, "BUMP: A Benchmark of Reproducible Breaking Dependency Updates," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2024, pp. 159–170. DOI: 10.1109/SANER60148.2024.00024. arXiv: 2401.09906 [cs]. Accessed: Sep. 4, 2025.

[13] M. Islam, A. K. Jha, S. Nadi, and I. Akhmetov, "PyMigBench: A Benchmark for Python Library Migration," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, May 2023, pp. 511–515. DOI: 10.1109/MSR59073.2023.00075. Accessed: Sep. 11, 2025.

[14] S. Mallet, J. El Khoury, and E. Egyed-Zsigmond, "Repousser les limites des benchmarks actuels pour une évaluation réaliste des LLMs en migration de code," in *Actes de CORIA-TALN-RJCRI-RECITAL 2025. Actes Des 32ème Conférence Sur Le Traitement Automatique Des Langues Naturelles (TALN), Volume 1 : Articles Scientifiques Originaux*, Marseille, France: Association pour le Traitement Automatique des Langues, Jun. 2025, pp. 682–696.

[15] X. Hou et al., *Large Language Models for Software Engineering: A Systematic Literature Review*, Apr. 2024. DOI: 10.48550/arXiv.2308.10620. arXiv: 2308.10620 [cs]. Accessed: Sep. 5, 2025.

[16] Q. Zhang et al., *A Systematic Literature Review on Large Language Models for Automated Program Repair*, May 2024. DOI: 10.48550/arXiv.2405.01466. arXiv: 2405.01466 [cs]. Accessed: Sep. 5, 2025.

[17] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, *A Survey on Large Language Models for Code Generation*, Nov. 2024. DOI: 10.48550/arXiv.2406.00515. arXiv: 2406.00515 [cs]. Accessed: Sep. 2, 2025.

[18] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, and H. Chen, *From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future*, Apr. 2025. DOI: 10.48550/arXiv.2408.02479. arXiv: 2408.02479 [cs]. Accessed: Sep. 5, 2025.

[19] "Large language models for code completion: A systematic literature review," *Computer Standards & Interfaces*, vol. 92, p. 103 917, Mar. 2025, ISSN: 0920-5489. DOI: 10.1016/j.csi.2024.103917. Accessed: Sep. 5, 2025.

[20] Y. Dong et al., *A Survey on Code Generation with LLM-based Agents*, Jul. 2025. DOI: 10.48550/arXiv.2508.00083. arXiv: 2508.00083 [cs]. Accessed: Sep. 5, 2025.

[21] A. Plaat, M. van Duijn, N. van Stein, M. Preuss, P. van der Putten, and K. J. Batenburg, *Agentic Large Language Models, a survey*, Apr. 2025. DOI: 10.48550/arXiv.2503.23037. arXiv: 2503.23037 [cs]. Accessed: Sep. 5, 2025.

[22] Z. Feng et al., *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*, Sep. 2020. DOI: 10.48550/arXiv.2002.08155. arXiv: 2002.08155 [cs]. Accessed: Oct. 9, 2025.

[23] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*, Sep. 2021. DOI: 10.48550/arXiv.2109.00859. arXiv: 2109.00859 [cs]. Accessed: Sep. 2, 2025.

[24] N. Friedman, *Introducing GitHub Copilot: Your AI pair programmer*, Jun. 2021. Accessed: Oct. 9, 2025.

[25] D. Fried et al., *InCoder: A Generative Model for Code Infilling and Synthesis*, Apr. 2023. DOI: 10.48550/arXiv.2204.05999. arXiv: 2204.05999 [cs]. Accessed: Sep. 2, 2025.

[26] E. Nijkamp et al., *CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis*, Feb. 2023. DOI: 10.48550/arXiv.2203.13474. arXiv: 2203.13474 [cs]. Accessed: Oct. 9, 2025.

[27] R. Li et al., *StarCoder: May the source be with you!* Dec. 2023. DOI: 10.48550/arXiv.2305.06161. arXiv: 2305.06161 [cs]. Accessed: Oct. 9, 2025.

[28] A. Lozhkov et al., *StarCoder 2 and The Stack v2: The Next Generation*, Feb. 2024. DOI: 10.48550/arXiv.2402.19173. arXiv: 2402.19173 [cs]. Accessed: Oct. 9, 2025.

[29] S. Yao et al., *ReAct: Synergizing Reasoning and Acting in Language Models*, Mar. 2023. DOI: 10.48550/arXiv.2210.03629. arXiv: 2210.03629 [cs]. Accessed: Oct. 9, 2025.

[30] *Codestral | Mistral AI*, https://mistral.ai/news/codestral. Accessed: Oct. 9, 2025.

[31] *Devstral | Mistral AI*, https://mistral.ai/news/devstral. Accessed: Oct. 9, 2025.

[32] *Qwen3-Coder: Agentic Coding in the World*, https://qwen.ai/blog?id=d927d7d2e59d059045ce758ded34f98c0186d2d7. Accessed: Oct. 9, 2025.

[33] *Introducing the Model Context Protocol*. Accessed: Oct. 9, 2025. [Online]. Available: https://www.anthropic.com/news/model-context-protocol.

[34] Z. Luo et al., *MCP-Universe: Benchmarking Large Language Models with Real-World Model Context Protocol Servers*, Aug. 2025. DOI: 10.48550/arXiv.2508.14704. arXiv: 2508.14704 [cs]. Accessed: Sep. 5, 2025.

[35] P. Liang et al., *Holistic Evaluation of Language Models*, Oct. 2023. DOI: 10.48550/arXiv.2211.09110. arXiv: 2211.09110 [cs]. Accessed: Sep. 2, 2025.

[36] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, *GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding*, Feb. 2019. DOI: 10.48550/arXiv.1804.07461. arXiv: 1804.07461 [cs]. Accessed: Oct. 9, 2025.

[37] A. Wang et al., *SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems*, Feb. 2020. DOI: 10.48550/arXiv.1905.00537. arXiv: 1905.00537 [cs]. Accessed: Oct. 9, 2025.

[38] D. Hendrycks et al., *Measuring Massive Multitask Language Understanding*, Jan. 2021. DOI: 10.48550/arXiv.2009.03300. arXiv: 2009.03300 [cs]. Accessed: Oct. 9, 2025.

[39] K. Cobbe et al., *Training Verifiers to Solve Math Word Problems*, Nov. 2021. DOI: 10.48550/arXiv.2110.14168. arXiv: 2110.14168 [cs]. Accessed: Sep. 2, 2025.

[40] S. Lin, J. Hilton, and O. Evans, *TruthfulQA: Measuring How Models Mimic Human Falsehoods*, May 2022. DOI: 10.48550/arXiv.2109.07958. arXiv: 2109.07958 [cs]. Accessed: Oct. 9, 2025.

[41] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02*, Philadelphia, Pennsylvania: Association for Computational Linguistics, 2001, p. 311. DOI: 10.3115/1073083.1073135. Accessed: Oct. 9, 2025.

[42] S. Ott, A. Barbosa-Silva, K. Blagec, J. Brauner, and M. Samwald, "Mapping global dynamics of benchmark creation and saturation in artificial intelligence," *Nature Communications*, vol. 13, no. 1, p. 6793, Nov. 2022, ISSN: 2041-1723. DOI: 10.1038/s41467-022-34591-0. arXiv: 2203.04592 [cs]. Accessed: Sep. 2, 2025.

[43] *Killed by LLM*, https://r0bk.github.io/killedbyllm/. Accessed: Oct. 9, 2025.

[44] N. Jain et al., *LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code*, Jun. 2024. DOI: 10.48550/arXiv.2403.07974. arXiv: 2403.07974 [cs]. Accessed: Sep. 2, 2025.

[45] C. S. Xia, Y. Deng, and L. Zhang, *Top Leaderboard Ranking = Top Coding Proficiency, Always? EvoEval: Evolving Coding Benchmarks via LLM*, Mar. 2024. DOI: 10.48550/arXiv.2403.19114. arXiv: 2403.19114 [cs]. Accessed: Sep. 2, 2025.

[46] C. Xu, S. Guan, D. Greene, and M.-T. Kechadi, *Benchmark Data Contamination of Large Language Models: A Survey*, Jun. 2024. DOI: 10.48550/arXiv.2406.04244. arXiv: 2406.04244 [cs]. Accessed: Sep. 4, 2025.

[47] S. Chen et al., *Recent Advances in Large Langauge Model Benchmarks against Data Contamination: From Static to Dynamic Evaluation*, Feb. 2025. DOI: 10.48550/arXiv.2502.17521. arXiv: 2502.17521 [cs]. Accessed: Sep. 4, 2025.

[48] M. Riddell, A. Ni, and A. Cohan, *Quantifying Contamination in Evaluating Code Generation Capabilities of Language Models*, Mar. 2024. DOI: 10.48550/arXiv.2403.04811. arXiv: 2403.04811 [cs]. Accessed: Sep. 4, 2025.

[49] N. Alzahrani et al., *When Benchmarks are Targets: Revealing the Sensitivity of Large Language Model Leaderboards*, Jul. 2024. DOI: 10.48550/arXiv.2402.01781. arXiv: 2402.01781 [cs]. Accessed: Sep. 2, 2025.

[50] S. Biderman et al., *Lessons from the Trenches on Reproducible Evaluation of Language Models*, May 2024. DOI: 10.48550/arXiv.2405.14782. arXiv: 2405.14782 [cs]. Accessed: Oct. 9, 2025.

[51] G. Zhao et al., *Large Language Models Badly Generalize across Option Length, Problem Types, and Irrelevant Noun Replacements*, Sep. 2025. DOI: 10.48550/arXiv.2502.12459. arXiv: 2502.12459 [cs]. Accessed: Oct. 9, 2025.

[52] V. Gupta, D. Pantoja, C. Ross, A. Williams, and M. Ung, *Changing Answer Order Can Decrease MMLU Accuracy*, Nov. 2024. DOI: 10.48550/arXiv.2406.19470. arXiv: 2406.19470 [cs]. Accessed: Oct. 9, 2025.

[53] J. He, M. Rungta, D. Koleczek, A. Sekhon, F. X. Wang, and S. Hasan, *Does Prompt Formatting Have Any Impact on LLM Performance?* Nov. 2024. DOI: 10.48550/arXiv.2411.10541. arXiv: 2411.10541 [cs]. Accessed: Oct. 9, 2025.

[54] S. Zhang et al., *NaturalCodeBench: Examining Coding Performance Mismatch on HumanEval and Natural User Prompts*, May 2024. DOI: 10.48550/arXiv.2405.04520. arXiv: 2405.04520 [cs]. Accessed: Sep. 2, 2025.

[55] S. Shashidhar, C. Fourrier, A. Lozovskaya, T. Wolf, G. Tur, and D. Hakkani-Tür, "Yourbench: Dynamic Evaluation Set Generation with LLMs," in *Second Conference on Language Modeling*, Aug. 2025. Accessed: Oct. 9, 2025.

[56] Q. Zhao et al., *MMLU-CF: A Contamination-free Multi-task Language Understanding Benchmark*, Dec. 2024. DOI: 10.48550/arXiv.2412.15194. arXiv: 2412.15194 [cs]. Accessed: Oct. 9, 2025.

[57] J. Vendrow, E. Vendrow, S. Beery, and A. Madry, *Do Large Language Model Benchmarks Test Reliability?* Feb. 2025. DOI: 10.48550/arXiv.2502.03461. arXiv: 2502.03461 [cs]. Accessed: Oct. 9, 2025.

[58] R. Jiang et al., *A Survey on Human Preference Learning for Large Language Models*, Jun. 2024. DOI: 10.48550/arXiv.2406.11191. arXiv: 2406.11191 [cs]. Accessed: Oct. 9, 2025.

[59] R. Wang, J. Guo, C. Gao, G. Fan, C. Y. Chong, and X. Xia, "Can LLMs Replace Human Evaluators? An Empirical Study of LLM-as-a-Judge in Software Engineering," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 1955–1977, Jun. 2025, ISSN: 2994-970X. DOI: 10.1145/3728963. arXiv: 2502.06193 [cs]. Accessed: Sep. 2, 2025.

[60] X. Yue et al., *MMMU: A Massive Multi-discipline Multimodal Understanding and Reasoning Benchmark for Expert AGI*, Jun. 2024. DOI: 10.48550/arXiv.2311.16502. arXiv: 2311.16502 [cs]. Accessed: Oct. 9, 2025.

[61] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, *Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation*, Oct. 2023. DOI: 10.48550/arXiv.2305.01210. arXiv: 2305.01210 [cs]. Accessed: Sep. 2, 2025.

[62] T. Liu, C. Xu, and J. McAuley, *RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems*, Oct. 2023. DOI: 10.48550/arXiv.2306.03091. arXiv: 2306.03091 [cs]. Accessed: Sep. 2, 2025.

[63] Y. Ding et al., *CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion*, Nov. 2023. DOI: 10.48550/arXiv.2310.11248. arXiv: 2310.11248 [cs]. Accessed: Sep. 2, 2025.

[64] J. Yang et al., *ExecRepoBench: Multi-level Executable Code Completion Evaluation*, Dec. 2024. DOI: 10.48550/arXiv.2412.11990. arXiv: 2412.11990 [cs]. Accessed: Sep. 2, 2025.

[65] S. Ren et al., *CodeBLEU: A Method for Automatic Evaluation of Code Synthesis*, Sep. 2020. DOI: 10.48550/arXiv.2009.10297. arXiv: 2009.10297 [cs]. Accessed: Sep. 2, 2025.

[66] X. Du et al., *ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation*, Aug. 2023. DOI: 10.48550/arXiv.2308.01861. arXiv: 2308.01861 [cs]. Accessed: Sep. 2, 2025.

[67] K. Jain, G. Synnaeve, and B. Rozière, *TestGenEval: A Real World Unit Test Generation and Test Completion Benchmark*, Mar. 2025. DOI: 10.48550/arXiv.2410.00752. arXiv: 2410.00752 [cs]. Accessed: Sep. 2, 2025.

[68] W. Li et al., *FEA-Bench: A Benchmark for Evaluating Repository-Level Code Generation for Feature Implementation*, Jun. 2025. DOI: 10.48550/arXiv.2503.06680. arXiv: 2503.06680 [cs]. Accessed: Sep. 2, 2025.

[69] Y. Lai et al., *DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation*, Nov. 2022. DOI: 10.48550/arXiv.2211.11501. arXiv: 2211.11501 [cs]. Accessed: Sep. 2, 2025.

[70] M. Bhatt et al., *Purple Llama CyberSecEval: A Secure Coding Benchmark for Language Models*, Dec. 2023. DOI: 10.48550/arXiv.2312.04724. arXiv: 2312.04724 [cs]. Accessed: Sep. 2, 2025.

[71] V. Nitin, R. Krishna, L. L. do Valle, and B. Ray, *C2SaferRust: Transforming C Projects into Safer Rust with NeuroSymbolic Techniques*, Jan. 2025. DOI: 10.48550/arXiv.2501.14257. arXiv: 2501.14257 [cs]. Accessed: Sep. 2, 2025.

[72] A. R. Ibrahimzada et al., "AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 2454–2476, Jun. 2025, ISSN: 2994-970X. DOI: 10.1145/3729379. arXiv: 2410.24117 [cs]. Accessed: Sep. 2, 2025.

[73] RamosDaniel, LynceInês, ManquinhoVasco, MartinsRuben, and L. GouesClaire, "BatFix: Repairing language model-based transpilation," *ACM Transactions on Software Engineering and Methodology*, Jun. 2024. DOI: 10.1145/3658668. Accessed: Sep. 2, 2025.

[74] S. Froimovich, R. Gal, W. Ibraheem, and A. Ziv, *Quality Evaluation of COBOL to Java Code Transformation*, Jul. 2025. DOI: 10.48550/arXiv.2507.23356. arXiv: 2507.23356 [cs]. Accessed: Sep. 8, 2025.

[75] A. Malyala, K. Zhou, B. Ray, and S. Chakraborty, *On ML-Based Program Translation: Perils and Promises*, Feb. 2023. DOI: 10.48550/arXiv.2302.10812. arXiv: 2302.10812 [cs]. Accessed: Sep. 2, 2025.

[76] P. Jana, P. Jha, H. Ju, G. Kishore, A. Mahajan, and V. Ganesh, "CoTran: An LLM-based Code Translator using Reinforcement Learning with Feedback from Compiler and Symbolic Execution," in Oct. 2024. DOI: 10.3233/FAIA240968. arXiv: 2306.06755 [cs]. Accessed: Sep. 2, 2025.

[77] X. Chen, C. Liu, and D. Song, *Tree-to-tree Neural Networks for Program Translation*, Oct. 2018. DOI: 10.48550/arXiv.1802.03691. arXiv: 1802.03691 [cs]. Accessed: Sep. 2, 2025.

[78] Q. Zheng et al., *CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X*, Jul. 2024. DOI: 10.48550/arXiv.2303.17568. arXiv: 2303.17568 [cs]. Accessed: Sep. 2, 2025.

[79] J. Pan, A. Sadé, J. Kim, E. Soriano, G. Sole, and S. Flamant, *SteloCoder: A Decoder-Only LLM for Multi-Language to Python Code Translation*, Dec. 2023. DOI: 10.48550/arXiv.2310.15539. arXiv: 2310.15539 [cs]. Accessed: Sep. 2, 2025.

[80] Z. Yuan, W. Chen, H. Wang, K. Yu, X. Peng, and Y. Lou, *TRANSAGENT: An LLM-Based Multi-Agent System for Code Translation*, Oct. 2024. DOI: 10.48550/arXiv.2409.19894. arXiv: 2409.19894 [cs]. Accessed: Sep. 2, 2025.

[81] R. Pan et al., "Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Apr. 2024, pp. 1–13. DOI: 10.1145/3597503.3639226. arXiv: 2308.03109 [cs]. Accessed: Sep. 2, 2025.

[82] J. Li et al., *LONGCODEU: Benchmarking Long-Context Language Models on Long Code Understanding*, Mar. 2025. DOI: 10.48550/arXiv.2503.04359. arXiv: 2503.04359 [cs]. Accessed: Sep. 2, 2025.

[83] P. Xue et al., *ClassEval-T: Evaluating Large Language Models in Class-Level Code Translation*, Apr. 2025. DOI: 10.48550/arXiv.2411.06145. arXiv: 2411.06145 [cs]. Accessed: Sep. 2, 2025.

[84] Y. Wang et al., *RepoTransBench: A Real-World Benchmark for Repository-Level Code Translation*, Dec. 2024. DOI: 10.48550/arXiv.2412.17744. arXiv: 2412.17744 [cs]. Accessed: Sep. 2, 2025.

[85] X. Zhang et al., *Skeleton-Guided-Translation: A Benchmarking Framework for Code Repository Translation with Fine-Grained Quality Evaluation*, Jan. 2025. DOI: 10.48550/arXiv.2501.16050. arXiv: 2501.16050 [cs]. Accessed: Sep. 2, 2025.

[86] C. Ziftci, S. Nikolov, A. Sjövall, B. Kim, D. Codecasa, and M. Kim, "Migrating Code At Scale With LLMs At Google," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, Jun. 2025, pp. 162–173. DOI: 10.1145/3696630.3728542. arXiv: 2504.09691 [cs]. Accessed: Sep. 2, 2025.

[87] L. Liang et al., *RustEvo^2: An Evolving Benchmark for API Evolution in LLM-based Rust Code Generation*, Mar. 2025. DOI: 10.48550/arXiv.2503.16922. arXiv: 2503.16922 [cs]. Accessed: Sep. 2, 2025.

[88] *OpenRewrite by Moderne | Large Scale Automated Refactoring*, https://docs.openrewrite.org/. Accessed: Oct. 10, 2025.

[89] S. Nikolov et al., *How is Google using AI for internal code migrations?* Jan. 2025. DOI: 10.48550/arXiv.2501.06972. arXiv: 2501.06972 [cs]. Accessed: Sep. 2, 2025.

[90] S. Kuhar et al., *LibEvolutionEval: A Benchmark and Study for Version-Specific Code Generation*, Nov. 2024. DOI: 10.48550/arXiv.2412.04478. arXiv: 2412.04478 [cs]. Accessed: Sep. 4, 2025.

[91] L. Fruntke and J. Krinke, "Automatically Fixing Dependency Breaking Changes," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 2146–2168, Jun. 2025, ISSN: 2994-970X. DOI: 10.1145/3729366. Accessed: Sep. 2, 2025.

[92] X. Xu, C. Zhu, and Y. Li, *CompSuite: A Dataset of Java Library Upgrade Incompatibility Issues*, May 2023. DOI: 10.48550/arXiv.2305.08671. arXiv: 2305.08671 [cs]. Accessed: Sep. 4, 2025.

[93] K. Jiang, B. Jin, and P. Nie, "CoUpJava: A Dataset of Code Upgrade Histories in Open-Source Java Repositories," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, Ottawa, ON, Canada: IEEE, Apr. 2025, pp. 441–445, ISBN: 979-8-3315-0183-9. DOI: 10.1109/MSR66628.2025.00075. Accessed: Sep. 2, 2025.

[94] L. Liu et al., *MigrationBench: Repository-Level Code Migration Benchmark from Java 8*, May 2025. DOI: 10.48550/arXiv.2505.09569. arXiv: 2505.09569 [cs]. Accessed: Sep. 2, 2025.

[95] *SWE-agent/mini-swe-agent: The 100 line AI agent that solves GitHub issues or helps you in your command line.* Accessed: Oct. 10, 2025.

[96] N. Mündler, M. N. Müller, J. He, and M. Vechev, *SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents*, Feb. 2025. DOI: 10.48550/arXiv.2406.12952. arXiv: 2406.12952 [cs]. Accessed: Sep. 2, 2025.

[97] Anthropic, *Building with Extended Thinking*, https://docs.claude.com/en/docs/build-with-claude/extended-thinking. Accessed: Oct. 11, 2025.

[98] X. Wang et al., *OpenHands: An Open Platform for AI Software Developers as Generalist Agents*, Apr. 2025. DOI: 10.48550/arXiv.2407.16741. arXiv: 2407.16741 [cs]. Accessed: Sep. 2, 2025.

[99] J. Yang et al., *SWE-smith: Scaling Data for Software Engineering Agents*, May 2025. DOI: 10.48550/arXiv.2504.21798. arXiv: 2504.21798 [cs]. Accessed: Sep. 2, 2025.

[100] J. Hejderup and G. Gousios, "Can We Trust Tests To Automate Dependency Updates? A Case Study of Java Projects," *Journal of Systems and Software*, vol. 183, p. 111 097, Jan. 2022, ISSN: 01641212. DOI: 10.1016/j.jss.2021.111097. arXiv: 2109.11921 [cs]. Accessed: Sep. 4, 2025.

[101] B. Atil et al., *Non-Determinism of "Deterministic" LLM Settings*, Apr. 2025. DOI: 10.48550/arXiv.2408.04667. arXiv: 2408.04667 [cs]. Accessed: Oct. 8, 2025.

[102] *OpenAI API - Reproducible outputs*. Accessed: Oct. 8, 2025. [Online]. Available: https://platform.openai.com/docs/advanced-usage/%5C#reproducible-outputs.

[103] *Experiment with parameter values | Vertex AI*, https://cloud.google.com/vertex-ai/generative-ai/docs/learn/prompts/adjust-parameter-values. Accessed: Oct. 8, 2025.